Large Language Model Partitioning for Low-Latency Inference at the Edge

Dimitrios Kafetzis[1], Ramin Khalili[2] and Iordanis Koutsopoulos[1]
[1]Athens University of Economics and Business, Department of Informatics, Athens, Greece
[2]Huawei European Research Center, Munich, Germany

***Abstract*—Large Language Models (LLMs) based on autoregressive, decoder-only Transformers generate text one token at a time, where a token represents a discrete unit of text. As each newly produced token is appended to the partial output sequence, the length grows and so does the memory and compute load, due to the expanding key-value (K/V) caches—which store intermediate representations of all previously generated tokens—in the multi-head attention (MHA) layer. As this iterative process steadily increases memory and compute demands, layer-based partitioning in resource-constrained edge environments often results in memory overload or high inference latency. To address this, aiming to reduce inference latency, we propose a resource-aware Transformer architecture partitioning algorithm, where the partitioning decision is updated at regular intervals during token generation. The approach is a myopic algorithm in the sense that it is based on instantaneously available information about device resources availability and network link bandwidths. When the algorithm is first executed, it generates a placement of blocks on devices, and in each consecutive time it is executed, it migrates these blocks among devices so that the sum of migration delay and inference delay remains low. Our approach partitions the decoder at the attention head-level, co-locating each attention head with its K/V cache and allowing dynamic migrations whenever resources become tight. By allocating different attention heads to different devices, we exploit parallel execution of attention heads and thus allow for substantial reductions in inference delays. Our experiments show that in small-scale settings (3–5 devices), the proposed method achieves within 15–20% of an exact optimal solver's latency, while in larger-scale tests it achieves notable improvements in inference speed and memory usage compared to state-of-the-art layer-based partitioning approaches.**

***Index Terms*—Large Language Models (LLMs), Transformer partitioning, Multi-head attention, Edge computing, Resource allocation, Low-latency inference.**

I. Introduction

Large Language Models (LLMs) have revolutionized a variety of natural language processing tasks, from question-answering and conversational ones to code generation. To achieve low-latency services in these applications, there is growing interest in running LLM inference at the edge, where devices such as smartphones, IoT nodes, or on-premise servers often have limited memory and compute capacity. One prerequisite for overcoming these limitations is to partition large models across multiple edge devices, harnessing collective resources instead of relying on a single node.

However, partitioning modern LLMs—often containing billions of parameters—remains highly non-trivial. One approach is to split the model along large block boundaries, e.g., entire decoder layers. Each decoder layer typically consists of multiple parallel attention head blocks and feed-forward networks, which can be assigned to different devices. This is what we call a coarse-grained partition, since it treats large decoder layers as a single unit. Moreover, such partitions are static in the sense that the assignment of blocks to devices remains fixed throughout the inference process [1]–[3]. While coarse-grained and static partitions can be effective for deep neural networks, large Transformer-based LLMs require more fine-grained and adaptive partitioning. This is due to their autoregressive decoding nature, where tokens are generated one by one. Specifically, as each new token is generated, the model stores intermediate representations—keys and values (K/V)—for all previously produced tokens in dedicated caches. These K/V caches appear in the multi-head attention (MHA) module of the Transformer and grow with every additional token in the sequence, since each new token must attend to all past tokens. As the inference task progresses and new tokens are generated, devices may become overloaded if

