EASTER: Learning to Split Transformers at the Edge Robustly

Xiaotian Guo, Quan Jiang, Yixian Shen, Andy D. Pimentel, *Senior Member, IEEE*, and Todor Stefanov, *Member, IEEE*

***Abstract*—Prevalent large transformer models present significant computational challenges for resource-constrained devices at the Edge. While distributing the workload of deep learning models across multiple edge devices has been extensively studied, these works typically overlook the impact of failures of edge devices. Unpredictable failures, due to, e.g., connectivity issues or discharged batteries, can compromise the reliability of inference serving at the Edge. In this article, we introduce a novel methodology, called *EASTER*, designed to learn robust distribution strategies for transformer models against device failures that consider the tradeoff between robustness (i.e., maintaining model functionality against failures) and resource utilization (considering memory usage and computations). We evaluate *EASTER* with three representative transformers—ViT, GPT-2, and Vicuna—under device failures. Our results demonstrate *EASTER*'s efficiency in memory usage, and possible end-to-end latency improvement for inference across multiple edge devices while preserving model accuracy as much as possible under device failures.**

***Index Terms*—Deep learning (DL), design space exploration (DSE), distributed inference, embedded system, robustness.**

I. Introduction

As ARTIFICIAL intelligence (AI) continues to evolve rapidly, transformer models are increasingly prevalent in various applications [1]. Advanced pretrained models, such as BERT and GPT-4 [2], have spurred a range of novel tools, including Copilot and ChatGPT. Typically, these models are executed on high-performance clusters with hundreds of GPUs, available as cloud services. However, the rise of Internet of Things (IoT) devices has driven a demand for deploying transformer-based tools at the Edge. Deploying these tools on edge or IoT devices offers significant advantages in terms of efficiency, security, and privacy. For example, a network of IoT devices in smart healthcare systems [3] within a hospital or a home setting, such as wearable health monitors, bedside monitors, and portable diagnostic devices, are equipped with sensors to collect vital signs and patient data in real time. By deploying deep neural networks, like transformer models, directly onto these devices, the system can locally analyze data, make immediate health assessments, or predict medical events without the need to send or store sensitive patient data in centralized cloud servers, thus enhancing user privacy and data security. This also allows for faster, potentially life-saving decisions by reducing the latency associated with data being sent to the cloud and the cloud processing of the data. However, deploying transformer-based tools at the Edge presents a significant challenge for edge or IoT devices due to the intensive computational and memory requirements of transformer models. For instance, the Vicuna-13B chatbot [4] requires 26 GB of memory for the model parameters and substantial computational resources for inference.

While constructing lightweight transformer models from larger counterparts using methods like model compression [5] or neural architecture search [6] is one approach, it often leads to a reduced performance/accuracy score and resource-intensive retraining of the newly derived models. In response, research has focused on fully distributing transformer inference across multiple edge devices without resorting to model compression or cloud servers. Methods like model partitioning [7] and data partitioning [8] have been explored to bridge the gap between limited edge device resources and the demands of large transformer models. Furthermore, by distributing the computational workload of a transformer across multiple edge devices, the system can operate more energy-efficiently, making it both cost-effective and sustainable for long-term deployment. However, these methods generally assume continuous availability of all participating devices, which is often unrealistic due to potential device unavailability or failures.

Addressing this issue, our study emphasizes the need for robust partitioning methods for distributed transformer inference. Distributed inference across multiple devices offers a promising solution for handling large transformer models (e.g., Llama [9]) that exceed the memory capacities of individual devices, such as IoT devices, smart surveillance cameras, user laptops, etc. Existing frameworks, like Alpa [10] and DeepSpeed [11], effectively support distributed large language model (LLM) training, but do not address at all robust distributed inference on edge devices and do not cater for resource heterogeneity in edge systems or IoT settings.

Therefore, this article introduces a novel methodology, called *EASTER*, designed to learn robust distribution strategies for transformers that ensure functional inference and

Manuscript received 31 July 2024; accepted 1 August 2024. This article was presented at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems (CASES) 2024 and appeared as part of the ESWEEK-TCAD Special Issue. This article was recommended by Associate Editor S. Dailey. *(Corresponding author: Xiaotian Guo.)*

Xiaotian Guo is with the Informatics Institute, University of Amsterdam, 1098 XH Amsterdam, The Netherlands, and also with the Leiden Institute of Advanced Computer Science, Leiden University, 2333 CA Leiden, The Netherlands (e-mail: x.guo3@uva.nl).

Quan Jiang is with the Computer Science and Technology Department, Nanjing Agricultural University, Nanjing 210095, China.

Yixian Shen and Andy D. Pimentel are with the Informatics Institute, University of Amsterdam, 1098 XH Amsterdam, The Netherlands.

Todor Stefanov is with the Leiden Institute of Advanced Computer Science, Leiden University, 2333 CA Leiden, The Netherlands.

Digital Object Identifier 10.1109/TCAD.2024.3438995

maintain close-to-original results under potential device failures. Learning such optimal strategies to distribute millions of neurons is challenging because a vast and complex design space needs to be explored. Typical transformer-based models consist of several stacked encoder and decoder blocks. The embedding dimension within each block, which represents the size of vectors used to encode images, words, or tokens, usually exceeds 100. For example, if the embedded dimension of an encoder block is 768 [12], and we consider each dimension-related connection as a neuron, then the encoder block has 768 neurons. If we want to distribute these 768 neurons over four devices evenly, the exact number of possible distributions is $\binom{768}{192} \times \binom{576}{192} \times \binom{384}{192}$. The vast number of potential possibilities to distribute just one encoder block across multiple devices is almost unimaginable, let alone when considering the distribution of multiple blocks in large transformer models. There is a critical need to explore this extensive design space efficiently to identify a neuron distribution strategy that maintains performance against potential device failures to ensure the robustness and reliability of the distributed system.

For different distribution strategies (design points) of transformers in the vast space, our algorithm is designed to efficiently and quickly explore and identify optimal design points, enabling robust and memory-efficient splitting of transformer models across multiple devices. We first narrow down the design space by considering the neuron importance in the transformer layers, as this assessment allows us to group neurons within each layer, significantly reducing their distribution complexity. Further, we achieve this by adaptively and recursively splitting the design space into several subspaces and learning the expected rewards associated with different subspaces. To this end, we have developed a variant of the upper confidence bounds applied to trees (UCT) algorithm [13], aiming to enhance splitting and prioritizing subspaces with the highest potential for robustness. By navigating and sampling both the most and potential promising subspaces rather than the entire vast space, our approach enhances search efficiency, while balancing exploration and exploitation to avoid the pitfalls of local optima. The final Pareto points/solutions offer an optimal blend of robustness against device failures and operational efficiency regarding computation and memory.

We also automate the process of dividing transformer models for distributed computing by converting them into a unified neural network intermediate representation (IR). This step is followed by automated code generation and the subsequent deployment of the models across multiple edge devices. Our experimental results demonstrate that the system configurations identified as Pareto-optimal points through the aforementioned design space exploration (DSE) method not only maintain system robustness but also achieve a notable reduction in memory usage. Furthermore, these configurations reduce the end-to-end inference latency for very large transformer models, demonstrating the effectiveness of our approach in optimizing both the performance and efficiency of distributed deep learning (DL) systems.

Our main novel contributions are summarized as follows.

1) A novel UCT-based DSE algorithm is proposed that efficiently narrows down the vast design space, facilitating the discovery of effective model partitioning strategies for robust transformer distribution that balance performance and resource usage.
2) By empirical validation, we demonstrate the efficacy of our *EASTER* methodology using typical transformers like ViT-16 [12], GPT2-Large [14], and Vicuna-7B [4], showcasing resilient model performance in image and common reasoning tasks.
3) We provide the first implementation of an end-to-end tool for splitting transformer models and also validate the advantages of distributed inference in terms of end-to-end inference latency and memory utilization compared to single-device inference.

II. Related Work

The proliferation of transformer models in various applications has necessitated their adaptation beyond the confines of powerful cloud computing resources, directing significant research interest toward edge deployments. This section reviews pertinent literature across three main themes relevant to our work on *EASTER*: 1) adaptation of large transformer models for resource-constrained edge devices; 2) resilience against device failures; and 3) efficiency in DSE.

