An Autonomous Parallelization of Transformer Model Inference on Heterogeneous Edge Devices

Juhyeon Lee[*]
Dept. of Electric Engineering, Sogang University
Seoul, South Korea
ljuh0928@sogang.ac.kr

Insung Bahk
Dept. of Artificial Intelligence Convergence, Hallym University
Chuncheon, South Korea
insung3511@hallym.ac.kr

Hoseung Kim
Dept. of Electrical and Computer Engineering, Sungkyunkwan University
Suwon, South Korea
ghtmd123@skku.edu

Sinjin Jeong
Dept. of Electronic Engineering, Pusan University
Pusan, South Korea
sjjeong@pusan.ac.kr

Suyeon Lee
School of Computer Science, Georgia Institute of Technology
Atlanta, USA
sylee0506@gatech.edu

Donghyun Min[*][†]
Dept. of Computer Science and Engineering, Sogang University
Seoul, South Korea
mdh38112@sogang.ac.kr

ABSTRACT

The utilization of advancing transformer-based deep neural network (DNN) models in edge environments holds the promise of improving productivity for intelligent tasks. However, deploying these models on edge devices with limited resources encounters significant performance challenges. Previous solutions have attempted to distribute computation tasks across devices and perform parallel inferences but often fall short of meeting service-level objectives (SLO). This limitation arises from their inability to effectively harness parallelization in transformer-based models and consider the resource diversity of edge devices. In this paper, we propose Hepti, a practical framework designed to facilitate parallel inference of transformer-based DNN models on heterogeneous edge environments. Hepti is armed with: 1) an understanding of transformer model architecture to enable effective parallel inference and 2) dynamic workload optimization to adapt to changing network and device resource capabilities. Our evaluations confirmed that the Hepti autonomously assesses the resource diversity of edge devices and network status. Furthermore, Hepti achieves a maximum performance improvement of 49.1% and 37.1% compared to the local inference approach and state-of-the-art model parallelisms on the BERT-Large model.

CCS CONCEPTS

• **Computing methodologies** → **Parallel algorithms**; **Distributed algorithms**; **Cooperation and coordination**.

KEYWORDS

Transformer architecture, model parallelism and partitioning, heterogeneous edge environments

ACM Reference Format:
Juhyeon Lee, Insung Bahk, Hoseung Kim, Sinjin Jeong, Suyeon Lee, and Donghyun Min. 2024. An Autonomous Parallelization of Transformer Model Inference on Heterogeneous Edge Devices. In *Proceedings of the 38th ACM International Conference on Supercomputing (ICS '24), June 04–07, 2024, Kyoto, Japan.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3650200.3656628

[*]Both authors contributed equally to this research.
[†]Corresponding author.

1 INTRODUCTION

Edge AI is a paradigm of operating AI workflows near the users at the network's edge, close to data sources. Unlike the traditional AI workflow, which sends data to the cloud, Edge AI benefits from low latency and low network traffic [4, 13, 21, 42]. Edge AI enables GPU-trained models to be performed in various edge device environments [3, 21, 35], including non-GPU devices, expanding DNN models to real-world applications. This expansion is expected to contribute to the growth of the Edge AI market, which is projected to reach $3.1 billion by 2027 [10, 19, 24].

Recently, state-of-the-art DNNs, especially transformer-based models, have revolutionized intelligent tasks in various fields, such as computer vision and natural language processing [7, 8, 11, 20, 22, 36, 37]. The overwhelming effectiveness of transformer models has made their deployment in edge environments crucial, shaping the landscape of upcoming Edge AI. For instance, such integration can increase real-time productivity and personalized interactions like detecting heartbeat frequency [33] and voice recognition [6].

However, deploying transformer-based DNN models in practical edge environments poses a critical challenge due to performance delays, which fail to meet service-level objectives (SLOs). This is primarily attributed to the limited computing resources of most edge hardware, while transformer model inference processes demand significant computational resources [4, 16, 25]. A case in point is the BERT-Large model [7], which is a representative NLP transformer model occupying 336 million parameters [32]. Similarly, the vision transformer (e.g., ViT-Large) [9] requires approximately 478 billion

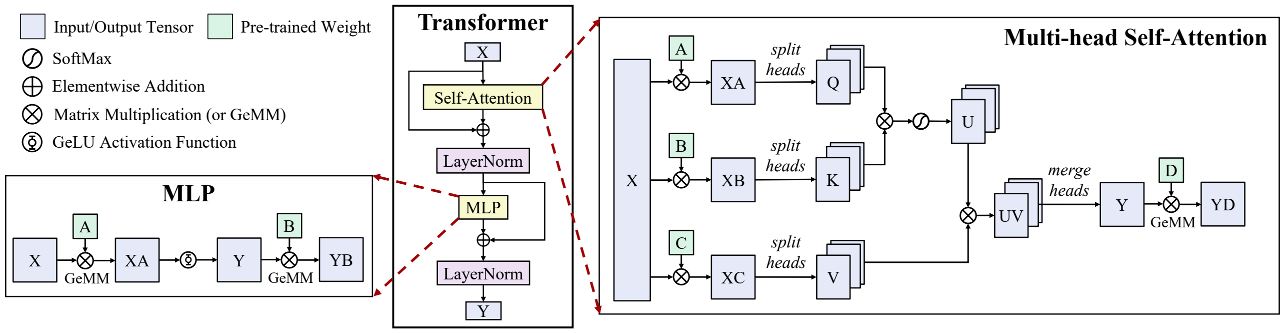


Figure 1: Block diagram of the general architecture of transformer. The transformer block itself is repeatedly used during inference. Note that if the ⊗ is accompanied by the GeMM notation, it indicates General Matrix Multiplication (GeMM); otherwise, it indicates Matrix-Matrix Multiplication (MatMul). In GeMM operation, MatMul and a bias addition are involved.

FLOPs for a single inference [27]. On the other hand, non-GPU edge devices commonly used for Internet-of-Thing (IoT) devices typically integrate low-performance processors, such as the ARM Cortex-A72 in the Raspberry Pi 4 board [2].

One promising approach to ensuring SLOs is to offload partial DNN model onto multiple nearby edge devices and parallelize the inference process [5, 12, 15, 17, 23, 32, 38, 40, 41]. The essence of DNN inference parallelization lies in accurately distributing computational tasks across edge devices [15, 39]. Attaining an optimal workload distribution requires a comprehensive understanding of the factors: (1) The architecture of the target DNN model; (2) The available network status for communication among devices; (3) The computing resources available on each device. If an excessive number of tasks are assigned to resource-limited local edge devices or a substantial workload is offloaded through limited network bandwidth to remote edge devices, there is a high likelihood of failing to meet the SLOs.

There have been prior works that involved parallel and distributed inference of DNN models in heterogeneous environments with varying computing resources [15, 17, 38, 40]. For example, both CoEdge [38] and EdgeFlow [26] proposed workload partitioning algorithms to parallelize CNN-based model inference in diverse edge environments. In CNN-based models, the output feature map of the convolution (Conv) operation is typically used as input for the next Conv, resulting in a significant volume of Conv operations. These algorithms were mainly designed to divide computation tasks specifically for these Conv operations. However, the architecture of the transformer model is more complex, featuring a distinct self-attention mechanism [37] that significantly differs from Conv operations. For a BERT [7] example, it consists of self-attention, layer normalization, and Multi-Layer Perceptron (MLP). As shown in Figure 1, the self-attention structure performs MatMul, GeMM, and SoftMax, rather than a sequence of Conv operations. CoEdge and EdgeFlow algorithms are not tailored for handling the structure of transformer models (Lack of (1)). Consequently, they cannot be directly applied to parallelize transformer-based model inference.

On the other hand, Megatron-LM [32] proposed a strategy for the transformer model parallelism across multiple same GPUs. Specifically, Megatron-LM reported the model parallelism mechanism for large language models by considering the operational logic of self-attention and MLP layers together while also taking into account the memory constraints of GPUs. However, it should be noted that Megatron-LM operates under the assumption of homogeneous environments. Thus, the predefined model parallelism in Megatron-LM cannot guarantee optimization when faced with varying memory constraints of heterogeneous edge environments. Furthermore, it does not provide workload partitioning algorithms that consider network dynamics and computing capabilities of different computing devices (Lack of (2) and (3)). Hence, the proposed solution is still less practical for achieving best distributed DNN inference in diverse edge environments.