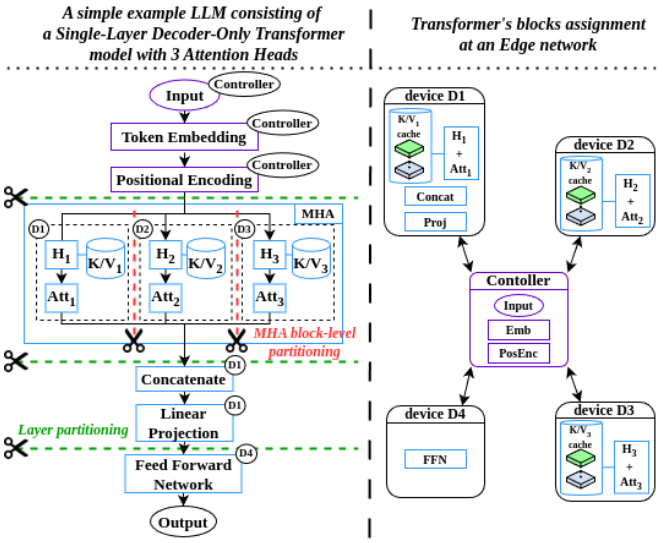

Fig. 1: Illustration of our approach, showing how attention head-level partitions (red lines) and additional cuts for feed-forward network (FFN) and linear projection (proj) blocks (green lines) enable flexible assignment of blocks across multiple edge devices. The controller node makes inference requests and handles partitioning decisions, while the attention heads (with their K/V caches), FFN, and proj blocks are allocated among devices D1, D2, D3, and D4.

ISBN 978-3-903176-73-7 © 2025 IFIP

each entire Transformer layer is treated as a single block.

In this paper, we focus on a *single-layer decoder-only Transformer architecture* (e.g., GPT [4]), meaning it includes just one instance of the core decoder sub-modules (MHA and feed-forward blocks) rather than stacking multiple such blocks. We propose a *fine-grained, attention head-level partitioning* approach that allows for more flexible use of edge resources.

The paper contributes to state of the art as follows:

- We study the problem of reducing inference delay of Transformer architectures by allocating different Transformer blocks to different devices (Figure 1).
- We develop an algorithm executed once before inference begins, and then at regular intervals during the generation process. When executed prior to token generation, the algorithm allocates different blocks of the Transformer architecture to different devices in order to reduce inference delay. When executed at regular intervals during token generation, the algorithm performs block migration among devices, taking migration delays into account.
- We demonstrate through numerical studies that our method achieves near-optimal performance in terms of inference latency reduction (within 15–20% of an exact solver) for small networks, and achieves up to 9–10 times speedup in inference latency compared to existing partitioning approaches [1], [3] in scenarios involving large numbers of generated tokens. These findings highlight the importance of attention head-level partitioning and resource management for low-latency autoregressive LLM inference at the edge.

The rest of this paper is organized as follows. Section II surveys related work on distributed Transformer inference and partitioning. Section III details our system model. Section IV presents our approach, and Section V reports simulation results. Finally, Section VI concludes the paper.

II. Related Work

The partitioning of deep learning models across resource-constrained networks has been studied extensively to address either the training or the inference stages [5]. A broad set of works aims to improve resource utilization, with emphasis on partitioning and parallelism techniques.

DNN partitioning and scheduling. Many approaches focus on splitting DNNs across devices to leverage the heterogeneous compute and memory resources of different nodes and thereby reduce computational or communication overhead. For instance, the work in [6] studies DNN partitioning for inference in resource-constrained networks, considering the joint problem of partitioning many DNNs (each split between an end-device and the Base Station) and scheduling those partitions at the Base Station to minimize inference delay. Also, SplitPlace [7] proposes AI-driven strategies for partitioning DNNs among edge devices based on resource availability and application demands. Similarly, the authors of [8] address the joint optimization of DNN partitioning and scheduling in end-edge-cloud systems to balance computational loads and reduce inference latency. Related lines of work couple partitioning with scheduling algorithms to improve resource-usage efficiency [9], using deep reinforcement learning for continuous task scheduling in edge networks (i.e., tasks arrive over time and must be allocated on the fly). Overall, these methods underscore the need for adaptive (dynamic or real-time) and task-specific (i.e., tailored to each DNN-inference job) partitioning to handle limited memory and compute resources effectively.