A. Adaptation of Transformer Models for Edge Constraints

The push toward deploying AI capabilities at the edge, driven by privacy concerns, latency reduction, and energy efficiency, has seen approaches like model compression [15], [16], [17] and neural architecture search [18], [19], [20], [21] gain prominence. Such approaches can compress original transformer models to smaller models for resource-constrained devices. However, they typically require iterative retraining and may result in accuracy loss. Another approach is to deploy the original models onto distributed edge computing platforms, such as health care systems [22], smart home systems [23], etc., in order to leverage all available resources collaboratively. Traditional layer and data partitioning methods like [7] and [24] are applied to fully distribute the workload of a large convolution neural network or a transformer-based model among multiple edge devices, thereby reducing the required computation resources of edge devices [25]. It involves breaking down a model's computational graph into smaller, manageable parts that can be processed in parallel across multiple devices. This is particularly challenging in edge computing due to the heterogeneous nature of devices and their limited computational capabilities. Model parallelism techniques like AlpaServe [10] developed for homogeneous data center clusters are targets for multibatch inference which would perform poorly for single batches in heterogeneous edge environments. PipeEdge [24] partitions a neural network model into multiple pipeline stages and applies a dynamic programming (DP) algorithm to determine the optimal partition scheduling strategy for heterogeneous computation and communication. However, all of the aforementioned approaches and methods assume that the involved edge computing devices and communication links between them are always available

and work properly. In contrast, our partitioning approach not only aims at maintaining computational efficiency but also considers the resilience of the system against possible temporary or permanent failures of devices, an aspect often overlooked in conventional partitioning strategies.

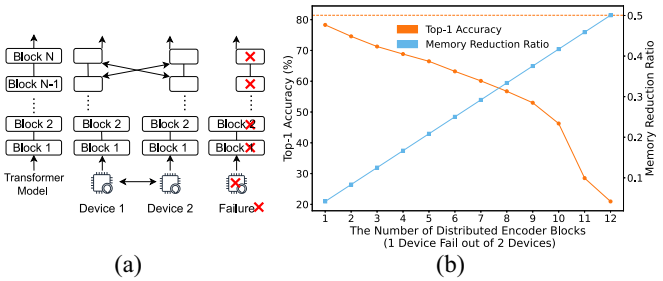
B. Resilience Against Edge Failures

Resilience against device failures at the Edge concerns the property of a model being resilient in terms of inference accuracy to the failure of physical computing devices due to power outages, unstable interdevice connections, other hardware/software failures, etc. In distributed inference settings, the missing neurons mapped on those failed devices may result in a significant accuracy drop of CNN or transformer models [Fig. 1(b)]. Existing approaches and methods to mitigate this risk introduce various strategies. The code distributed computing (CDC) method proposed in [26] exemplifies an early attempt to enhance the resilience by utilizing an additional device to backup the computations of distributed devices. This method effectively mitigates the impact of single device failures but does not scale well to scenarios involving multiple simultaneous device failures without introducing excessive redundancy and associated computational overheads. ElasticDL, introduced by Zhou et al. [27], represents a significant advancement by integrating fault tolerance and elastic scheduling within a Kubernetes-native DL framework. While ElasticDL enhances system resilience and adaptability, its practical deployment on edge devices is hampered by Kubernetes' complexity and the limited computational resources of edge environments.

In contrast to the aforementioned approaches, our methodology *EASTER* introduces a comprehensive solution designed to enhance the resilience of transformer models in the face of the unpredictable and dynamic nature of edge computing environments. Unlike previous methods that often rely on additional hardware resources, complex orchestration, or prior knowledge of potential failure types, *EASTER* employs a novel partitioning strategy that inherently accommodates multiple device failures without necessitating extra devices or computational redundancy. Our approach leverages advanced machine learning techniques to adaptively distribute model computations across edge devices, optimizing for both resilience and resource efficiency. By intelligently partitioning the model in a manner that anticipates and mitigates the impact of device failures, *EASTER* ensures robust inference accuracy under a wide range of failure conditions without the limitations imposed by specific assumptions or the need for supplementary computational overhead.

C. Efficiency in Design Space Exploration

In the context of DSE, the original UCT algorithm [13], known for its efficacy in balancing the exploration–exploitation tradeoff in single-objective optimization problems, is ingeniously adapted to the multiobjective optimization landscape in our work. This adaptation involves selecting promising parts of the search space by not only leveraging the UCT's inherent strengths but also enhancing it with traditional machine learning techniques for more efficient splitting and exploration of the design space. Such an integration significantly augments the UCT framework, enabling it to navigate complex, multidimensional optimization problems with greater precision and efficiency.



(a) (b)

Fig. 1. Comparative analysis of layer partitioning and its impact on memory reduction and for accuracy. (a) Layer partitioning method [7] with failures. (b) Top-1 accuracy versus memory reduction ratio.

Existing DSE methods, such as the multiobjective tree-structured Parzen estimator (MOTPE) [28] and the nondominated sorting genetic algorithm II (NSGA-II) [29], are well known for their efficiency in multiobjective optimization. MOTPE is renowned for its sample efficiency and capability to handle high-dimensional spaces through its Bayesian optimization framework, which is particularly beneficial in scenarios with limited evaluation budgets. NSGA-II, on the other hand, excels in finding a diverse set of solutions across the Pareto front through its evolutionary algorithm, effectively managing the tradeoffs between conflicting objectives. However, existing methods fall short in adapting to our specific scenario, which requires robust splitting of the transformer model block by block while simultaneously optimizing memory usage and inference latency. These methods lack customization for navigating the vast design space of our scenario.

To address this gap, we enhance the UCT algorithm with machine learning techniques to combine the UCT's dynamic exploration–exploitation mechanism with the predictive and generalization capabilities of machine learning. This not only provides an efficient method to identify and explore promising spaces but also enhances the algorithm's ability to adaptively refine its search strategy based on learned insights. Our enhanced UCT approach, when compared to methods like MOTPE and NSGA-II, offers a complementary strategy ideally suited for scenarios where understanding and leveraging the structure of the search space is crucial. This tailored approach significantly boosts our search efficiency and the quality of outcomes, making it a particularly effective solution for our specific robustness needs for splitting transformer models.

III. Robust Model Splitting

In this section, we provide an example to illustrate why splitting a transformer model robustly is needed and why DSE matters in this context. Moreover, we describe how transformers can be splitted in a robust fashion.

A. Motivational Example

The process of splitting a transformer model for distributed inference across edge devices is crucial for running large models in environments with limited resources. Although some

frameworks like PipeEdge [24] could distribute transformer models across multiple IoT devices with orchestration, the crux of the problem lies in the robustness of the pipeline paradigm they utilize: a single failure within the pipeline can compromise the entire computation process. Thus, our discussion focuses on an alternative paradigm, namely, partitioning the layers themselves within a DL model across multiple devices [7]. A transformer model, composed of N encoder or decoder blocks, is designed for various tasks, such as classification or text generation. As illustrated in Fig. 1(a), by dividing blocks in the transformer model into two parts evenly, specifically on a block-by-block basis, we can distribute its workload across two devices. Each device then processes its allocated half blocks, necessitating periodic synchronization of their intermediate results to maintain consistency throughout the computation process. However, such a distribution strategy still introduces a vulnerability: should one of the two devices fail, it results in the loss of half the blocks' processing capability, thereby significantly impacting the model's overall performance and reliability. This scenario underlines the need for a robust distribution strategy that can minimize the risk and impact of device failures.

Taking the ViT-16 transformer model [12] as an example, it contains 12 encoder blocks stacked one by one. The significant impact of a device failure on the model performance is highlighted in Fig. 1(b). When splitting and distributing the model's blocks across two devices, a device failure leads to a substantial drop in Top-1 accuracy, as critical block information is lost. This scenario is graphically represented with Top-1 accuracy (red line) and memory reduction ratio (blue line) against the number of distributed blocks (x-axis), demonstrating that as more blocks are distributed instead of fully replicated, the memory efficiency on the operational device improves, but at the cost of reduced accuracy due to the potential loss of computational resources during a device failure. For instance, when distributing all 12 encoder blocks of the ViT model across two devices, should one device fail due to a power outage or disconnection, half of the weights and intermediate results would be lost. In such a scenario, the top-1 accuracy could drop to 20.95%, significantly impairing the model performance of distributed inference.

This tradeoff between memory reduction and model accuracy underlines the challenge: finding a method to split encoder/decoder blocks that maximizes model accuracy retention while achieving optimal memory efficiency. The goal is to develop a strategy that ensures even if one or more devices fail, the distributed model can maintain as much of its original performance as possible. As mentioned in Section I, given the vast design space for distributing neurons in each encoder/decoder block, it is crucial to employ DSE to identify the most efficient distribution pattern, aiming to minimize accuracy loss while maximizing resource utilization for optimal model deployment in distributed environments.

B. Robust Model Splitting

In the context of a transformer model containing N encoder or decoder blocks, we introduce an innovative uneven splitting method, called *Partial Split*, for distributing these blocks across multiple devices with robustness in mind. This method particularly aims at enhancing the model's resilience to device failures while reducing the memory usage on each device.