In this paper, we propose Hepti (**H**eterogeneous **E**dges' **P**arallel **T**ransformer **I**nference), an inference framework, which supports heterogeneity-conscious parallelization of the transformer model on edge environments. Our main contributions are the following:

(1) Hepti supports three different parallel inference strategies by understanding the transformer architecture to cope with varying edge devices' memory status.
(2) Hepti generates the acceptable workload partitioning during inference by considering the network status and computing capabilities of heterogeneous devices.
(3) We have confirmed that Hepti reduces the latency of transformer model inference by up to 30.7% compared to existing transformer model parallelism methods.
(4) In the face of changing network conditions and computing capabilities, Hepti exhibits greater robustness than existing workload partitioning methodologies in heterogeneous edge environments.

2 BACKGROUND

2.1 Architecture of Transformer-based DNNs

The transformer model, a prominent deep learning architecture in natural language processing, uses a self-attention mechanism [34]. This mechanism computes the quantified relevance for each input element based on its relationships with all other elements in the input sequence. The self-attention is employed in parallel, creating multi-heads per input tensor. It allows the model to consider contextual information, ensuring an improved understanding of long-range connections of the input sequence. Figure 1 depicts the

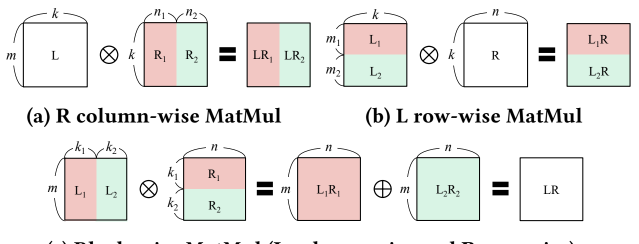


Figure 2: The divide-and-conquer approach-based MatMul. The L and R matrices are an activation tensor and a weight, respectively. In cases (a) and (b), the R and L matrices are split by column and row, respectively. These split sub-matrices are then computed in parallel and results are concatenated to form the final output. Case (c) additionally requires intermediate matrix transfer and summation operation.

basic structure of the transformer. It is composed of a multi-head self-attention block and a feed-forward block such as MLP. Both layer normalization and residual connection exist at the beginning and end of the self-attention and MLP blocks, respectively. The details of the components are as follows.

Multi-head Self-Attention. In the multi-head self-attention block, Query (Q), Key (K), and Value (V) are obtained by multiplying the input with each corresponding weight matrix and splitting them by the number of heads. For each head, attention scores are calculated by performing MatMul for Q and K. Next, the softmax function is used for these scores, generating attention weights (U). These weights are used to compute a weighted sum of the V, generating an output representation of each head (UV). After the outputs of each head are merged (Y), the final multi-head self-attention output is produced through GeMM.

Multi-Layer Perceptron. The input of the MLP block is the output that comes from the self-attention mechanism. This input undergoes a linear transformation using GeMM operation. Subsequently, the result is passed through an activation function, such as the GeLU [14]. The activation function introduces non-linearity to the model, allowing it to capture features in the data. Lastly, the final output is generated through another GeMM operation, aiming to capture the higher-level features within the input.

Layer Normalization and Residual Connection. The final output of the self-attention and MLP block is added to the original input (residual connection) and then normalized across all channels (layer normalization) within the input. It helps stabilize training and prevents vanishing gradient problems.

2.2 Prior Works for DNN Model Parallelism

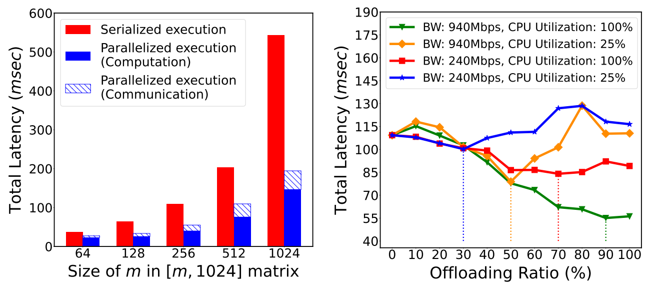
Within the domain of DNN model parallelism, the common approach involves distributing tensor operations across multiple computational devices. An example of such an approach is the fused-tile partitioning (FTP) technique [40]. The FTP disassembles Conv layers into sub-tasks for distribution among edge devices. The FTP process entails fusing layers and backpropagating the input's receptive field to compute the output tensor for each task. It can reduce the memory traffic when accessing tensor data for fused Conv layers. However, an issue arises due to the potential input tensor overlap, leading to redundant computations, such as halo cells within tiled convolutions. Moreover, the FTP is limited to CNN models, diverging notably from transformer architectures.

In contrast, the state-of-the-art approaches [29, 32] have been proposed for parallelizing both multi-head self-attention and MLP blocks of the transformer model within either multiple GPUs or TPUs, respectively. For multi-head self-attention, they distribute the 1-dimensional tiled (1D-tiled) weight parameters along with a column for MatMul operations associated with Q, K, and V across computing devices (depicted as (a) in Figure 2). This design allows each MatMul calculation corresponding to each attention head to be performed independently on a GPU, eliminating the need for inter-GPU communication for self-attention. For the last GeMM operation within self-attention, it is confirmed that further splitting the weight into 2-dimension (2D-tiled) makes more efficient in terms of communication cost according to the study [29].

In the case of MLP of the Megatron-LM, the weight of the first GeMM is split by columns in an R column-wise manner, while the weight of the second GeMM is split by rows in a block-wise manner (illustrated as (c) in Figure 2). The design allows the second GeMM independently on each GPU using the output of the first GeMM without communication. In the case of MLP of the study [29], the weight of the first GeMM is 2D-tiled. The weight of the second GeMM is also 2D-tiled and the activation tensor is 1D-tiled by columns (illustrated as (c) in Figure 2 with the 2D-tiled weight in the case). Compared to the Megatron-LM, in which partial-sum must be transferred between devices, this method is more efficient in terms of communication costs because it requires only exchanges for partial-sum involved in tiled areas of the final output. However, they unavoidably need all-reduce (i.e., collective communication consisting of reduce-scatter and all-gather operations) operation for the intermediate results among devices to perform the subsequent operation in self-attention and MLP block. Their model parallelism also assumes homogeneous GPUs or TPUs rather than heterogeneous environments. Thus, they cannot determine the acceptable partitioning for distributing workloads in heterogeneous edge environments. This highlights the need for a more practical parallelism strategy to achieve acceptable distributed DNN inference across diverse edge devices.

2.3 Heterogeneity-aware Workload Partitioning

The workload of the neural network encompasses the tensor operations needed for layer execution. In previous research [15, 17, 38], various partitioning methods have been explored to offload specific tensor operations within CNN workloads onto different heterogeneous devices, enabling parallel processing. The primary aim of these methods is to satisfy the service-level objectives (SLOs), such as latency metrics. To achieve this goal, they collect device-specific profiled data and use this information to determine how workloads should be partitioned before inference. The profiled data includes several parameters such as computing intensity, CPU frequency, maximum available memory capacity for each device, and network status. Based on the profiled data, they make optimal partitioning decisions for parallel inference on heterogeneous devices. Specifically, they determine how to divide the input tensor along the height dimension in CNN models, allowing each device to execute



(a) The execution time of serial and parallel MatMul operation. **(b) A comparison of total latency depending on offloading ratio.**

Figure 3: Impact of parallelization for MatMul operation and change of optimal workload partitioning point.

tiled Conv operations. This process is only activated during the setup phase whenever a DNN application is launched. Once the decision is laid out, their DNN inference runtime engines statically adhere to the predefined partitioning decision until the entire task is completed. However, when dealing with transformers that integrate both self-attention and MLP components, applying conventional partitioning methodologies dedicated to Conv operation is not immediately feasible. Furthermore, if the actual values of profiled data change during the inference phase, it can fail to meet the SLOs. Since existing partitioning algorithms operate under the assumption of static network or device capabilities, they do not change predetermined decisions.