Pipeline and parallelism. A separate body of work focuses on pipeline- and parallelism-based solutions. Techniques such as GPipe [10] and PipeDream [11] accelerate model training via pipeline parallelism, splitting large models across accelerators and mitigating pipeline stalls. Extensions like AutoPipe [12] and PipeDream-2BW [13] further optimize memory footprints by applying pipeline-parallel training. On the other hand at the inference stage, Galaxy [3] adopts tensor and sequence parallelism for Transformer inference at the edge. Meanwhile, long-sequence parallelism techniques [14] are designed for large input sequences and manage them using data sharding (dividing input tokens among multiple devices) or pipeline sharding (distributing computation stages).

Inference for Transformers. Beyond training, methods for distributed Transformer inference have also gained traction. SplitLLM [2] studies collaborative cloud-edge inference through device selection and model partitioning, yet it focuses primarily on single-shot inference (i.e., processing an input batch in one pass) without explicitly tackling the token-by-token memory demands of autoregressive generation. EdgeShard [1] distributes large Transformers into shards (subdivisions of the model) but relies on a layer-wise design (i.e., treating each entire decoder layer as a single block). This approach may cause stalls and does not address the growing K/V caches, which store the key-value vectors for each newly generated token, enabling subsequent tokens to attend to them without recomputing at every decoding step.

In contrast, our approach partitions the Transformer architecture at the level of an attention head block, co-locates caches with attention head blocks to reduce communication load, and accommodates expanding memory footprints as tokens are generated. In Section V, we compare our approach to static pipeline-based partitioning, showing that our proposed attention head-level partitioning outperforms those methods for growing K/V caches.

III. System Model and Problem Formulation

In this section, we formalize how an inference request for a single-layer, decoder-only Transformer is processed across a network of heterogeneous devices in an edge environment.

A. Inference Request and Autoregressive Generation

We consider a single inference request for text generation that is placed as an input to a device, initiated by a text of length L_0 tokens. During the inference, the model generates a sequence of up to N tokens (units of text) one by one. At step $n \leq N$, the previously generated tokens $\{1, \ldots, n-1\}$ are fed into the decoder-only Transformer, along with the input

tokens, to produce the n-th token. We denote by $L_n = L_0 + n$ the total sequence length in tokens at step n.

a) Toy Example: Consider a simple text generation task in which the initial text is "`The cat`" (i.e., $L_0 = 2$). At step $n = 1$, the model might output "`sat`," extending the sequence to "`The cat sat`," so $L_1 = 3$. At $n = 2$, it might produce "`on`," yielding "`The cat sat on`" ($L_2 = 4$).

B. Devices, Memory, and Compute Capacities

Let $G = (\mathcal{V}, \mathcal{E})$ denote the edge network, where $\mathcal{V}$ is the set of heterogeneous devices and $\mathcal{E}$ is the set of communication links between them. Among these devices, one node acts as a controller, gathering resource state information from the rest. The inference steps are divided into intervals of size $\lambda \geq 1$, where λ is an integer and represents the number of tokens generated per interval. These intervals are indexed by integers $\tau \in \{1, \ldots, T\}$ with T representing the total number of intervals during the inference. At each interval τ, the controller collects the following information from every $j \in \mathcal{V}$:

- Available memory $M_j(\tau)$ (bytes),
- Max compute capacity W_j (FLOPs/sec),
- Available compute capacity $C_j(\tau) \leq W_j$ (FLOPs/sec), due to concurrent background processes,
- Link bandwidths $R_{j,k}(\tau)$ (bytes/sec) for communication with every $k \in \mathcal{V}$.

In this work, we assume that the memory and compute resources available at a device, as well as the link bandwidths, remain constant during an interval, or that these values represent the average predictions for the interval. The controller then decides how to allocate (or migrate) the model's blocks to the devices, as will be explained in Section IV.

C. Single-Layer Decoder-Only Transformer Blocks

Definition of Blocks and Resource Requirements: We consider a single-layer, decoder-only Transformer architecture. The layer consists of:

- $\mathcal{H} = \{1, \ldots, h\}$, the set of attention head blocks in the multi-head attention (MHA) module,
- a feed-forward network block, ffn,
- an output projection block, $proj$.