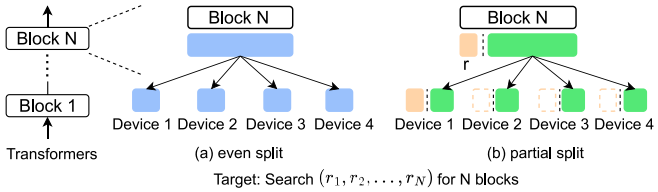


Fig. 2. Partial split. (a) Even split. (b) Partial split.

As illustrated in Fig. 2(a) for example, evenly distributing a transformer block among four edge devices poses a significant risk, namely, the model functionality is severely compromised, for example, when three out of these four devices fail or lose connection, as only a minimal fraction of attention connections remains operational for inference. To address this vulnerability, our method diverges from this conventional even splitting approach.

Instead, our method illustrated in Fig. 2(b) employs a strategic replication of a certain fraction r of critical connections (the yellow box) across multiple devices, based on their weight importance. The remaining, less critical connections (the large green box), constituting a $(1 - r)$ fraction, are then evenly distributed. This selective replication ensures that even in the event of multiple device failures, the most vital connections within each transformer block are retained, thereby preserving the model functionality and inference capabilities to a large extent. During runtime, the device initiating an inference request for image classification or text generation tasks loads both the replicated part (the yellow box) and its split part (the small green box) of the model. The other devices in the network load only their respective split parts. Notably, the replicated part remains unloaded on these devices (the dotted yellow boxes). This runtime loading strategy ensures that extra replicas are not redundantly loaded on other devices, thereby optimizing resource utilization and enhancing overall system efficiency.

IV. Problem Formulation

The aforementioned uneven splitting method facilitates robust distribution of the computational workload of a transformer model across edge devices. However, the limited memory capacities of edge devices introduce challenges in determining the optimal fraction r for each transformer block that could preserve the model functionality and inference capabilities to a large extent. A large fraction r would require high memory usage per device, potentially exceeding the memory capacity of resource-limited edge devices. Conversely, a very small fraction r might compromise the proper model functionality in case multiple devices fail. Thus, an important tradeoff emerges between the memory usage per device and the model functionality that is dependent on the fraction r of critical connections that are replicated for each block.

For a transformer model with N blocks, we define a parameter set $R = \{r_1, r_2, \ldots, r_N\}$, where $r_i \in [0..1]$ represents the fraction of replicated connections for block i. Each set of parameter values R corresponds to different memory usage m_j per device $D_j \in D$ and different model functionality in

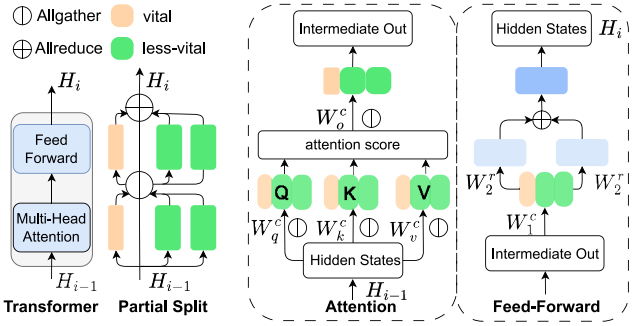


Fig. 3. Transformer partitioning.

case some devices fail at runtime when a transformer model is distributed over a set of edge devices D. Therefore, our objective is to find an optimal set of parameter values R_{opt} which maximizes the model accuracy or performance score in case of failing devices with possible minimum memory usage $(m_1, m_2, \ldots, m_{|D|})$. Given the typically large value of N for prevalent transformer models and the continuous range of $r \in [0..1]$, a vast and complex design space needs to be explored in order to find an optimal solution.

V. *EASTER* Methodology

In this section, we present our novel methodology designed to learn robust distribution strategies for transformer models against device failures that consider the tradeoff between robustness (i.e., maintaining model functionality against failures) and resource utilization (including memory usage and computations). First, we provide more details about our robust partial split method introduced in Section III. Next, we present our DSE approach to solve the optimization problem, formulated in Section IV, that is required to achieve an efficient and robust partial split and distribution of transformer models on multiple edge devices. Finally, we introduce the end-to-end tool we have developed to automate our robust partial split method and distributed deployment of transformer models.

A. Partial Split Method for Transformers

In this section, we explain how the transformer model is split according to a parameter set R. Consider the example shown in Fig. 3 where *Block N* in a transformer model is distributed across two devices and the obtained fraction $r_N \in R$ for this example is 0.25.

The vital part of connections in the attention and feedforward blocks is represented by the two yellow boxes that are both replicated across the two devices. The remaining, less-vital part of connections for each block is split in two (the green boxes) and distributed evenly across the two devices.

To determine the vital part of connections, we calculate and use an importance score for each connection. For example, taking a general linear transformation in the feedforward block, we first calculate the importance of connections corresponding to this linear transformation using the Taylor score [30] as follows:

$$I_{W^k} = |\Delta\mathcal{L}| = \left|\mathcal{L}_{W^k} - \mathcal{L}_{W^k=0}\right| \approx \left|\frac{\partial \mathcal{L}}{\partial W^k} W^k\right| \qquad (1)$$

where I_{W^k} represents the importance score of the kth connection/weights associated with the linear transformation, and $|\Delta\mathcal{L}|$ represents the loss changes when we remove this connection from the layer. After we calculate the importance score of every connection in a layer, we sort the connections based on the importance score in descending order, thereby creating a separate sorted list for every layer. If the target fraction of replicated connections for a layer is r then we start from the beginning of the sorted list and take the first $r\%$ of the connections, thereby classifying them as vital. The rest are classified as less vital. Furthermore, it is crucial to understand that when we find that nearly all connections in a layer have similarly high importance scores (i.e., nearly all are vital), the DSE process (see Section V-B) is designed to adjust the fraction value r of this layer close to 1.0, instead of maintaining the initial value. This adjustment is crucial to preserve and replicate the layer as much as possible to avoid significant performance degradation. During the (design-time) DSE process, the sets of small r values for important layers or blocks, leading to a considerable drop in model performance, are automatically categorized into less promising subspaces. This mechanism ensures that our DSE process systematically avoids configurations that would negatively impact the model's effectiveness significantly. This adaptive approach ensures that our method retains crucial connectivity to effectively retain model performance. Below, we provide details on how our partial split method is further tailored for the attention and feedforward blocks within the transformer architecture to efficiently reduce computational workload and memory usage when the transformer is distributed across different edge devices.

1) Attention Block: As depicted in Fig. 3, the hidden states H_{i-1} coming from the previous transformer block are transformed into queries (Q), keys (K), and values (V) using the weight matrices W_q, W_k, and W_v. Our method splits these matrices along their column dimensions (denoted by W_q^c, W_k^c, and W_v^c) and distributes them across devices. Consequently, each device generates the corresponding segments of Q, K, and V (denoted by the yellow and green boxes), necessitating an all-gather communication operation to concatenate the corresponding segments into complete Q, K, and V tensors. Taking the linear transformation with W_q weights (Fig. 3) in the attention block as an example, the query matrix Q is generated by W_q. If the embedded dimension of the input tensor H_i is D, we compute the D importance scores for query Q using (1). Once the replication factor r_i is determined, we rank and split the W_q weights along its column dimensions based on the rank indices derived from the D scores. We choose to replicate the top $r\%$ of weight W_q and allocate the remaining $(1-r)\%$ to multiple devices. For the small portion of W_q on each device, we replace the original matrix multiplication (matmul) operation with a small matmul operation containing its corresponding different part of weight W_q. To maintain output accuracy, a communication operation for gathering the partial Q output is added after the small matmul. After the attention block multiplies the attention scores with values (V), the linear transformation with weight matrix W_o maps the multiplication result to match the dimension size of the intermediate output. In our method, we also split W_o into segments along the column dimension. Each segment of W_o^c produces a partial part of the intermediate output. Similarly, an

extra all-gather communication operation is added to collect the segments and ensure the correctness.

Apart from these layers, the embedding layer follows a similar strategy for its matmul operation. However, we abstain from applying our partial split method to layernorm layers due to their relatively minimal weight and computational demand. Importantly, the full replication of layernorm weights on each device is prioritized to ensure model stability, given their significant role [31].

2) Feedforward Block: This block within a transformer block involves two linear transformations with weight matrices W_1 and W_2 to process the intermediate output and generate the hidden states H_i going to the subsequent transformer block. The first weight matrix is split along the column dimension (denoted as W_1^c in Fig. 3). The second weight matrix is split along the row dimension (denoted as W_2^r). The partial output tensors (the yellow and green boxes) produced by W_1^c can directly go through the nonlinear activation and serve as the input for the second linear transformation which is also split and denoted as W_2^r. This design eliminates the need for an all-gather operation to concatenate the partial outputs produced by the first linear transformation, thereby reducing both the computational workload per device and the interdevice communication overhead. Finally, a collective all-reduce operation is applied to sum the partial output from all devices to form the correct hidden states output H_i.