3 MOTIVATION

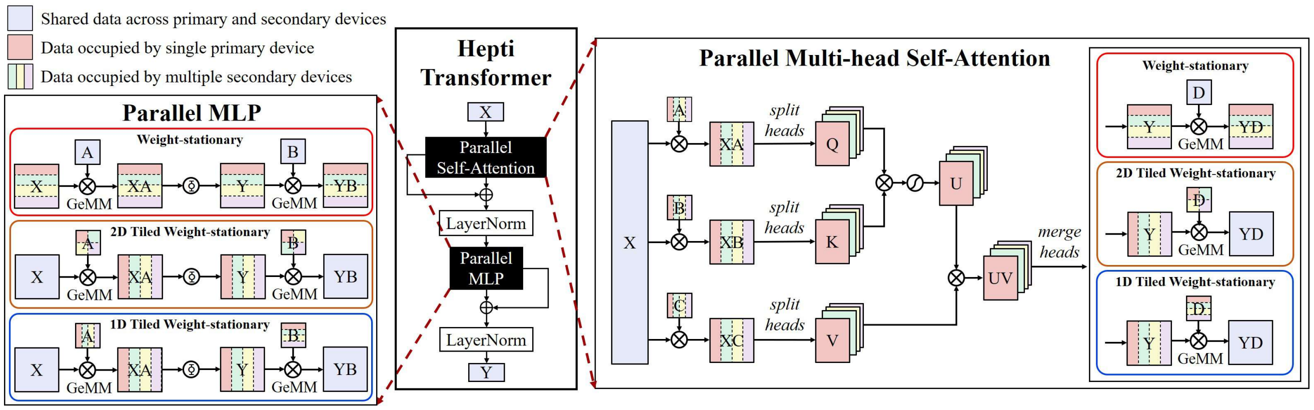
Lack of Understanding of Transformer Parallelism. Existing workload partitioning algorithms designed for parallel inference on heterogeneous devices lack an understanding of the transformer architecture. These algorithms were initially tailored for partitioning CNN-based models, primarily relying on dividing tensor by height dimension for Conv operation. In contrast, the transformer architecture heavily relies on MatMul and GeMM operations. This primitive operation is frequently used in self-attention and MLP. Therefore, as depicted in Figure 2, MatMul-dedicated parallelization strategies are required based on matrix split in a row or column direction. In order to quantitatively analyze how the partitioning approach in Figure 2 contributes to the actual performance gain of MatMul operation in an edge environment, we conduct experiments for the following scenarios. The edge device that triggers DNN inference is a primary device and another is a secondary device. The detailed experimental setup is described in Section 5.

- Scenario 1: When MatMul operation is serialized on a single primary edge device
- Scenario 2: When MatMul operation is parallelized on two heterogeneous edge devices (Case (b) in Figure 2)

Figure 3 (a) shows the execution time of each scenario. We conducted MatMul of two matrices with sizes [m, 1024] and [1024, 1024] by varying the value of m to change the matrix size. The network bandwidth and the offloading ratio from primary to secondary devices are fixed at 940 Mbps and at 90%, respectively. Because self-attention involves five MatMul operations in total, we performed MatMul five times in this experiment. In Scenario 1, the entire MatMul takes place only on a primary device, where the computation delay itself accounts for the total latency of MatMul. In contrast, in Scenario 2, the execution time is a summation of the maximum computation delay for divided MatMul operations and the communication delay for tensor offloading. According to Figure 3 (a), the execution time of Scenario 2 decreased by at least 25.1% and by as much as 64.2% compared to Scenario 1. This is mainly due to the significant reduction in computation delay in Scenario 2 compared to the communication delay incurred. For example, when the size of the MatMul is [256, 1024], Scenario 1 had a computation delay of 109.3 ms. On the other hand, in Scenario 2 where MatMul is parallelized, each device's computation delay decreased by a maximum of 34.2 ms. The communication delay showed only 20.8 ms. We observed the total execution time eventually shortened to 55.1 ms. The performance gains obtained in the parallel processing of matrices between heterogeneous devices remain consistent even when the size of the MatMul operation varies. We recognized the need to devise a parallelization approach for the transformer model by applying the parallel operation method of MatMul to the conventional inference workflow.

Insufficient Workload Partitioning for Heterogeneous Devices. The DNN workload partitioning involves deciding how tensor operations within a layer should be divided and offloaded to different devices for processing. However, earlier approaches to transformer model parallelism focused solely on parallelizing layer execution within homogeneous environments. The absence of practical workload partitioning logic is crucial because the overall performance relies on the computing capabilities of individual devices involved in parallel execution, as well as the network bandwidth. Figure 3 (b) provides a quantitative analysis of overall latency fluctuations while varying offloading ratios for five iterations of MatMul operation. In this experiment, we gradually adjusted the MatMul workload on the secondary device in 10% increments, upon the two network bandwidths (240 and 940 Mbps) and two computing capabilities (100% and 25% CPU utilization) of the secondary device. The value of m of the entire MatMul was kept fixed at 256. The detailed experimental setup is described in Section 5.

At a network bandwidth of 940 Mbps and CPU utilization of 100%, we observed that increasing the offloaded tensor operation from the primary device to the secondary device reduced computation delay while increasing communication delay. To illustrate, at a 10% offloading ratio, computation delay accounted for 89.3% of the total latency, while communication delay constituted 10.69%. On the other hand, at a 100% offloading ratio point, computation delay and communication delay made up 57.5% and 42.5%, respectively. We identified the best offloading point, resulting in a total latency of only 55.1 ms at a 90% offloading ratio. When the CPU utilization is limited to 25%, while keeping the same network bandwidth, the optimal offloading ratio shifted from 90% to 50%. This is because the performance gain from the secondary device compared to the communication delay has decreased. A similar phenomenon was also observed with a 240 Mbps network bandwidth. Specifically, if CPU utilization decreases from 100% to 25%, the optimal offloading point shifts from 70% to 30%. This experimental result underscores the fact that determining the optimal offloading ratio depends on the unique computing capabilities of each device. Next, when the



Shared data across primary and secondary devices
Data occupied by single primary device
Data occupied by multiple secondary devices
Parallel MLP
Weight-stationary
2D Tiled Weight-stationary
1D Tiled Weight-stationary
X
A
XA
Y
B
YB
GeMM
Hepti Transformer
Parallel Self-Attention
LayerNorm
Parallel MLP
Parallel Multi-head Self-Attention
split heads
Q
K
V
C
XB
XC
U
UV
merge heads
D
YD

Figure 4: Block diagram of Transformer architecture used in Hepti.

network bandwidth is changed from 940 Mbps to 240 Mbps while keeping CPU utilization of 100%, we observed a change in the optimal offloading ratio from 90% to 70%. This is because each device's computation delay relative to the communication delay shifted as the network bandwidth decreased. We found that network bandwidth and computing capability variations also affect the optimal offloading ratio. These observations and implications motivated our research into the optimal parallel inference methods for MatMul-centric transformer models, taking into account the computing capabilities and network status of heterogeneous devices.

4 HEPTI DESIGN AND WORKFLOW

Hepti is a framework that supports both parallel inference of transformer model (§ 4.2) and a partitioning engine for dynamic workload partitioning (§ 4.3) in a heterogeneous environment. Hepti considers transformer architecture but provides three inference modes depending on how the weight is stationary for each edge device. It also takes into account variations in network status and computing capabilities of each edge device for acceptable workload partitioning decisions. With these technical functionalities, Hepti autonomously determines and employs the parallel execution strategies for transformer DNN models.

4.1 Overall Behavior of Hepti

When the inference is requested, then Hepti transitions to the profiling stage to identify each edge device available for inference and collect accurate profiles for the computing resources of each device. The profiling targets statistics data about the CPU frequency, available memory size, and network bandwidth of other devices. These factors are summarized in Table 1 (a) and will be leveraged for partitioning transformer workload.