Hence, we define $\mathcal{B} = \mathcal{H} \cup \{ffn\} \cup \{proj\}$ as the set of blocks in the layer. For each attention head $i \in \mathcal{H}$, there is an associated K/V cache whose size grows token by token. Specifically, at an interval τ, each attention head i stores more keys/values than it did at the previous interval. If $i \notin \mathcal{H}$, then i is either ffn or $proj$, each having its own memory demands. For each block $i \in \mathcal{B}$, we define:

- $m_i(\tau)$ as the maximum memory requirement (bytes), and
- $b_i(\tau)$ as the maximum compute requirement (FLOPs).

Concretely, if $i \in \mathcal{H}$, $m_i(\tau)$ represents the memory footprint of the K/V cache of attention head i plus its parameters at interval τ. If $i = ffn$ or $proj$, $m_i(\tau)$ is the memory needed to store that block's parameters. Because of the model's autoregressive nature, $m_i(\tau)$ and $b_i(\tau)$ increase with τ (reflecting the fact that, as inference process progresses, more tokens are generated and added to the K/V cache of each attention head).

D. Placement (Allocation) of Blocks

We define a binary variable $x_{ij}(\tau)$ that indicates whether block i is allocated on device j at interval τ:

- If $i \in \mathcal{H}$ (i.e., an attention head),

$$x_{ij}(\tau) \;=\; \begin{cases} 1, & \text{if attention head } i \text{ and its K/V cache,} \\ & \text{are running at device } j \text{ at } \tau \\ 0, & \text{otherwise.} \end{cases}$$

- If $i \notin \mathcal{H}$ (i.e., $i = ffn$ or $proj$),

$$x_{ij}(\tau) \;=\; \begin{cases} 1, & \text{if block } i \text{ is running on device } j \text{ at } \tau, \\ 0, & \text{otherwise.} \end{cases}$$

In all cases, we require that one block is placed on one and only one device at each interval τ:

$$\sum_{j \in \mathcal{V}} x_{ij}(\tau) \;=\; 1, \quad \forall\, i \in \mathcal{B}.$$

1) Memory Capacity Constraint: At any interval τ, the total memory usage of blocks assigned to device j must not exceed $M_j(\tau)$. Thus,

$$\sum_{i \in \mathcal{B}} m_i(\tau)\, x_{ij}(\tau) \;\;\leq\;\; M_j(\tau), \quad \forall j,\, \forall \tau. \tag{1}$$

Multiple blocks can be co-located on a device j if their combined memory fits within $M_j(\tau)$.

2) Migration Cost: If a block i migrates from device j to device k at an interval τ, it incurs a migration delay:

$$D^i_{\text{mig}}\big(j \to k,\, \tau\big) \;=\; \frac{m_i(\tau - 1)}{R_{j,k}(\tau)}, \tag{2}$$

where $m_i(\tau - 1)$ is the memory footprint of block i (including its K/V cache, if i is an attention head) at previous interval, and $R_{j,k}(\tau)$ is the available bandwidth on link (j, k) at the current interval.

a) When migration occurs: A migration $j \to k$ for block i means $x_{ij}(\tau - 1) = 1$ and $x_{ik}(\tau) = 1$. We generally allow at most one migration per attention head per-interval to avoid back-and-forth overhead.

b) Remark – Why we consider a single-layer decoder: In our model, we focus on a single decoder layer to show fine-grained partitioning at the attention head-level and dynamic assignment of the partitioned blocks. This choice keeps the modeling complexity manageable for a proof-of-concept. Although we model a single-layer decoder for clarity, our attention head-level partitioning and migration approach can be applied independently to each layer in multi-layer Transformers.

E. Communication and Processing Delays

a) Communication Latency: We adopt an abstract model of the single-layer decoder:

- At interval τ, each attention head $i \in \mathcal{H}$ produces output that must be sent to $proj$ block.
- Then $proj$ transmits its output to ffn block.