B. Design Space Exploration

To solve the optimization problem formulated in Section IV, we have devised a DSE approach that effectively navigates in the vast and complex design space mentioned in Section IV. Our DSE approach leverages supervised learning techniques to progressively concentrate the search for an optimal solution within increasingly smaller and more promising spaces, thereby enhancing search efficiency. As depicted in Fig. 4, the approach starts by randomly generating several design points $\mathbb{R} = \{R_1, R_2, \ldots, R_p\}$ (yellow points), and evaluate the objectives $\mathbb{F}(\mathbb{R})$ using the fitness function $\mathbb{F}$ for each design point $R_i \in \mathbb{R}$ to form an initial learnable space $D = (\mathbb{R}, \mathbb{F}(\mathbb{R}))$. Here, $R_i = \{r_1^i, r_2^i, \ldots, r_N^i\}$ is a set of fractions corresponding to a specific partial split strategy for all N blocks in a transformer model. The fitness function $\mathbb{F}$ concerns the evaluation of various conflicting objectives, such as memory usage, energy consumption, performance, etc. It can be implemented using analytical models, real measurements, etc. In this article, our fitness function is based on real measurements to ensure an accurate and practical evaluation of the objective values. Taking the ViT-16 model as an example, we directly measure the peak memory usage on real devices during run-time, and we take the Top-1 accuracy of the ImageNet-1K validation dataset as the performance metric. Then, our DSE approach recursively splits the design space D and obtains a set of split boundaries. Subsequently, we apply these learned boundaries to generate new design points within specific promising design spaces to improve the search efficiency. We apply the calculation equation in line 24 of Algorithm 1 to identify which area within $\mathbb{R}$ is most likely to contain optimal design points and then concentrate our search

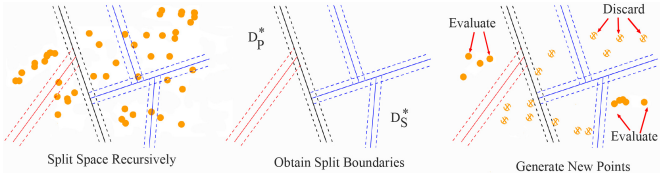


Fig. 4. Our DSE approach.

on this smaller, promising area, denoted as D_P^* and shown in the middle of Fig. 4. However, an early decision about the promising area might inadvertently overlook other areas that could contain optimal points as well. To mitigate this, while the majority of our design points are generated within the currently perceived promising area D_P^*, we also allocate a smaller portion of design points to generate from other spaces, represented by D_S^*. This approach iteratively learns the entire space $\mathbb{R}$ and allows us to more accurately identify the most promising regions for optimal points.

Algorithm 1 describes, in more detail, the aforementioned DSE approach illustrated in Fig. 4. The algorithm consists of two main steps and takes as an input the maximum search trials T, the number of new random design points n_p for updating the search space D, a lower bound (lb) to determine the maximum number of design points in an unsplittable area, and the exploration factor α which determines the degree of exploration. A higher value for α encourages more exploration in the search space. The output of Algorithm 1 is space $D_P = \{(R_1, F_{R_1}), \ldots, (R_{|P|}, F_{R_{|P|}})\}$ of Pareto-optimal solutions where every solution $R_i = \{r_1^i, r_2^i, \ldots, r_N^i\}$ is a set of fractions corresponding to a Pareto-optimal partial split strategy for all N blocks in a transformer model. In line 1, we first randomly initialize a number of design points and evaluate their objectives using the fitness function, yielding an initial learnable search space D.

In step 1 (lines 3–8), the algorithm narrows down the space via support vector machine (SVM) classifiers and generates a series of SVM boundaries. In lines 3–6, we select the nondominated points from D to create a new primary space marked as D_P, and the rest of the points are put into a new secondary space marked as D_S. In lines 7 and 8, the *NarrowDown* function is applied to recursively split D_P and D_S into smaller spaces D_P^* and D_S^*. Concurrently, all involved splitting SVM boundaries are aggregated into the boundary sets $\mathbb{CL}_P$ and $\mathbb{CL}_S$.

In step 2 (lines 9–12), we generate new design points and evaluate these new design points using the fitness function FITNESS. To balance the exploration–exploitation tradeoff, 80% of these new points ($\mathbb{R}_P$) are derived from D_P^* in line 9, while the remaining 20% (i.e., for $\alpha = 0.2$) of the new design points ($\mathbb{R}_S$) are derived from D_S^* in line 10. This ratio, while adjustable, typically requires experimental trials for better search efficiency. Then, we apply the fitness function to evaluate the objective values for these new points and add them to the search space D in line 12. This iterative process is repeated until the maximum number of trials T is reached (see line 2). Ultimately, the Pareto-optimal points comprising space D_P found by this DSE process represent the optimal solutions that balance the memory usage and the model functionality.

Algorithm 1: DSE

Input : Maximum trials T; Population size n_p; lb, exploration factor α;
Output: Space D_P with Pareto points;

1 Initialize randomly D with points (R, F_R):
  $D \leftarrow \{(R_1, \text{FITNESS}(R_1)), \cdots, (R_{n_p}, \text{FITNESS}(R_{n_p}))\}$
2 **while** $|D| \leq T$ **do**
  // Step 1: Narrow Down Search Space
3 **foreach** $(R_i, F_{R_i}) \in D$ **do**
4 **if** F_{R_i} *is nondominated* **then**
5 $D_P \leftarrow D_P \cup (R_i, F_{R_i})$
6 $D_S \leftarrow D \setminus D_P$; $\mathbb{CL}_P \leftarrow \emptyset$; $\mathbb{CL}_S \leftarrow \emptyset$
7 $D_P^*, \mathbb{CL}_P =$ NarrowDown$(D_P, \mathbb{CL}_P)$
8 $D_S^*, \mathbb{CL}_S =$ NarrowDown$(D_S, \mathbb{CL}_S)$
  // Step 2: Add New Random Points,
  // Evaluate and Update D
9 $\mathbb{R}_P =$ NewPoints$((1-\alpha) * n_p, \mathbb{CL}_P)$
10 $\mathbb{R}_S =$ NewPoints$(\alpha * n_p, \mathbb{CL}_S)$
11 **foreach** $R_i \in (\mathbb{R}_P \cup \mathbb{R}_S)$ **do**
12 $D \leftarrow D \cup (R_i, \text{FITNESS}(R_i))$
13 **return** D_P
14
15 **Function** NarrowDown$(D, \mathbb{CL})$**:**
16 **foreach** $(R_i, F_{R_i}) \in D$ **do**
17 $\mathbb{R}_D \leftarrow \mathbb{R}_D \cup R_i$
18 $(\mathbb{R}_{D_1}, \mathbb{R}_{D_2}) = \text{KMeansTwoClustersOn}(\mathbb{R}_D)$
19 $D_L = (\mathbb{R}_{D_1}, 1) \cup (\mathbb{R}_{D_2}, -1)$
20 $\text{CL}, D_1, D_2 = \text{SVMTrainedOn}(D_L)$
21 **if** $(|D| < lb) \vee (\text{CL}(D_1) = \text{CL}(D_2))$ **then**
22 **return** D, $\mathbb{CL}$
23 **else**
24 $\text{UCB}(\mathbb{R}_{D_i}) = \overline{\mathbb{F}(\mathbb{R}_{D_i})} + \alpha\sqrt{\frac{\log|\mathbb{R}_D|}{|\mathbb{R}_{D_i}|}}: i = 1, 2$
25 $\mathbb{R}_{D^*} = \underset{\mathbb{R}_{D_i}}{\text{argmax}}\ \text{UCB}(\mathbb{R}_{D_i})$; $D^* = (\mathbb{R}_{D^*}, \mathbb{F}(\mathbb{R}_{D^*}))$
26 $\mathbb{CL} \leftarrow \mathbb{CL} \cup \text{CL}$
27 **return** NarrowDown$(D^*, \mathbb{CL})$
28 **Function** NewPoints$(N, \mathbb{CL})$**:**
29 $\mathbb{R} \leftarrow \emptyset$
30 **while** $|\mathbb{R}| < N$ **do**
31 $R =$ *RandomPoint*;
32 $\mathbb{R} \leftarrow \mathbb{R} \cup R$
33 **foreach** $\text{CL}_i \in \mathbb{CL}$ **do**
34 **if** $\text{CL}_i(R) = -1$ **then**
35 $\mathbb{R} \leftarrow \mathbb{R} \setminus R$
36 *break*
37 **return** $\mathbb{R}$

In lines 15–27, the *NarrowDown* function recursively splits the search space D and obtains a series of learned split boundaries $\mathbb{CL}$. In line 18, we initially employ the K-means clustering method to categorize/divide the design points within $\mathbb{R}_D$ into two distinct clusters $\mathbb{R}_{D_1}, \mathbb{R}_{D_2}$. Following this clustering, we calculate the average objective values for each cluster. The cluster with the higher average objective values is considered to be situated in a more favorable space. Consequently, in line 19, we assign a label of 1 to the design points in this more promising cluster, while design points in the less favorable cluster are labeled as -1, and we put all labeled points in a new set D_L. In line 20, we train the SVM classifier *CL* with the new set of labeled points D_L and split D_L into two spaces D_1 and D_2. In lines 21 and 22, if the number of design points in D is below the lower bound lb or if the SVM classifier *CL* predicts only a single category, both indicating that space D is nondivisible, the recursive function *NarrowDown* terminates and returns the set of classifiers $\mathbb{CL}$. Otherwise, in lines 24–26, we mark the space with the larger UCB value [13], calculated in line 24, as the more promising design space D^*, and add the SVM classifier *CL* into the recursive splitting set $\mathbb{CL}$.