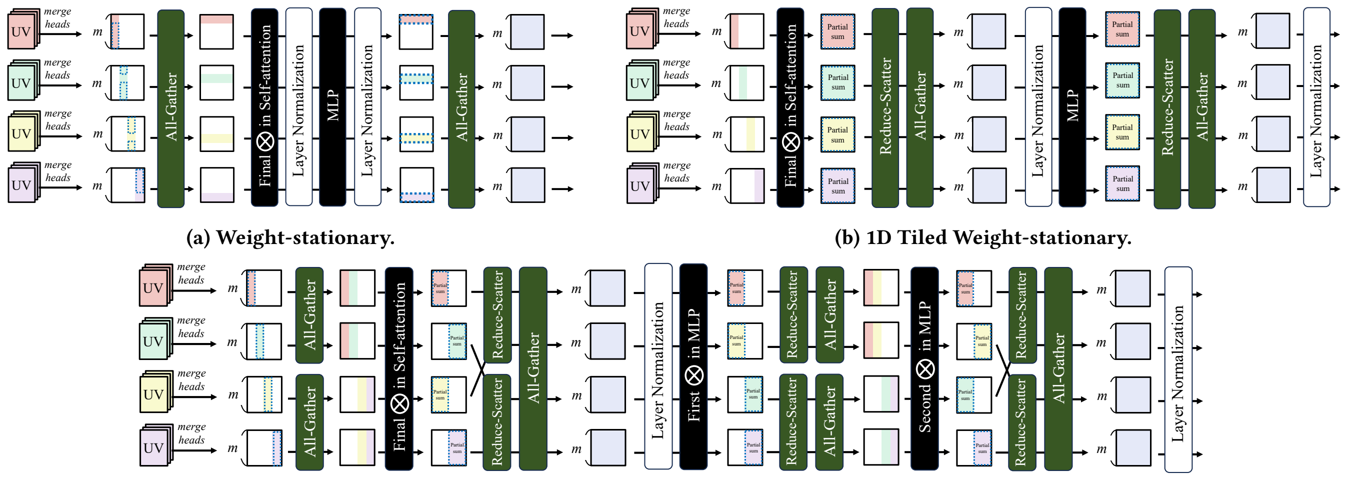
Once the profiling is completed, the Hepti initiates the parallel inference. During the runtime process, Hepti can repeatedly transition between the partitioning and actual parallel inference stages. At the partitioning stage, Hepti determines how much workload each participating device will handle for computation. At the parallel inference stage, Hepti performs the parallel inference executions. The repeated transitions between these two stages are intended to maintain the acceptable workload partitioning decision, which can be changed during the inference process.

Specifically, Hepti finds the acceptable workload partitioning that can minimize the delay of computation and communication between devices by using the proposed dynamic workload partitioning. If partitioning is performed in the middle of the inference process, Hepti does not transition to the profiling stage again. Instead, the primary device identifies the available memory size and computing intensity of the secondary devices. The size of memory information is conveyed at every moment when the secondary device returns intermediate inference results or exchanges tensors with the primary device. The primary device also estimates the computing intensity of the secondary device based on the time taken until the previously requested parallel inference results are returned and finally approximately estimates the network bandwidth. Whenever the workload partitioning decision for each device is made, the primary device offloads the allocated tensor workload to other devices and initiates their inferences. Simultaneously, the inference of the primary device begins.

The profiling and partitioning are triggered by the primary device while the runtime process of model inference involves parallel execution by both the primary and secondary devices. Any edge device can perform dual roles as the primary and the secondary device, owing to its capability for in-device transformer inference.

4.2 Hepti Inference Workflow

Once the workload partitioning decision is made, Hepti can perform parallel inference using the primary and the secondary devices. Figure 4 illustrates the three types of parallel inference workflows of Hepti. If the available memory on the secondary devices for offloading operations is sufficient to load the weight parameters, Hepti operates in a Weight-stationary (WS) manner. Motivated by Megatron-LM [32] and to cope with the case where the available memory of the secondary devices is severely limited, Hepti also supports 2D-tiled Weight-stationary (2D-tiled WS), and 1D-tiled Weight-stationary (1D-tiled WS). These manners are based on a method of splitting weight parameters to fit within the available memory size of secondary devices. According to the study [29], it has been observed and validated that the 2D-tiled WS manner exhibits superior performance when the number of computing devices (or cores) exceeds 16. Otherwise, the 1D-tiled WS manner outperforms. Consequently, in the design strategy adopted by



(a) Weight-stationary.

(b) 1D Tiled Weight-stationary.

(c) 2D Tiled Weight-stationary.

Figure 5: Illustration of communication and computation of tensors during transformer inference in three manners of Hepti.

Hepti, the 2D-tiled WS manner is prioritized in the case where a scalable inference design is needed due to a substantial number of devices and limited memory capacity. Conversely, the 1D-tiled WS manner is employed in alternative scenarios. The mode of Hepti is determined during the dynamic workload partitioning algorithm, described in Section 4.3.

Weight-stationary (WS) manner. First, in Hepti, the initial input data generated near the edge is transformed into matrix form. Then, it is copied to each secondary device to perform transformer operations in parallel. Afterward, Hepti performs R column-wise MatMul for K, Q, and V in self-attention with input matrix. It allows for facilitating subsequent splitting and merging operations on the resulting matrix, which are carried out with respect to attention heads. Then, for the last GeMM operation in self-attention, the WS uses the L row-wise MatMul. Accordingly, certain portions of matrices from each device are interchanged to construct input matrices for GeMM, organized in a row-wise partitioned manner. When using L row-wise MatMul, each device can independently perform computations up to the final layer normalization of the transformer. Figure 5 (a) illustrates how the matrix tensors of each device are exchanged and processed in Hepti's WS mode. In Figure 5 (a), each device exchanges data using all-gather communication for the final GeMM operation in self-attention. Consequently, each device acquires all tensors along with the row dimension, enabling L row-wise MatMul operations. Thus, each operation can be performed independently during the next layer normalization and MLP without further exchanges. After the iteration of the transformer, all tensors held by each device are propagated to prepare for the next iteration.

1D-tiled weight-stationary (1D-tiled WS) manner. Once all devices occupy the initial input matrix, the operation process of self-attention is identical to the Weight-stationary (WS) manner. To leverage the inherent parallelism in multi-head attention operations within self-attention, Hepti performs an R column-wise MatMul, enabling the matrix results corresponding to each attention head to be used as inputs directly on each device. However, the 1D-tiled WS manner uses block-wise MatMul to perform the final GeMM of self-attention. This approach divides the weight parameters, which are often larger than the input size. After the MatMul computation, collective communication such as all-reduce, involving reduce-scatter and all-gather operations between devices and additional matrix addition operations are needed, as illustrated in Figure 5 (b). After self-attention, and during the layer normalization operation, both devices must perform the same operation for the entire matrix. The reason for redundant layer normalization operations on each device is to prepare for the subsequent R column-wise parallel MatMul operation in the MLP. The R column-wise parallel MatMul also divides weight parameters across devices to save memory. The final GeMM in MLP is again performed using block-wise MatMul, necessitating an all-gather operation for communicating tensors at the end.

2D-tiled weight-stationary (2D-tiled WS) manner. The 2D-tiled WS manner still follows the above process of the self-attention except the last GeMM operation. For the last GeMM, this approach takes block-wise MatMul as a basic but divides weight parameters along with both row and column. Specifically, to perform block-wise MatMul when weights are split on a row-wise basis, it is necessary to gather the columns of the tensor corresponding to the portion of the tensor operated on the rows of the weight. For instance, assuming four devices utilize 2D-tiled WS, as depicted in Figure 4, the first two devices share tensors, while the remaining two devices share tensors through an all-gather operation, as illustrated in Figure 5 (c). As a result, each device generates partial sums for the block-wise MatMul. To obtain the final result of the block-wise MatMul, an all-reduce operation is subsequently performed on the partial sums, concluding the GeMM operation of self-attention. This all-reduce process is also essential for the two GeMM operations in the MLP. Although the implementation of the 2D-tiled WS manner is more difficult compared to the 1D-tiled WS manner, it not only allows for a further reduction in the size of the weight split across devices but also proves to be communication-efficient due to the minimized amount of tensor movement between devices.