If, for a particular head i, the head is on device j and $proj$ is on device k, the latency to transfer the data $W_{i\to proj}(\tau)$ is:

$$D^{i}_{j\to k}(\tau) \;=\; \frac{W_{i\to proj}(\tau)}{R_{j,k}(\tau)}, \tag{3}$$

and if $proj$ resides on k while ffn is on ℓ, we have

$$D^{proj}_{k\to \ell}(\tau) \;=\; \frac{W_{proj\to ffn}(\tau)}{R_{k,\ell}(\tau)}. \tag{4}$$

When multiple blocks on the same device j send outputs to the same device k, they must share the outgoing link (j,k). We assume that these transmissions are performed in series.

b) Processing Delay: For block i at τ, if i is on device j, we define

$$D_{ij}(\tau) \;=\; \frac{b_i(\tau)}{C_j(\tau)}, \tag{5}$$

as the time to process $b_i(\tau)$ FLOPs, given available compute $C_j(\tau) \le W_j$. However, if multiple blocks (e.g., several attention heads) share device j concurrently, they also share $C_j(\tau)$. One simple scheduling policy is to process them sequentially on j, in which case we sum their compute demands; though more sophisticated concurrency models can be used as needed.

F. Total Inference Delay at τ

Suppose we fix a placement $\mathcal{A}(\tau)$, i.e. $x_{ij}(\tau)$ for all $i \in \mathcal{B}$ at interval τ, where j indexes the devices. Then $d(\tau,i)$ denotes the device hosting block i. The total inference delay $D_{\mathrm{T}}(\tau)$ can be decomposed as follows, adapting the standard decoding pipeline (input $\to$ attention heads $\to proj \to ffn$):

$$D_{\mathrm{T}}(\tau) = \max_{i\in\mathcal{H}}\Big\{ D^{\mathrm{in}}_{\mathrm{in}\to d(\tau,i)}(\tau) + D_{i,d(\tau,i)}(\tau) + D^{i}_{d(\tau,i)\to d(\tau,proj)}(\tau)\Big\} + D^{proj}_{d(\tau,proj)\to d(\tau,ffn)}(\tau), \tag{6}$$

where:

- $D^{\mathrm{in}}_{\mathrm{in}\to d(\tau,i)}(\tau)$ is the delay of moving input tokens from their initial storage (e.g. a controller node) to the attention head's device $d(\tau,i)$;
- $D_{i,d(\tau,i)}(\tau)$ is the processing delay of attention head i on its device $d(\tau,i)$ (per (5));
- $D^{i}_{d(\tau,i)\to d(\tau,proj)}(\tau)$ is the communication delay from attention head i to $proj$ (per (3));
- $D^{proj}_{d(\tau,proj)\to d(\tau,ffn)}(\tau)$ is the delay from $proj$ to ffn (per (4)).

When multiple blocks share a device or link, there are concurrency effects:

- *Compute concurrency:* If multiple heads or other blocks run on the same device j and share compute $C_j(\tau)$, the processing times will depend on the block compute scheduling policy.
- *Link concurrency:* If multiple heads on the device j send outputs to the device k simultaneously, the transfer delays in $D^{i}_{d(\tau,i)\to d(\tau,proj)}(\tau)$ may be aggregated or scheduled.

Equation (6) retains a simplified pipeline form for clarity.