In lines 28–37, the *NewPoints* function randomly generates N new design points using the input set of SVM classifiers $\mathbb{CL}$. In lines 31 and 32, a random design point R is generated and added to the set of new points $\mathbb{R}$. Then, point R is classified using the set of trained SVM classifiers $\mathbb{CL}$ in lines 33–36. That is, if all SVMs in $\mathbb{CL}$ classify point R to belong to the class with label 1 then point R remains in the set; otherwise, it is removed (line 35). Finally, in line 37, the new set of random points $\mathbb{R}$ is returned.

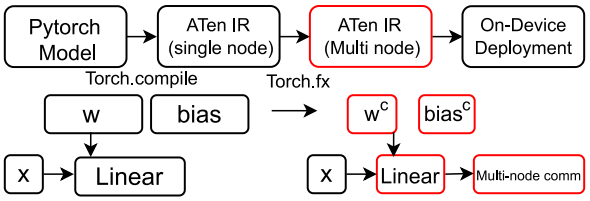


Fig. 5. Multinode IR conversion tool.

C. Multinode Intermediate Representation

We have developed an end-to-end tool that facilitates automated model partitioning and its distributed deployment, in line with one of the Pareto-optimal partial split strategies $R_i \in D_P$ found by our DSE Algorithm 1 presented in Section V-B. In general, traditional frameworks for DL model deployment on edge devices, such as TVM [32], IREE [33], and others, do not sufficiently support distributed inference. Therefore, our end-to-end tool is implemented to transform CNNs or transformer models from Huggingface [34] into optimized multinode computation graphs, thereby making them suitable for efficient deployment across multiple devices. Our tool is versatile enough to support both CNNs and transformer models but in this article we focus on its application to transformer models.

As illustrated in Fig. 5, our tool begins by utilizing the existing "torch.compile" [35] method to convert an initial PyTorch transformer model into the low-level ATen IR for a single node. Subsequently, an automated conversion process is employed to replace the single-node ATen IR into a multinode variant. For instance, in handling linear transformations, the tool splits the associated coefficients and redefines new

Linear transformations that are adapted to the altered shapes of coefficients or inputs as illustrated by the red boxes in Fig. 5. Modifications to these operations are facilitated using "torch.fx" [36], accommodating the new coefficient dimensions. Our own customized multinode communication operations, such as GatherByIndex, AllReduceByIndex, AllConcatByIndex, etc., are integrated after the modified operation (see red box "Linear" in Fig. 5) to ensure the calculation correctness. To enhance the tool's versatility, we implement these communication operations in C++ such that they can be integrated into other inference engines. We have also developed a compatible interface that enables the conversion of this multinode IR into formats supported by various other inference engines (e.g., NCNN [37], IREE, etc.). Its compatibility and ease of integration with these existing edge frameworks enhances both usability and scalability. Additionally, a robust fault handler is incorporated to ensure reliable execution during distributed inference, providing resilience against potential device failures or network disruptions. An inner timeout mechanism governed by periodic heartbeats [38] can prevent the distributed system from deadlocks that might arise due to device failures or other operational anomalies.

VI. Evaluation of Our *EASTER* Methodology

In this section, we evaluate our *EASTER* methodology to demonstrate its efficacy on typical transformer models and showcase resilient models' performance. We describe our experimental setup followed by presenting and discussing some experimental results obtained during automated DSE experiments, we have performed using Algorithm 1 and the end-to-end tool introduced in Section V-C.

A. Experimental Setup

To evaluate *EASTER*, we perform experiments with three typical transformer models, namely, ViT-16 [12], GPT2-Large [14], and Vicuna-7B [4] representing three different kinds of transformer architectures, taken from the Huggingface open-source community [34]. Given their widespread use in image and text tasks, and their diversity in transformer blocks, operation counts, and memory requirements, we consider these transformers to be representative targets to demonstrate the merits of our methodology. We compare the searching efficiency of our Algorithm 1 on these models with two state-of-the-art multiobjective optimization algorithms, namely, the NSGA-II Genetic Algorithm [29] and MOTPE [28]. The task of our DSE experiments is to simultaneously minimize the maximum memory usage per device and the model performance score (loss) under severe device failures. To ensure a fair comparison with NSGA-II and MOTPE, we set the maximum number of search iterations to 2500 for each DSE experiment. The searching time for the three methods are quite similar, with the majority of time being consumed by the objective evaluations. For the first objective (maximum memory usage per device), we normalize its value range to [0, 1] by dividing the memory usage $m_j(R_i)$ by the total memory usage on a single device D_j. Lower values indicate reduced replication and more balanced model distribution. To evaluate the second objective (performance score S) of the models, we employ distinct techniques tailored to each model's specific domain. For the ViT-16 model, we measure the Top-1 error score on the ImageNet-1k dataset for image tasks. A lower error represents higher image classification capabilities, and the lower the error the better. For the two LLMs (GPT2-Large and Vicuna-7B), we utilize zero-shot perplexity (PPL) analysis on the WikiText2 and PTB datasets to assess the models' language understanding and generalization capabilities. A lower PPL score, especially in a zero-shot context, means a better ability to handle unseen data.

To validate the performance of Pareto-optimal points from the DSE process using Algorithm 1, we apply the split fractions R_i, found by the algorithm, to the two LLMs by distributing each LLM across four devices, i.e., four GPU units in our experiments. We disable three GPU units to simulate severe device failure scenarios in order to assess the models' robustness. We apply a separate and more diverse collection of reasoning and generative datasets [39] to test the models' performance (robustness) against severe failures in practical reasoning tasks, namely, ARC-easy, ARC-challenge, WinoGrande, HellaSwag, BoolQ, PIQA, and OpenbookQA. These diverse datasets provide a comprehensive platform for testing the models' reasoning and generative capabilities.

To evaluate the resilience of our methods under varying failure conditions, we deployed three models across four edge devices and examined model performance in scenarios where 1 (1D-Fail), 2 (2D-Fail), or 3 devices (3D-Fail) experience failures. We take the state-of-art layer partitioning method (LP) [7] from the domain of distributed CNN inference as inspiration to implement a similar method for linear operations within encoder/decoder blocks of transformer models. Subsequently, we benchmark this LP-inspired partitioning method, which does not utilize the notion of neuron importance, against our approach in terms of robustness. For the three transformer models, we assess the robustness of our method using different sets of R values for the partial split strategy, allowing for a comprehensive comparison of how well each method retains model performance against device failures.

To actually test distributed inference for transformers across multiple edge devices, our experimental edge test-bed consists of eight NVIDIA Jetson Xavier NX devices connected over a 1000-Mb/s network router. Each device has an embedded MPSoC featuring a 6-core Carmel ARMv8.2 CPU, an NVIDIA Volta GPU with 384 CUDA cores, 48 Tensor cores, and 8 GB of LPDDR4x memory. We demonstrate the functionality of our multinode implementation, generated by our end-to-end tool introduced in Section V-C, and the advantages of distributing large transformer models over multiple edge devices/boards by conducting a series of benchmarks on the aforementioned edge test-bed using the three representative transformer models ViT-16, GPT2-Large, and Vicuna-7B under four different distributed system configurations: single device, two devices, four devices, and eight devices. In all experiments, transformer blocks were evenly distributed across the devices. We mainly evaluate two metrics: 1) overall end-to-end inference latency and 2) memory reduction with different distribution configurations.