Hepti's WS manner involves relatively fewer tensor exchanges and computations compared to the 1D- and 2D-tiled WS manners. This is because the L row-wise parallel MatMul approach is used instead of block-wise MatMul. However, if the available memory on the secondary device significantly decreases during inference in Hepti's WS mode, it leads to an out-of-memory issue, forcing the inference to terminate prematurely. To prevent this, Hepti continuously monitors the available memory status of devices and dynamically switches between three modes. The memory status information is conveyed during intermediate tensor exchanges between the primary and the secondary device during inference. With the received memory information, Hepti can repeatedly execute the dynamic workload partitioning algorithm when the tensors are fully merged on the primary device at the specific step of the inference workflow.

4.3 Dynamic Workload Partitioning

The goal of Hepti is to meet the SLO by controlling the amount of workload distribution from a latency perspective for parallel inference. To achieve this, Hepti needs to solve the problem of finding an acceptable workload partitioning decision, considering the computational capacity of each device and the network status. To address this optimization problem, Hepti's partitioning engine employs a dynamic workload partitioning algorithm. This algorithm autonomously makes an acceptable decision during the runtime inference phase. For example, Hepti can make a new partitioning decision at the beginning of self-attention, as the primary device possesses the entire tensor at that point. In our dynamic workload partitioning algorithm, we set the mathematical model which is composed of constraints based on profiled data and solves an objective function to meet the latency deadline of SLO. In the following, we define a problem formulation using notations in Table 1. Then, we present the dynamic workload partitioning algorithm.

Problem Formulation. Consider e edge devices and the MatMul operation between matrices of dimensions $[m, k]$ and $[k, n]$ where the m, k, and n are integers. There are several numerical constraints when deciding the partitioning size of each device. Equations (1) and (2) represent size constraints when splitting tensors. Specifically, these require that the a_i values in π must be non-negative integers. The concatenation size of all partitioned tensors, along with the split direction, should match the length of the specific direction. For example, in the case of L row-wise parallel MLP of Hepti, the sum of a_i values must equal the m value.

$$\sum_{i \in N} a_i = \begin{cases} m & \text{if L row-wise MatMul} \\ n & \text{if R column-wise MatMul} \\ k & \text{if Block-wise MatMul} \end{cases} \tag{1}$$

$$a_i \geq 0, i \in N \tag{2}$$

Then, using Equation (3), we determine the workload size as the magnitude of elements involved in the dot-product operation for MatMul. The size of workload $r_{b,i}$ determined by a_i is constrained by the Equation (4). This equation restricts the workload size of the transformer block to ensure that it does not exceed the available memory of the device.

Table 1: Notation and description of arguments of the profiling and partitioning engine of Hepti.

Argument	Description
$N = [1, 2, \ldots, e]$, $e \in \mathbb{N}$	The set of available edge devices including the primary edge device.
ρ_i, $i \in \mathbb{N}$	Computing intensity (# of CPU cycles required to FLOP) of i-th device.
f_i, $i \in \mathbb{N}$	Average CPU frequency of i-th device during feed-forward process.
s_i, $i \in \mathbb{N}$	The available memory size of i-th device, excluding system memory.

(a) Each argument used in profiling of the Hepti.

Argument	Description
$B = [1, 2, \ldots, b]$, $b \in \mathbb{N}$	The sequence of transformer blocks (e.g., self-attention, layer normalization, and MLP).
$\pi = [a_1, a_2, \ldots, a_i]$, $i \in N$, $a_i \in \mathbb{N}$	The set of the sizes in the split direction of the partitioned tensors (matrices) each device covers.
$r_{b,i}$, $b \in B, i \in N$	The workload size of partitioned MatMul that need to be computed on each device for specific transformer block.
$c_{b,i}$, $b \in B, i \in N$	The size of data to be received from other devices to proceed the specific transformer block.
D (msec)	The service-level deadline as an objective variable of the partitioning engine.
$BW_{i,j}$ (Mbps), $i \in N$	Bandwidth between i-th and j-th devices.
T^c_{bi} (msec), $b \in B, i \in N$	Computation inference latency in b-th block on i-th device
T^x_{bi} (msec), $b \in B, i \in N$	Communication delay in b-th block of i-th device
T (msec)	Maximum sum of T^x_{bi} and T^c_{bi} for each block and device

(b) Each argument used in partitioning of the Hepti.

$$r_{b,i} = \begin{cases} k \cdot n \cdot a_i & \text{if L row-wise MatMul} \\ k \cdot m \cdot a_i & \text{if R column-wise MatMul} \\ a_i \cdot m \cdot n & \text{if Block-wise MatMul} \end{cases} \tag{3}$$

$$r_{b,i} \leq s_i, b \in B, i \in N \tag{4}$$

While executing a single transformer block, Hepti experiences delays in both computation and communication aspects. We estimate each delay by approximating the computing cycles of specific partitioned tensor operations. The total computation latency of the i-th partition is evaluated using Equation (5) and (6). To estimate the required CPU clock cycles for processing $r_{b,i}$, we calculate $\rho_i \cdot r_{b,i}$. Then, we divide it by f_i to calculate the overall computation time.

We evaluate the total communication latency corresponding to the i-th device based on Equation (6). The $c_{b,i}$ can be derived from a_i. For example, $c_{b,i}$ is $(m - a_i) \cdot n \cdot 4$ (byte size) for the last layer normalization block in a WS manner. By dividing $c_{b,i}$ by network bandwidth, we measured the time required for the necessary communication. The latency, T'_{bi}, required for collective communication (e.g., all-gather and reduce-scatter) is aggregated across devices, and the resulting sum is denoted as T^x_b. Equation (7) defines the total latency T, which is a sum of computation and communication latency. The T represents the longest time taken to perform a single inference. Finally, our target objective is to decide on an optimal π solution, minimizing the T so that it does not exceed the execution deadline D.

$$T^c_{bi} = \frac{\rho_i \cdot r_{b,i}}{f_i} \tag{5}$$

$$T^x_b = \sum_{i \in N} T'_{bi}, \quad T'_{bi} = \frac{c_{b,i}}{BW_{i,j}} \qquad b \in B \quad i, j \in N \tag{6}$$

$$T = \sum_{b \in B} \max_{i \in N}(T^c_{bi} + T^x_b) \tag{7}$$

Hence, our defined Hepti's inference optimization can be formulated as the following Equation (8).

$$\mathcal{P}1 : \min T \ s.t. \ T \le D, \ (1), (2), (4) \tag{8}$$

Linear Programming-based Approximation. Solving $\mathcal{P}1$ is not practical in terms of time complexity since $\mathcal{P}1$ has non-convex optimization constraints and objective functions. Particularly, it is challenging due to the discrete nature of the integer variable a_i in Equation (1). Thus, by introducing continuous real numeric variable λ to the $\mathcal{P}1$, we transform it into a linearly approximated problem.

First, we set the variable λ to the a_i as shown in Equation (9). The range of λ is constrained by Equation (10). Then, optimization of our dynamic partitioning algorithms can be rewritten as the following Equation (11). The objective function and other constraints are still linear with a_i, which is now continuous. For other numerical constraints of Equation (10), they have still linear relationships. Hence, $\mathcal{P}2$ is a mixed integer linear programming problem.

$$a_i = \begin{cases} \lambda_i \cdot m & \text{if L row-wise MatMul} \\ \lambda_i \cdot n & \text{if R column-wise MatMul} \\ \lambda_i \cdot k & \text{if Block-wise MatMul} \end{cases} \tag{9}$$

$$\sum_{i \in N} \lambda_i = 1, \ s.t. \ 0 \le \lambda_i \le 1, i \in N \tag{10}$$

$$\mathcal{P}2 : \min T \ s.t. \ T \le D, \ (4), (9), (10) \tag{11}$$

Algorithms. Our dynamic workload partitioning engine takes as inputs the (N, B, D) tuple, as well as (ρ, f_i, s_i) for each device and $BW_{i,i+1}$ between devices. If all secondary devices are unavailable, then Hepti executes serial transformer inference. Otherwise, the engine checks s_i and determines the Hepti's operation mode. The engine conservatively decides on 1D-tiled WS or 2D-tiled WS manner if at least one of the devices participating in parallel inference does not have sufficient memory to load the model parameters. Hepti uses 1D-tiled WS when there are fewer than 16 devices, and 2D-tiled WS otherwise. Then, the engine uses a mixed-ILP solver to find λ_i of π that can solve $\mathcal{P}2$. If there are elements with λ_i equal to zero in π, the corresponding devices do not participate in parallel inference. In this case, we remove the device from N and re-evaluate the partitioning decision for the remaining devices. For the obtained a_i values, rounding to the nearest integer is applied to determine the final element of π. The resulting π from our partitioning finally contains a decision on how much transformer model inference workload is allocated to each device. In the case of the 2D-tiled WS manner, the aforementioned specific-axis partitioning algorithm is simply executed twice, once for the row and once for the column direction to split the matrix in 2-dimension.