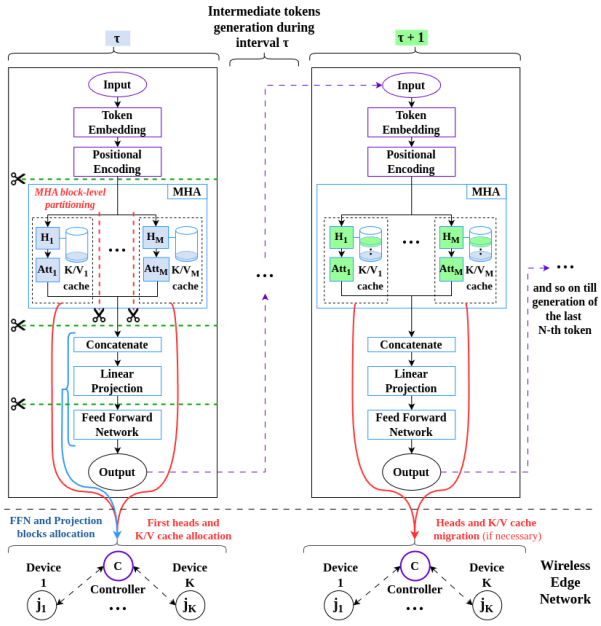

Fig. 2: A single-layer decoder-only Transformer at intervals τ and $\tau+1$, continuing autoregressively up to the N-th token generation. Red lines show attention head-level partitioning in MHA; green lines indicate projection and feed-forward partitioning. Cyan vs. green blocks highlight new tokens building on the K/V caches of the attention heads. At the bottom, a controller allocates these blocks across edge devices.

G. Decision at Each τ

We must decide a new placement $\mathcal{A}(\tau)$, i.e. a mapping $\{i \mapsto d(\tau,i)\}$ for each block $i \in \mathcal{B}$ at each interval τ. If block i was on device $d(\tau-1,i)$, but we now place it on $d(\tau,i) \neq d(\tau-1,i)$, that constitutes a migration with cost $D^{i}_{\mathrm{mig}}\big(d(\tau-1,i) \to d(\tau,i),\,\tau\big)$ per (2). We sum the delays of these migrations:

$$D^{\mathrm{total}}_{\mathrm{mig}}(\tau) \;=\; \sum_{i\in\mathcal{B}} D^{i}_{\mathrm{mig}}\big(d(\tau-1,i) \to d(\tau,i),\,\tau\big), \tag{7}$$

assuming migrations happen sequentially or that no two or more blocks migrate over the same link simultaneously.

a) Objective: At each τ, we need to find an assignment $\mathcal{A}(\tau)$ in order to minimize $D_{\mathrm{T}}(\tau) + D^{\mathrm{total}}_{\mathrm{mig}}(\tau)$, subject to the memory constraint (1), and the network's communication/processing costs. At $\tau = 1$, we initialize $\mathcal{A}(1)$ to minimize $D_{\mathrm{T}}(1)$, while for $\tau > 1$ we also include the migration cost $D^{\mathrm{total}}_{\mathrm{mig}}(\tau)$.

1) Per-Interval Assignment Policy (Myopic Algorithm): The controller node executes a centralized algorithm that decides the assignment policy (Figure 2) i.e., the block-to-device mapping $\mathcal{A}(\tau)$ at each interval τ as follows:

- The controller receives updated $\{C_j(\tau)\}$, $\{M_j(\tau)\}$, and $\{R_{j,k}(\tau)\}$ from all devices/links.
- It observes the previous assignment $\mathcal{A}(\tau-1)$.
- It computes the new assignment $\mathcal{A}(\tau)$ by solving a constrained optimization that minimizes $D_{\mathrm{T}}(\tau) + D^{\mathrm{total}}_{\mathrm{mig}}(\tau)$ under the memory constraint (1).
- It performs any required block migrations.

If we had a priori knowledge (or accurate predictions) of the capacity evolutions $\{C_j(\tau), M_j(\tau), R_{j,k}(\tau)\}$ on each τ, then we would solve for a global schedule $X = \{X(1), \ldots, X(T)\}$ where each $X(\tau)$ is a matrix $\mathcal{B} \times \mathcal{V}$ with $x_{ij}(\tau) = 1$ indicating that block i runs on device j during the interval τ. In this work, we focus on the simpler myopic scheme that takes as input the current block allocation and the current availability of resources in memory, compute and bandwidth and performs the migration that gives the best cost (migration plus inference) as perceived at the next interval. This approach is more realistic, since it is more plausible to know the instantaneous bandwidth, memory, and compute resource amounts and decide based on them on what the best next move is, rather than knowing the entire process of evolution of these resources.