The end-to-end latency (T) of a model is measured from the time a user input is received until the time the complete output

TABLE I
EXECUTION TIME OF MAIN STEPS IN *EASTER*

	Importance Calculation		Evaluation Time Per DSE Trial	
	CPU (s)	GPU (s)	CPU (s)	GPU (s)
ViT-16	200.90	4.15	2480.30	156.73
GPT2-Large	43.35	4.70	35.48	4.65
Vicuna-7B	430.42	8.44	48.742	7.82

is generated. For the ViT-16 model, user inputs are images with dimensions (3×224×224), whereas for the two LLMs (GPT2-Large and Vicuna-7B), user inputs are sequences of 128 tokens. The reported latency is computed by averaging time T for 100 user inputs. To measure T and break it down to computation time (T_{cal}) and communication/synchronization overhead (T_{comm}) in our distributed inference execution, we employ a specific adjustment of the timeout parameter values in our multinode communication operations introduced in Section V-C. More specifically, setting the timeout values to zero permits each device to function independently, i.e., without interdevice data communication and synchronization delays, thereby enabling the measurement of the pure computation time T_{cal}. Altering the timeout values to one second activates interdevice communication and synchronization actions besides the pure computations, thereby facilitating the measurement of the total end-to-end inference latency T. We then determine the communication/synchronizaton overhead T_{comm} by calculating the difference $T-T_{\text{cal}}$, thereby effectively quantifying the additional time needed for interdevice data communication and synchronization.

To determine the aforementioned memory reduction, we continuously monitor the peak memory usage of each device in our edge test-bed during runtime for every distributed system configuration.

B. Execution Time Evaluation of the EASTER Method

We evaluate the execution time of the main steps of our *EASTER* method on two different hardware platforms, namely, a platform based on an Intel Core i9-13900K CPU and a platform based on an NVIDIA H100 SXM5 GPU. For each transformer model, we measure the time required to calculate the importance scores of connections within the model as well as the time to evaluate a single design point during the DSE process.

The importance score calculation is performed only once. Illustrating this calculation for the ViT-16 model, we randomly take 50 samples from the ImageNet-1K training dataset where each sample is a batch of 128 random images. Using each sample and the ViT-16 model, we apply (1) to calculate an importance value for every connection within the model, i.e., we calculate 50 values per connection in total. Then, we compute the average of these 50 values for each connection and use this average value as the importance score of the connection in our DSE process. For the GPT2-Large and Vicuna-7B transformer models, the importance scores are calculated similarly through 50 random samples from the language datasets. The time required to execute the importance score calculation for the three transformer models is shown in Columns 2 and 3 of Table I. For example, on the GPU-based platform, the complete set of importance scores of all connections in the ViT-16 model is computed in just 4.15 s. Computing the same set of scores on the CPU-based platform takes 200.9 s. In Columns 4 and 5 of Table I, we provide the evaluation time for a single design point in our DSE process. For example, on the CPU-based platform, evaluating the Top-1 accuracy of the ViT-16 model takes approximately one hour to complete. Conversely, the powerful GPU platform validates the Top-1 accuracy for a specific design point in under 3 min.

C. DSE Results and Comparison

We have performed three distinct DSE experiments for the ViT-16, GPT2-Large, and Vicuna-7B models by employing our *EASTER* methodology and Algorithm 1 along with the NSGA-II and MOTPE algorithms for comparison purposes. The Pareto-optimal points found by each of these three algorithms are separately plotted in Fig. 6. The yellow triangles represent the points found by MOTPE, the blue crosses represent NSGA-II points, and the red dots correspond to points found by our Algorithm 1 within *EASTER*. The x-axis in Fig. 6(a)–(c) represents the normalized maximum memory usage per device explained in Section VI-A. The y-axis represents the Top-1 error for ViT-16 and the PPL for GPT2-Large and Vicuna-7B. The rationale behind using the Top-1 error and PPL is explained in Section VI-A.

To quantitatively assess the effectiveness of *EASTER*, NSGA-II, and MOTPE, as well as to compare them, we calculate the well known and widely used hypervolume metric (hv), based on the Pareto-optimal points plotted in Fig. 6, that serves as an indicator of the search space coverage in DSE. As shown in Fig. 6, our *EASTER* methodology and algorithm demonstrate superior performance because of the higher hypervolume value *hv*, indicating more effective search space coverage of *EASTER* compared to NSGA-II and MOTPE. For example, the Pareto-optimal points found by *EASTER* for Vicuna-7B and shown in Fig. 6(c) dominate those found by NSGA-II and MOTPE, resulting in higher hypervolume value of 3.24 and highlighting the *EASTER* effectiveness in identifying optimal solutions.

As explained in Section VI-A, we apply the split fractions R_i, found by Algorithm 1, to the models by distributing each model across four devices. Moreover, we disable *three of the four* devices in order to simulate severe device failure scenarios to assess the models' robustness. The results for the LLMs (GPT2-Large and Vicuna-7B) are shown in Table II.

The first column specifies three different R_i settings for each of the two LLMs together with the baseline setting, named $R = 1$. The baseline setting $R = 1$ for each LLM is the original model fully replicated over the four devices with no loss of model weights/connections due to failures. Note that the evaluation metrics associated with settings A–C are also shown in Fig. 6(b) and (c)—see the red dots marked with A–C. The second column in Table II shows the maximum memory usage per device under the aforementioned settings. The remaining columns show the evaluation accuracy (in %) of the operational part of the model, i.e., the part still running on the nonfailing device, across several zero-shot open-ended tasks on widely recognized

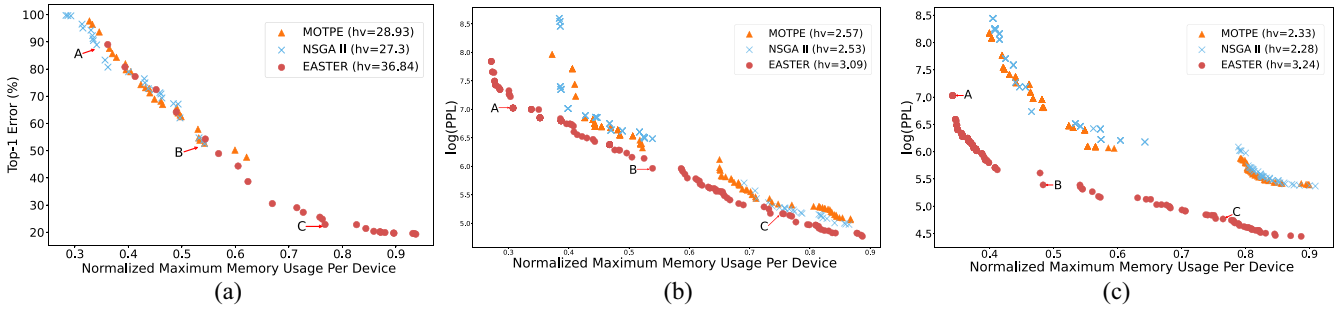


Fig. 6. Comparison of DSE results delivered by *EASTER*, NSGA-II, and MOTPE for (a) ViT-16, (b) GPT2-Large, and (c) Vicuna-7B.

TABLE II
ZERO-SHOT PERFORMANCE (MAX. PER-DEVICE MEMORY USAGE AND ACCURACY-%) WITH THREE OUT OF FOUR EDGE DEVICES FAILING

Models	Memory (reduction ratio)	ARC-c	ARC-e	WinoGrande	HellaSwag	OBQA	PIQA	BoolQ
Vicuna-7B (A)	9.24 GB(-65.80%)	21.93	33.71	52.25	29.36	17.00	57.73	62.14
Vicuna-7B (B)	13.06 GB(-51.60%)	27.13	44.49	57.14	34.39	20.60	64.58	62.17
Vicuna-7B (C)	20.65 GB(-23.50%)	38.31	67.97	67.56	50.40	28.20	72.96	79.05
Vicuna-7B (R=1)	27.00 GB(baseline)	43.17	75.63	69.46	56.48	33.00	77.31	80.98
GPT2-Large (A)	1.08 GB/(-66.20%)	19.54	29.88	50.12	26.41	12.60	55.28	54.22
GPT2-Large (B)	1.72 GB/(-46.10%)	19.54	33.96	49.49	28.39	11.60	59.85	60.92
GPT2-Large (C)	2.42 GB/(-24.50%)	18.86	44.61	53.35	31.89	18.00	65.45	62.05
GPT2-Large (R=1)	3.20 GB/(baseline)	21.67	53.16	55.33	36.40	19.40	70.35	60.49

common sense reasoning datasets [39]: ARC-e(asy), ARC-c(hallenge), WinoGrande, HellaSwag, BoolQ, PIQA, and OpenBookQA.