Table 2: Testbed configurations for Hepti evaluation.

Primary device	**Specifications**
CPU	ARM Quad core Cortex-A72 @ 1.8GHz
Memory	8GB LPDDR4-3200 SDRAM
Network	2.4 GHz LAN & Gigabit Ethernet
Kernel version	Linux-Kernel-5.15.0-1035-raspi
Secondary device	**Specifications**
CPU	Intel(R) Dual Core i3-8130U @ 2.20GHz
Memory	4GB DDR4 SDRAM
Network	2.4 GHz LAN & Gigabit Ethernet
Kernel version	Linux-Kernel-6.2.0-26-Generic

5 EVALUATION

5.1 Experimental Setup

Our evaluation was conducted on a Raspberry Pi 4 board and non-GPU server as a heterogeneous edge computing platform. To increase heterogeneity, we leveraged `cpulimit` software tool to limit the CPU utilization and `tc` tool to change network bandwidth between devices. The detailed configurations of the testbed are described in Table 2. We used PyTorch [28] to implement the deep learning inference engine of Hepti and PyTorch Distributed communication package as a distributed backend framework. All experiments are initiated from the Raspberry Pi 4 board, which acts as a role of the primary edge device. The primary edge device also performs a dynamic workload partitioning algorithm based on `IBM DOcplex` [1], which provides a library for linear programming. Our transformer model workloads include BERT-large and MobileBERT, representing BERT [7] variants to examine models that have a fundamental transformer architecture. Additionally, our approach extends to other models sharing basic transformer structures such as Whisper [30], and Vision Transformers [8] without significant modifications. However, models necessitating a fundamentally different attention mechanism lie beyond the scope of our approach. We compare and evaluate Hepti with the following baselines by processing text input with 256 tokens in a single batch. First, we compare our parallel inference approach against

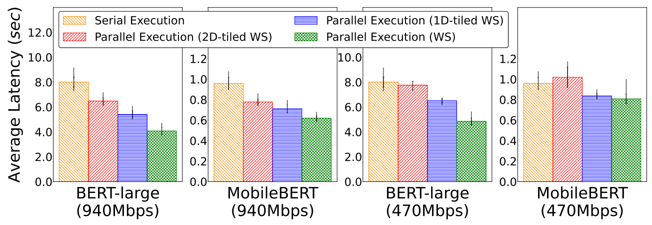


Figure 6: End-to-end latency of serial execution and parallel execution with 2D-tiled WS, 1D-tiled WS and Hepti parallel inference approaches for two network bandwidth variants. In BERT-large with WS manner, at 940 Mbps, 87.5% of the workload was offloaded, while at 470 Mbps, 93.7% of the workload was offloaded.

the parallelism approach of previous studies [29, 32], while keeping the partitioning decision the same. We also compare our dynamic workload partitioning algorithm approach against the partitioning approach of CoEdge [38], while our parallel inference methods for the transformer are equally provided.

5.2 Parallel Inference Performance Comparison

To see the effectiveness of our parallel inference method, we measured and compared the end-to-end latency of the Hepti with the approaches of local inference and 1D- and 2D-tiled WS manner-based inferences which are taken by previous studies [29, 31, 32]. In this experiment, we control our partitioning algorithm to be performed only once before the inference to determine the workload partitioning ratio. A memory size of 1.63 GB was allocated to the secondary device to ensure that model parameters could be loaded. Figure 6 shows the average end-to-end latency for each inference methodology when the network bandwidth is 940 Mbps and 470 Mbps.

According to the results of the 940 Mbps environment of Figure 6, serial execution takes 8 and 0.95 seconds for BERT-large and MobileBERT, respectively. On the other hand, the 2D-tiled WS methodology takes only 6.4 and 0.77 seconds, respectively. This method performs parallel inference, minimizing the number of data communications between the two devices and saving 21.6% of the memory of the secondary device. Similarly, the 1D-tiled WS methodology takes only 5.4 and 0.71 seconds respectively. The 1D-tiled WS approach reduces 16.6% of the end-to-end latency compared to the 2D-tiled WS approach because of the number of devices. 2D-tiled WS manner demands more weight-slicing operations and more communication points than the 1D-tiled WS manner. However, as revealed in the previous study [29], the 2D-tiled WS approach can reduce communication overhead as the number of devices increases, leading to performance improvements. Consequently, this enables faster parallel inference compared to the 1D-tiled WS method in such environments.

We found that the parallel inference method in a WS manner significantly improves performance, taking only 4.07 seconds for BERT-large and 0.61 seconds for MobileBERT. This reduction in time corresponds to a decrease of 37.1% and 20.4% in total overhead compared to 2D-tiled WS-based inference. Also, compared to the parallel execution with 1D-tiled WS manner, Hepti's WS manner takes 24.6% and 13.1% less total overhead. In the parallel approach of 1D- and 2D-tiled WS, block-wise parallel MatMul is continuously used in both self-attention and MLP. In contrast, Hepti also supports L row-wise MatMul. Unlike block-wise MatMul, L row-wise MatMul does not require inter-device data transfers for partitioned matrices or additional matrix addition operations. Thus, it reduces performance overhead in MatMul (or GeMM).

Even when the network bandwidth decreases to 470 Mbps, Hepti's WS manners still outperform serial execution for BERT-large and MobileBERT as shown in Figure 6. Particularly, we confirm that Hepti, operating in a WS manner, reduces the latency for BERT-large by around 46.6% and 29.8% compared to 2D- and 1D-tiled WS manner each. However, the 2D-tiled WS and the 1D-tiled WS of the Hepti show longer end-to-end latency compared to the serial execution for MobileBERT. This is due to communication overhead outweighing performance gains from parallelization. Unlike large models, MobileBERT's lightweight nature contributes to low computational latency. Applying 1D- or 2D-tiled WS-based inference techniques to lightweight models actually incurs significant communication overhead, leading to overall performance degradation. More specifically, MobileBERT includes additional feed-forward network (FFN) layers that contain two MatMuls each like the MLP block. The 1D- and 2D-tiled WS manner requires communications in every FFN layer because they use the block-wise parallel MatMul. On the contrary, Hepti can execute all FFN layers without communication as WS manner uses L row-wise MatMul.

To identify the underlying factors influencing the overall inference performance of Hepti's parallelization strategies, we conducted measurements of both computational and communication overheads in each transformer component. Table 3 compares the amount of data transfer during the parallel inference in 1D-tiled WS and Hepti. In the case of 1D-tiled WS, we observed that 1.05 GB of data is transferred per device at the beginning of every layer normalization, which requires exchanges for the entire tensor. However, Hepti's WS manner uses L row-wise parallel MatMul operations. Thus, it reduces primary device data transfer to just 245.8 KB, an 88.3% reduction compared to the 1D-tiled WS manner. We have confirmed that the WS manner is more efficient in terms of in-device communication compared to other methods. Because our experimental setup uses only two devices, the case of the 2D-tiled WS manner is the same as the 1D-tiled WS manner in terms of communication points and transferred tensor size.

Table 3: The total byte size of tensors required for communication between devices during BERT-large inference.