In the next section, we present our proposed practical heuristic of this placement and migration procedure.

IV. Resource-Aware Algorithm for LLM Block Assignment to Devices

In this section, we present a practical heuristic for the myopic assignment procedure described above. Recall that the controller that executes the centralized heuristic algorithm at each interval τ, knows the device compute capacities $\{C_j(\tau)\}$, memory capacities $\{M_j(\tau)\}$, bandwidths $\{R_{j,k}(\tau)\}$, and the previous placement $\{x_{ij}(\tau - 1)\}$. At each τ, the controller seeks to find a block-to-device mapping $\mathcal{A}(\tau) = \{x_{ij}(\tau) : i \in \mathcal{B}, j \in \mathcal{V}\}$, to minimize the interval's total delay (inference plus migration), subject to the time-varying memory constraints (1) and the placement definition from Section III-D.

A. Algorithm Overview

At a high level, the algorithm steps are as follows:

1) Collect resource availability from devices and links: The algorithm gathers the updated device compute capacities $\{C_j(\tau)\}$, memory $\{M_j(\tau)\}$, and link bandwidths $\{R_{j,k}(\tau)\}$, along with the previous assignment $\{x_{ij}(\tau - 1)\}$.
2) Compute feasibility scores: For each block $i \in \mathcal{B}$ and device j, we calculate a scoring function $\mathcal{S}(i, j, \tau)$ to gauge the suitability of placing i in j. This step is per block, so it does not yet consider concurrency with other blocks on j. A score $\mathcal{S}(i, j, \tau) \leq 1$ simply indicates that i by itself could fit on j given j's resources.
3) Assign blocks and handle migrations: We pick the device j^* yielding the lowest feasible score for each block i. If $j^* \neq j_{\text{old}}$, we incur a migration cost $D^i_{\text{mig}}\big(j_{\text{old}} \to j^*, \tau\big)$ per (2). If no device is feasible for i alone, we attempt to resolve overload by migrating other blocks from devices with insufficient memory or compute.
4) Verify constraints or backtrack: After assigning all blocks, we check memory demands (sum of all blocks' usage on device j vs. $M_j(\tau)$) and compute demands (total load vs. $C_j(\tau)$). If any constraint is violated, we backtrack by removing a minimal set of blocks (the fewest blocks needed to resolve the violation) from the overburdened device and reassigning them. This ensures concurrency among blocks is handled collectively.
5) Return the new placement to the controller: Once we finalize $\{x_{ij}(\tau)\}$ for all blocks i, the controller applies the assignment, triggers migrations if needed, and proceeds to the next interval $\tau + 1$.

We effectively execute the algorithm at every interval τ, where the size λ of each interval τ is chosen so each interval is on the order of a few seconds, thus allowing enough time for migrations to occur.

a) Scoring Function: To decide on which device j block i should be placed at τ, we define the scoring function

$$\mathcal{S}(i, j, \tau) \;=\; \max\Big\{ \tfrac{m_i(\tau)}{M_j(\tau)},\, \tfrac{b_i(\tau)}{C_j(\tau)},\, \mathrm{CommFactor}(i, j, \tau) \Big\}.$$

This scoring function measures how each block i would use memory, impose compute load, and incur communication overhead on a candidate device j. Specifically:

- Memory/Compute feasibility: The terms $\frac{m_i(\tau)}{M_j(\tau)}$ and $\frac{b_i(\tau)}{C_j(\tau)}$ measure whether block i alone could fit into j's resource capacities. A low ratio suggests device j can handle i, assuming j's resources are not already fully consumed by other blocks.
- Communication overhead: $\mathrm{CommFactor}(i, j, \tau)$ approximates data transfer times if i must exchange information with blocks on different devices.