Analyzing the results in the second column of Table II, we observe that the memory reduction for setting C with $R \approx 0.75$ compared to the baseline clearly shows that the accuracy loss is relatively small. The memory reduction for settings A and B in this worst-case scenario (3D-Fail) confirms the efficacy of our *EASTER* methodology. For example, the Vicuna-7B model experiences a significant memory reduction of up to 65.80% (from 27.00 to 9.24 GB), but still retains competitive accuracy compared to the original GPT2-Large model across several evaluated tasks like WinoGrande and BoolQ. Although the memory reduction comes with a certain accuracy tradeoff, especially for tasks like ARC-c, ARC-e, etc., this remains within an acceptable range given the significant benefits of reduced memory demands and improved computational efficiency across multiple constrained devices. The GPT2-Large model in setting B with a memory reduction of 66.20% shows a relatively minor performance decline in terms of accuracy for datasets like ARC-c, WinoGrande, and BoolQ. Here, ARC-e task shows the highest accuracy sensitivity to memory reduction, i.e., a decrease of 23.28% in accuracy. However, it is important to note that our DSE methodology and algorithm prioritize the optimization for general PPL scores, rather than tailoring the search to enhance specific task scores. To further improve the accuracy of different datasets, our DSE method can be applied to search for optimal design points targeting the accuracy separately for each dataset. This approach allows for maintaining robust performance while ensuring minimal accuracy drop for individual datasets. However, it is important to recognize that this will result in different optimal design points (different sets of R values) for each dataset.

Overall, both models demonstrate a notable degree of performance resilience under extreme failure scenarios, indicating their potential for effective deployment in environments with memory constraints, such as edge devices.

D. Robustness Verification Against Varying Failures

To deepen our understanding of *EASTER*'s robustness, we compare our robustness-aware method against the LP-inspired method which does not utilize the notion of the importance of neurons. To maintain a fair comparison, we select the settings marked as A–C in Fig. 6 to split the transformer models across four devices according to the R values associated with the three marked settings by utilizing the two methods.

As depicted in Fig. 7, the x-axis categorizes the failure scenarios (1D-Fail, 2-D-Fail, or 3D-Fail), whereas the y-axis quantifies model performance, measured by the Top-1 accuracy on the ImageNet-1k validation dataset or perplexity (PPL) value. Please note the logarithmic scale for the PPL scores. The graphical representation uses blue bars to indicate the performance of the traditional layer-wise partitioning (LP-inspired) method in the face of device failures, while orange bars illustrate the performance of our *EASTER* method.

Consider Fig. 7(a) and the 2D-Fail scenario. When the ViT-16 model is split with $R = 0.33$, the Top-1 error of the LP-inspired method is as high as 94.742%, in contrast to our method, which significantly lowers the Top-1 error to 54.626%. By increasing the R value from 0.33 to 0.53, we observe a further reduction of the Top-1 error to 31.238%. Increasing the R value further to 0.77 results in the Top-1 error dropping to 20.614%, which is very close to the baseline Top-1 error of 18.572%. Note that our method can achieve this baseline error if we set R to 1.0 (as shown in Fig. 7) because this setting "forces" our method to perform full replication of neurons, i.e., no accuracy loss is encountered due to device

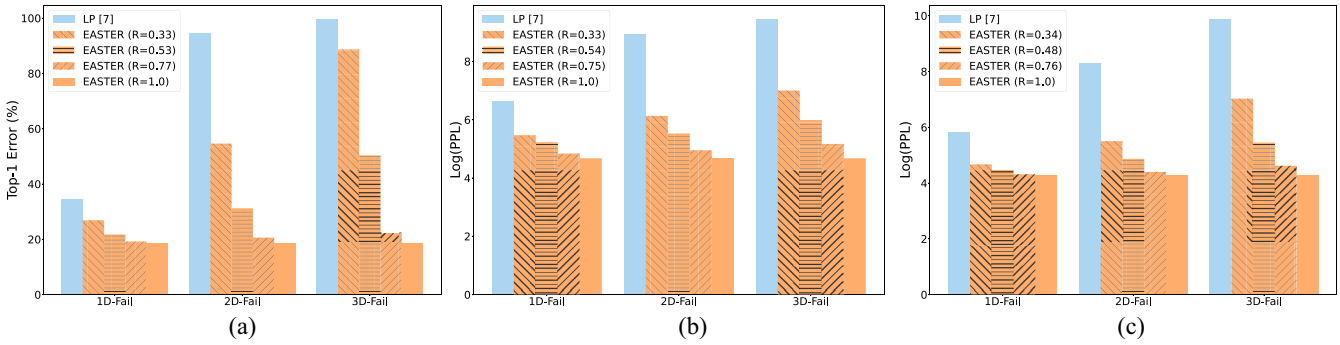


Fig. 7. Robustness comparison of *EASTER* with layer-wise partitioning [7] across four devices. (a) ViT-16. (b) GPT2-Large. (c) Vicuna-7B.

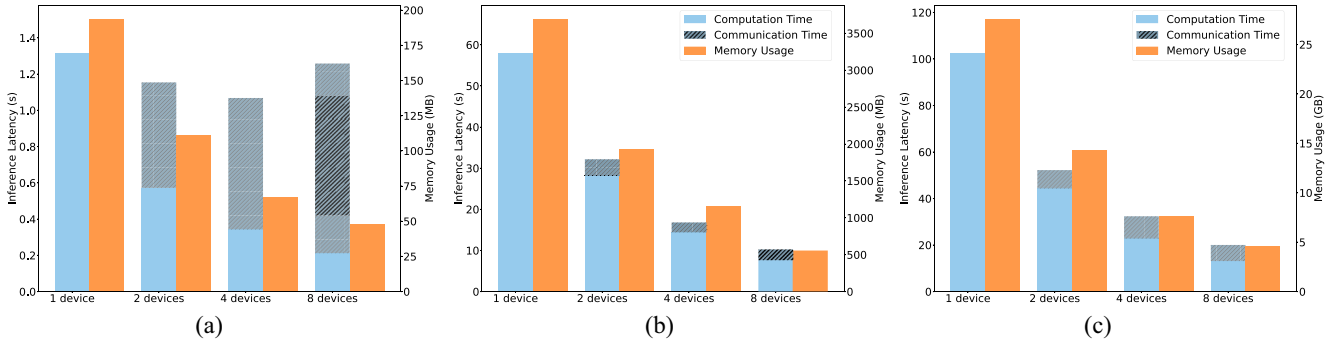


Fig. 8. Inference latency, communication time, and memory usage for different models across device configurations. (a) ViT-16. (b) GPT2-Large. (c) Vicuna-7B.

failures. Similarly, with the Vicuna7B model, the logarithmic value of perplexity (PPL) observed using the LP-inspired method under a 2D-Fail condition is 8.29. In contrast, our method achieves a log(PPL) of 5.50 with an R value of 0.34. Further increasing the R value to 0.76 results in an even lower log(PPL) which is very close to the baseline ($R = 1.0$).

These results clearly demonstrate that our *EASTER* method significantly outperforms the LP-inspired method in maintaining model performance against device failures. Moreover, increasing the R value, which dictates the degree of neuron replication, can further improve model robustness.

E. Distributed Inference

In this section, we evaluate our end-to-end tool that facilitates automated model partitioning and its deployment on distributed edge devices. Our tool is specifically implemented to convert standard PyTorch transformer models into optimized multinode implementations following our *EASTER* methodology, making the models suitable for efficient distributed deployment on edge devices. We present empirical results, obtained by using our edge test-bed described in Section VI-A, in order to demonstrate the advantages of *EASTER* in terms of overall end-to-end inference latency and maximum memory usage per device in a distributed system running transformer models. Here, in all experiments, transformer blocks are evenly distributed across the devices. In Fig. 8, the light blue bars represent the computation time T_{cal} of the distributed inference process, the gray blue bars indicate the communication/synchronization overhead T_{comm}, whereas the orange bars in Fig. 8 denote the maximum memory usage per device. The data is presented for different numbers of collaborating edge devices across the three models.

As shown in Fig. 8, in most cases, the overall end-to-end inference latency improves when increasing the number of edge devices. As the number of devices increases, in all cases, computation time T_{cal} (light blue bars) reduces correspondingly. Only in the case of ViT-16 [Fig. 8(a)], this advantage is counterbalanced by a rise in the communication overhead (gray bars), which, in an eight-device setup, surpasses the computational savings, leading to an overall increase in the inference latency. Conversely, for GPT2-Large, the communication overhead, while increasing with more devices, still remains a smaller fraction compared to the computation time. This results in a near-linear acceleration, with an overall inference latency decrease from 58.00 s using one device to 7.62 s using eight devices. The increase in communication overhead therefore seems more pronounced in smaller transformer models like ViT-16, that represents a fundamental tradeoff between computation and communication.

The results shown in Fig. 8 clearly indicate that with an increasing number of devices (from 1 to 8 devices), there also is a noticeable decrease in memory usage per device. For instance, the maximum on-device memory usage for ViT-16 decreases from 193.8 MB in a single-device configuration to 48.1 MB in an eight-device configuration. Similarly, GPT2-Large exhibits a significant memory reduction from 3.6 GB on a single device to 556.3 MB across eight devices. A significant reduction in memory usage per device from 27.6 GB on a single-device configuration to 4.6 GB on an eight-device configuration is observed for Vicuna-7B as shown in Fig. 8(c). Such reduction enables the models to run the complete float32

version at the edge without the need for extra swap space or model quantization, highlighting *EASTER*'s effectiveness in memory savings.