Communication point	**1D-tiled WS manner**		**WS manner**	
	Pri. dev	**Sec. dev**	**Pri. dev**	**Sec. dev**
Before final GeMM in self-attention	N/A	N/A	114.7 KB	114.7 KB
Before first layer normalization	1.05 GB	1.05 GB	N/A	N/A
Before second layer normalization	1.05 GB	1.05 GB	N/A	N/A
After second layer normalization	N/A	N/A	131.1 KB	917.5 KB
Total	**2.1 MB**	**2.1 MB**	**245.8 KB**	**1.03 GB**

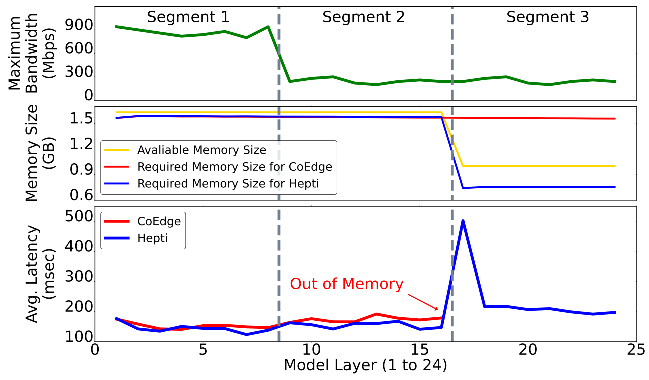


Figure 7: The inference latency of BERT-large under varying network status and available memory size of the secondary device. The inference process is composed of 24 transformer iterations in BERT-large.

To analyze computational overhead, we calculated the number of element-wise multiplications and additions required for MatMul and GeMM operations in self-attention and MLP for both 1D-tiled WS manner and WS manner of the Hepti with BERT-large. Our experiments revealed that the total number of element-wise multiplications was the same for 1D-tiled WS, and Hepti. However, 1D-tiled WS manner required 1,048,576 more element-wise additions compared to Hepti for each transformer layer. This is because 1D-tiled WS manner uses block-wise parallel MatMul method during the last GeMM of self-attention and the second GeMM of MLP, necessitating additional addition operations for sub-matrices. In contrast, Hepti in a WS manner, only uses L row-wise MatMul and does not require additional addition operations. Next, both 1D-tiled WS manner and Hepti perform layer normalization for the tensor data each device possesses. In the case of 1D-tiled WS, since both primary and secondary devices share the entire tensor data, they perform an equal number of normalization operations, which increases with the number of devices. This approach aims to minimize inter-device communication and the model size by partitioning weights of MatMul and GeMM operations. On the other hand, Hepti conducts layer normalization solely for the sub-matrices split in the row direction on both devices, regardless of the number of devices, resulting in a constant total workload. In the case of 2D-tiled WS, the total number of element-wise multiplications is the same as in the 1D-tiled WS approach. Also, the number of total addition operations is the same as in the 1D-tiled WS approach due to the experimental setting. Generally, the entire number of element-wise multiplications is identical in 1D- and 2D-tiled WS manners. In contrast, in the 2D-tiled WS method, the number of element-wise addition operations is influenced by the number of devices dividing each dimension of weights. In our experimental setup, the dimensions of weights are partitioned by the same devices as in the 1D-tiled WS.

5.3 Adaptability to Heterogeneous Settings

In this experiment, we used BERT-large to investigate the impact of Hepti's dynamic partitioning algorithm on parallel inference within a heterogeneous environment, comparing it to CoEdge. We

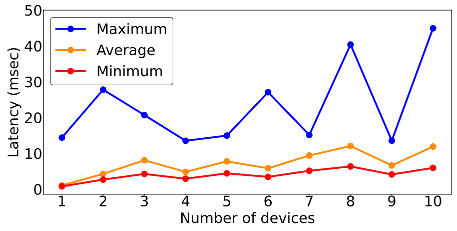


Figure 8: The performance overhead of dynamic workload partitioning under varying the number of devices.

considered variations in network bandwidth and available memory size on the secondary device. Since CoEdge lacks support for parallel inference with transformer models, we configured it to follow Hepti's parallel inference approach in WS manner. Figure 7 presents recorded parallel processing latency during the transformer layer computation point across different heterogeneous scenarios. In Segment 1, the network bandwidth is 800-940 Mbps, and there is sufficient memory space on the secondary device to accommodate model parameters. Specifically, a minimum of 1.63GB of memory is guaranteed to enable parallel inference in the WS manner on the secondary device. Therefore, both CoEdge and Hepti can perform parallel inference in WS mode during Segment 1.

Moving to Segment 2, the network bandwidth drops to 200-300 Mbps, causing CoEdge to experience a 12.42% latency increase compared to Hepti. The reason is that CoEdge sticks to the partitioning decision fixed during the setup phase before the inference initiation, with no modifications during the inference phase. Conversely, Hepti continues to deliver optimal performance because it operates a dynamic partitioning algorithm. This algorithm allows Hepti to adapt to changes in network bandwidth and generate optimal partitioning decisions during the inference phase.

Transitioning from Segment 2 to Segment 3, the available memory on the secondary device decreases from 1.63 GB to 1 GB. In this case, WS transformer parallel processing becomes infeasible due to limited memory space. In the case of CoEdge, the inference program halts due to out-of-memory issues. This is because CoEdge is unable to adjust its partitioning decisions during inference. In contrast, Hepti can dynamically switch to the 1D-tiled WS manner, which requires only 757 MB of memory space on the secondary device in the experiment. Thus, Hepti can sustain inference even in environments with reduced memory usage. Our experiment demonstrates that Hepti exhibits superior inference robustness in heterogeneous edge environments compared to CoEdge. Note that in a scenario where network bandwidth is also heavily constrained, the performance may degrade compared to serial inference. For example, when limited to a network bandwidth of less than 100 Mbps, Hepti's 1D-tiled WS experiences a 1.9% latency increase compared to the serial execution. This was attributed to communication overhead outweighing the computational gains from parallelization.

5.4 Overhead of Dynamic Workload Partitioning Algorithm

Hepti's dynamic workload partitioning algorithm operates iteratively, even during inference. To assess its impact on overall inference latency, we measured the performance overhead of this dynamic partitioning algorithm. Figure 8 illustrates how the partitioning algorithm's overhead varies as the number of participating devices in parallel inference increases. The algorithm's behavior is influenced by the values of computing intensity and network bandwidth in the input, so we conducted experiments with a hundred randomly sampled instances. Figure 8 reveals that as the number of devices increases up to ten, Hepti's average latency for the partitioning algorithm becomes negligible. For example, with ten devices, the average latency is only 11.89 milliseconds Compared to the previous end-to-end latency of Hepti for BERT-large at 940 Mbps network bandwidth (as shown in Figure 6), which was 4.07 seconds, the partitioning algorithm latency contributes to only 0.29%. However, the maximum execution time can reach 44.9 milliseconds, particularly when using ten devices. It occurs when a specific λ_i, a constituent of π, becomes zero, resulting in some devices being excluded from inference. This necessitates a rerun of the partitioning algorithm. Even so, the partitioning algorithm is iterated every 24 iterations of the BERT-large transformer, adding about 1.076 seconds of parallel inference delay. The total latency is still significantly less than the 8 seconds, a latency of local inference time for BERT-large at 940 Mbps bandwidth (as shown in Figure 6).

6 RELATED WORKS

DeepThings [40] proposed a method of distributing input tensor data to enable independent model inference on Convolution (Conv) layers. This method aims to measure and distribute the receptive field of the input in order to compute the output data with a specific range across homogeneous edge devices. MoDNN [23] partitioned the input tensor data of Conv layers and fully-connected layers on a mobile computing environment. In MoDNN, the workload is divided into segments using a greedy approach, and these segments are allocated more heavily to devices that possess higher computational capacity.

CoEdge [38] proposed a framework for parallel inference for Conv-centric CNN architecture across heterogeneous edge devices. CoEdge introduced an adaptive workload partitioning technique that considers not only available computing resources but also network bandwidth among devices. This workload partitioning aims to reduce both inference latency and energy consumption. EdgeFlow [17] redesigned and extended existing partitioning methods to suit the directed acyclic graph model. This adaptation was rooted in the assumption that the architecture of the DNN model is represented as a graph rather than a linear chain. Specifically, the partitioning algorithm of EdgeFlow comprehends the model graph's input-output relationship to allocate specific layer operations to each edge device. These studies successfully achieved the parallelism of CNN model inference. Hepti performs parallel inference following the CoEdge approach for CNN models.