Finally, if the reduction in computation time due to distributed inference is outweighed by the increase in communication time, the overall end-to-end latency increases. We can adjust timeout thresholds in the system to manage the tradeoff between computation and communication times. By implementing such a timeout mechanism, we ensure that if synchronization among distributed devices does not conclude within the set time period, the system proceeds without further delay, thus maintaining timely execution. This approach not only mitigates potential increases in communication time but also safeguards against the detrimental effects of prolonged synchronization wait times.

The above findings validate the efficiency of *EASTER* in optimizing memory usage per device in distributed transformer inference, particularly in edge computing environments where resource constraints are a critical factor.

VII. Conclusion

This article introduces *EASTER*, a novel method designed to robustly partition transformer models across edge devices, effectively addressing the challenge of potential device failures at the Edge. The *EASTER* method navigates the vast design space of splitting strategies by learning the expectation of different design subspaces. It also outperforms traditional state-of-the-art DSE methods in searching efficiency for our distribution problem. Through extensive experimentation, *EASTER* has been proven to identify Pareto solutions within a limited number of experimental trials efficiently.

Utilizing our developed end-to-end tool, we have the capability to evaluate the distributed implementation on actual hardware boards, which allows us to confirm the advantages in memory usage and inference latency that distributed inference brings. Moreover, our findings substantiate that partial splitting significantly enhances model robustness in the face of device failures. This approach not only minimizes memory consumption on each device but also has the potential to reduce overall end-to-end latency, presenting a valuable opportunity for deploying large-scale transformer models within edge computing environments.

References

[1] Y. Cao et al., "A comprehensive survey of AI-generated content (AIGC): A history of generative AI from GAN to ChatGPT," 2023, *arXiv:2303.04226*.

[2] OpenAI, "GPT-4 technical report," 2023, *arXiv:2303.08774*.

[3] M. N. Birje and S. S. Hanji, "Internet of things based distributed healthcare systems: A review," *J. Data, Inf. Manage.*, vol. 2, pp. 149–165, 2020.

[4] L. Zheng et al., "Judging LLM-as-a-judge with MT-bench and Chatbot arena," in *Proc. 37th Conf. Neural Inf. Process. Syst.*, 2023, pp. 1–39.

[5] J. Lin et al., "AWQ: Activation-aware weight Quantization for LLM compression and acceleration," 2023, *arXiv:2306.00978*.

[6] X. Zhu, J. Li, Y. Liu, C. Ma, and W. Wang, "A survey on model compression for large language models," 2023, *arXiv:2308.07633*.

[7] R. Stahl, A. Hoffman, D. Mueller-Gritschneder, A. Gerstlauer, and U. Schlichtmann, "DeeperThings: Fully distributed CNN inference on resource-constrained edge devices," *Int. J. Parallel Program.*, vol. 49, no. 4, pp. 600–624, 2021.

[8] L. Zhou, M. H. Samavatian, A. Bacha, S. Majumdar, and R. Teodorescu, "Adaptive parallel execution of deep neural networks on heterogeneous edge devices," in *Proc. 4th ACM/IEEE Symp. Edge Comput.*, 2019, pp. 195–208.

[9] H. Touvron et al., "Llama 2: Open foundation and fine-tuned chat models," *arXiv preprint arXiv:2307.09288*, 2023.

[10] Z. Li et al., "AlpaServe: Statistical multiplexing with model parallelism for deep learning serving," in *Proc. OSDI*, 2023, pp. 663–679.

[11] J. Rasley, S. Rajbhandari, O. Ruwase, and Y. He, "Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters," in *Proc. ACM SIGKDD'26*, 2020, pp. 3505–3506.

[12] A. Dosovitskiy et al., "An image is worth 16x16 words: Transformers for image recognition at scale," in *Proc. ICLR*, 2020, pp. 1–22.

[13] L. Kocsis and C. Szepesvári, "Bandit based monte-carlo planning," in *Proc. ECML*, 2006, pp. 282–293.

[14] A. Radford, J. Wu, R. Child, D. Luan, D. Amodei, and I. Sutskever, "Language models are unsupervised multitask learners," *OpenAI Blog*, vol. 1, no. 8, p. 9, 2019.

[15] M. W. U. Rahman et al., "Quantized transformer language model implementations on edge devices," 2023, *arXiv:2310.03971*.

[16] Y. Bondarenko, M. Nagel, and T. Blankevoort, "Understanding and overcoming the challenges of efficient transformer quantization," 2021, *arXiv:2109.12948*.

[17] Z. Li and Q. Gu, "I-ViT: Integer-only quantization for efficient vision transformer inference," in *Proc. ICCV*, 2023, pp. 17065–17075.

[18] C. Gong et al., "Nasvit: Neural architecture search for efficient vision transformers with gradient conflict aware supernet training," in *Proc. ICLR*, 2021, pp. 1–18.

[19] K. T. Chitty-Venkata, M. Emani, V. Vishwanath, and A. K. Somani, "Neural architecture search for transformers: A survey," *IEEE Access*, vol. 10, pp. 108374–108412, 2022.

[20] Y. Guo et al., "Nat: Neural architecture transformer for accurate and compact architectures," in *Proc. NeurIPS*, 2019, pp. 1–12.

[21] S. Li et al., "Hyperscale hardware Optimized neural architecture search," in *Proc. ASPLOS*, 2023, pp. 343–358.

[22] S. Rani, M. Chauhan, A. Kataria, and A. Khang, "IoT equipped intelligent distributed framework for smart healthcare systems," in *Towards the Integration of IoT, Cloud and Big Data: Services, Applications and Standards*. Singapore: Springer, 2023, pp. 97–114.

[23] B. L. R. Stojkoska and K. V. Trivodaliev, "A review of Internet of Things for smart home: Challenges and solutions," *J. Cleaner Prod.*, vol. 140, pp. 1454–1464, Jan. 2017.

[24] Y. Hu et al., "Pipeedge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices," in *Proc. DSD*, 2022, pp. 298–307.

[25] X. Guo, A. D. Pimentel, and T. Stefanov, "Automated exploration and implementation of distributed CNN inference at the edge," *IEEE Internet Things J.*, vol. 10, no. 7, pp. 5843–5858, Apr. 2023.

[26] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Robustly executing DNNs in IoT systems using coded distributed computing," in *Proc. DAC*, 2019, pp. 1–2.

[27] J. Zhou et al., "ElasticDL: A Kubernetes-native deep learning framework with fault-tolerance and elastic scheduling," in *Proc. WSDM'16*, 2023, pp. 1148–1151.

[28] Y. Ozaki, Y. Tanigaki, S. Watanabe, M. Nomura, and M. Onishi, "Multiobjective tree-structured Parzen estimator," *J. Artif. Intell. Res.*, vol. 73, pp. 1209–1250, Apr. 2022.

[29] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Trans. Evol. Comput.*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[30] X. Ma, G. Fang, and X. Wang, "LLM-pruner: On the structural pruning of large language models," in *Proc. NIPS*, 2023, pp. 1–19.

[31] R. Xiong et al., "On layer normalization in the transformer architecture," in *Proc. Int. Conf. Mach. Learn.*, 2020, pp. 10524–10533.

[32] T. Chen et al., "TVM: An automated end-to-end optimizing compiler for deep learning," in *Proc. OSDI 18*, 2018, pp. 578–594.

[33] V. Ben et al. "IREE: An MLIR-based compiler and runtime for ML models from multiple frameworks." 2019. [Online]. Available: https://iree.dev/

[34] T. Wolf et al., "Huggingface's transformers: State-of-the-art natural language processing," 2019, *arXiv:1910.03771*.

[35] J. Ansel et al., "PyTorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation," in *Proc. ASPLOS*, 2024, pp. 929–947.

[36] J. Reed, Z. DeVito, H. He, A. Ussery, and J. Ansel, "Torch. FX: Practical program capture and transformation for deep learning in python," in *Proc. Mach. Learn. Syst.*, 2022, pp. 638–651.

[37] L. Tencent. "NCNN." 2017. [Online]. Available: https://github.com/Tencent/ncnn

[38] Z. Hou, Y. Huang, S. Zheng, X. Dong, and B. Wang, "Design and implementation of heartbeat in multi-machine environment," in *Proc. AINA'17*, 2003, pp. 583–586.

[39] L. Gao et al. (Zenodo, Genéve, Switzerland). *A Framework for Few-Shot Language Model Evaluation*. Dec. 2023. [Online]. Available: https://zenodo.org/records/10256836