In contrast, Megatron-LM [32] and several studies [29, 31] proposed model parallelism for transformer models. By leveraging MatMul parallelism, Megatron-LM examined the operational behavior of the transformer model. In Megatron-LM, a partitioning algorithm that takes into account the network conditions and computational capabilities of each heterogeneous device has not been proposed. Unlike an intra-layer partitioning-based transformer parallelism of these studies, an inter-layer model parallelism was suggested in PipeEdge [18]. Specifically, PipeEdge divides the batch of the input into multiple micro-batches, and pipelines the execution of these micro-batches across multiple edge devices.

7 CONCLUSION

This paper proposes Hepti, a practical inference framework designed to facilitate the transformer model parallelism in heterogeneous edge device environments. Hepti incorporates three parallelization methods for transformer inference and autonomously determines acceptable workload partitioning strategies for parallelization. The experimental results show the partitioning overhead is negligible while simultaneously improving the performance advantages of inference.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their insightful comments and suggestions, which greatly contributed to refining this work. This work was carried out by members of VERITROSS (VRTSS-1).

REFERENCES

[1] 2016. User's Manual for CPLEX. https://www.ibm.com/docs/en/icos/12.9.0?topic=cplex-users-manual.

[2] 2019. Raspberry Pi 4 Tech Specs. https://www.raspberrypi.com/products/raspberry-pi-4-model-b/specifications/.

[3] 2023. Edge AI and Vision Alliance. www.edge-ai-vision.com/.

[4] Jiasi Chen and Xukan Ran. 2019. Deep learning with edge computing: A review. *Proc. IEEE* 107, 8 (2019), 1655–1674.

[5] Elias De Coninck, Tim Verbelen, Bert Vankeirsbilck, Steven Bohez, Sam Leroux, and Pieter Simoens. 2015. Dianne: Distributed artificial neural networks for the internet of things. In *Proceedings of the 2nd Workshop on Middleware for Context-Aware Applications in the IoT.* 19–24.

[6] Li Deng, Geoffrey Hinton, and Brian Kingsbury. 2013. New types of deep neural network learning for speech recognition and related applications: An overview. In *2013 IEEE international conference on acoustics, speech and signal processing.* IEEE, 8599–8603.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805* (2018).

[8] A Dosovitskiy, L Beyer, A Kolesnikov, D Weissenborn, X Zhai, and T Unterthiner. 2020. Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[9] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, et al. 2020. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[10] Grand View Research. 2022. *Edge AI Market Size, Share and Trends Analysis Report By Component (Hardware, Software, Edge Cloud Infrastructure, Services), By End-use Industry, By Region, And Segment Forecasts, 2023 - 2030.*

[11] Meng-Hao Guo, Jun-Xiong Cai, Zheng-Ning Liu, Tai-Jiang Mu, Ralph R Martin, and Shi-Min Hu. 2021. Pct: Point cloud transformer. *Computational Visual Media* 7 (2021), 187–199.

[12] Ramyad Hadidi, Jiashen Cao, Michael S Ryoo, and Hyesoon Kim. 2020. Toward collaborative inferencing of deep neural networks on Internet-of-Things devices. *IEEE Internet of Things Journal* 7, 6 (2020), 4950–4960.

[13] Qiang He, Zeqian Dong, Feifei Chen, Shuiguang Deng, Weifa Liang, and Yun Yang. 2022. Pyramid: Enabling hierarchical neural networks with edge computing. In *Proceedings of the ACM Web Conference 2022.* 1860–1870.

[14] Dan Hendrycks and Kevin Gimpel. 2016. Gaussian error linear units (gelus). *arXiv preprint arXiv:1606.08415* (2016).

[15] Xueyu Hou, Yongjie Guan, Tao Han, and Ning Zhang. 2022. DistrEdge: Speeding up convolutional neural network inference on distributed edge devices. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS).* IEEE, 1097–1107.

[16] Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861* (2017).

[17] Chenghao Hu and Baochun Li. 2022. Distributed inference with deep learning models across heterogeneous edge devices. In *IEEE INFOCOM 2022-IEEE Conference on Computer Communications.* IEEE, 330–339.

[18] Yang Hu, Connor Imes, Xuanang Zhao, Souvik Kundu, Peter A Beerel, Stephen P Crago, and John Paul Walters. 2022. Pipeedge: Pipeline parallelism for large-scale model inference on heterogeneous edge devices. In *2022 25th Euromicro Conference on Digital System Design (DSD)*. IEEE, 298–307.

[19] International Data Corporation (IDC). [n.d.]. https://www.idc.com/getdoc.jsp?containerId=prAP50688623.

[20] Wonjae Kim, Bokyung Son, and Ildoo Kim. 2021. Vilt: Vision-and-language transformer without convolution or region supervision. In *International Conference on Machine Learning*. PMLR, 5583–5594.

[21] En Li, Liekang Zeng, Zhi Zhou, and Xu Chen. 2019. Edge AI: On-demand accelerating deep neural network inference via edge computing. *IEEE Transactions on Wireless Communications* 19, 1 (2019), 447–457.

[22] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. 2021. Swin transformer: Hierarchical vision transformer using shifted windows. In *Proceedings of the IEEE/CVF international conference on computer vision*. 10012–10022.

[23] Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. 2017. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*. IEEE, 1396–1401.

[24] Markets and Markets. 2023. Edge AI Software Market: Global Forecast to 2028. https://www.marketsandmarkets.com/Market-Reports/edge-ai-software-market-70030817.html.

[25] Sachin Mehta and Mohammad Rastegari. 2021. Mobilevit: light-weight, general-purpose, and mobile-friendly vision transformer. *arXiv preprint arXiv:2110.02178* (2021).

[26] Massimo Merenda, Carlo Porcaro, and Demetrio Iero. 2020. Edge machine learning for ai-enabled iot devices: A review. *Sensors* 20, 9 (2020), 2533.

[27] Deepak Narayanan, Mohammad Shoeybi, Jared Casper, Patrick LeGresley, Mostofa Patwary, Vijay Korthikanti, Dmitri Vainbrand, Prethvi Kashinkunti, Julie Bernauer, Bryan Catanzaro, et al. 2021. Efficient large-scale language model training on gpu clusters using megatron-lm. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 1–15.

[28] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[29] Reiner Pope, Sholto Douglas, Aakanksha Chowdhery, Jacob Devlin, James Bradbury, Jonathan Heek, Kefan Xiao, Shivani Agrawal, and Jeff Dean. 2023. Efficiently scaling transformer inference. *Proceedings of Machine Learning and Systems* 5 (2023).

[30] Alec Radford, Jong Wook Kim, Tao Xu, Greg Brockman, Christine McLeavey, and Ilya Sutskever. 2023. Robust speech recognition via large-scale weak supervision. In *International Conference on Machine Learning*. PMLR, 28492–28518.

[31] Jeff Rasley, Samyam Rajbhandari, Olatunji Ruwase, and Yuxiong He. 2020. Deepspeed: System optimizations enable training deep learning models with over 100 billion parameters. In *Proceedings of the 26th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining*. 3505–3506.

[32] Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. 2019. Megatron-lm: Training multi-billion parameter language models using model parallelism. *arXiv preprint arXiv:1909.08053* (2019).

[33] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[34] Ilya Sutskever, Oriol Vinyals, and Quoc V Le. 2014. Sequence to sequence learning with neural networks. *Advances in neural information processing systems* 27 (2014).

[35] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[36] Lewis Tunstall, Leandro Von Werra, and Thomas Wolf. 2022. *Natural language processing with transformers*. " O'Reilly Media, Inc.".

[37] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[38] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2020. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking* 29, 2 (2020), 595–608.

[39] Minjia Zhang, Zehua Hu, and Mingqin Li. 2021. DUET: A compiler-runtime subgraph scheduling approach for tensor programs on a coupled CPU-GPU architecture. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 151–161.

[40] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

[41] Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. 2019. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*. 195–208.

[42] Zhi Zhou, Xu Chen, En Li, Liekang Zeng, Ke Luo, and Junshan Zhang. 2019. Edge intelligence: Paving the last mile of artificial intelligence with edge computing. *Proc. IEEE* 107, 8 (2019), 1738–1762.