A device j is said to be individually feasible for block i if $\mathcal{S}(i, j, \tau) \leq 1$. However, in the presence of multiple blocks competing for device j, we rely on the final constraints check (Step 4 in the algorithm in section IV-A) to ensure that the sum of $m_i(\tau)$ across all assigned blocks does not exceed $M_j(\tau)$, and likewise that total compute can be scheduled within $C_j(\tau)$. If assigning i to j alongside other blocks pushes j's memory or compute usage beyond feasible limits, we resolve this by invoking `ResolveResourceOverload` (Section IV-B1) to attempt block migrations that free up memory or compute resources.

b) Termination Criteria: To ensure the assignment procedure at each interval τ completes in finite time, we impose a runtime safeguard with two stopping conditions: *(i)* a time limit $T_{\max}$ (i.e., a real-time upper bound on how long the controller can spend adjusting assignments), and *(ii)* an iteration bound $U = |\mathcal{B}| \times |\mathcal{V}|$, which caps repeated reassignments (migrations/backtracking) so the algorithm does not endlessly retry assignments of blocks. If either limit is exceeded, the algorithm returns INFEASIBLE. These conditions keep our myopic approach tractable for each interval.

B. Explanation of the Main Steps of Algorithm 1

Algorithm 1 runs at each interval τ and the pseudocode follows.

- Lines 1–3: We reset the migration/backtrack counters, record the start time, and collect up-to-date memory and compute capacities for each device at τ and as well as the links' bandwidth.

- Line 4: We generate a sorted list of blocks (attention heads, feed-forward, projection, etc.) based on descending memory or compute demand, ensuring higher-demand blocks are considered first.
- Lines 5–22: For each block, we select the device with the lowest $\mathcal{S}(i, j, \tau)$. We then tentatively place the block there and verify total resource usage on that device. If usage exceeds its limits, we revert the assignment and call `ResolveResourceOverload` to migrate other blocks and free capacity. Any block migration increments a counter, and exceeding its bound leads to INFEASIBLE.
- Lines 23–29: If memory or compute constraints remain violated after assignment, we invoke `BacktrackForResourceViolations` to relocate blocks causing the conflict. If too many backtracking or migration steps occur, or if we reach the time limit, we return INFEASIBLE.
- Line 30: Once all constraints are satisfied, the algorithm concludes by returning the final feasible mapping of all blocks to devices for the interval τ.

Algorithm 1 Resource-Aware algorithm for LLM block assignment at interval τ

Input: Network $G = (\mathcal{V}, \mathcal{E})$, set of blocks $\mathcal{B}$, interval index τ, previous assignment $\{x_{ij}(\tau - 1)\}$, iteration bound U, time limit $T_{\max}$

Output: Assignment $\{x_{ij}(\tau)\}$ or INFEASIBLE if no valid solution is found

1: Initialize migrationCount $\leftarrow 0$, backtrackCount $\leftarrow 0$
2: Start timer to enforce $T_{\max}$
3: `UpdateResourceUsage(`τ`)` // gather $M_j(\tau), C_j(\tau)$, and link bandwidths $\{R_{j,k}(\tau)\}$ for each device $j \in \mathcal{V}$.
4: Sort $\mathcal{B}$ into $blocksQueue$ (descending by $m_i(\tau)$ or $b_i(\tau)$)
5: **for each** block $i \in blocksQueue$ **do**
6: Compute score $\mathcal{S}(i, j, \tau)$ for all $j \in \mathcal{V}$
7: Select device $j^* \leftarrow \arg\min_j \mathcal{S}(i, j, \tau)$
8: **if** $\mathcal{S}(i, j^*, \tau) \leq 1$ **then**
9: $x_{ij^*}(\tau) \leftarrow 1$ // tentative assignment
10: **if** $\sum_{i'} m_{i'}(\tau)\, x_{i'j^*}(\tau) > M_{j^*}(\tau)$ **or** $\sum_{i'} b_{i'}(\tau)\, x_{i'j^*}(\tau) > C_{j^*}(\tau)$
11: $x_{ij^*}(\tau) \leftarrow 0$; // undo assignment
12: `ResolveResourceOverload`(i, τ)
13: migrationCount $\leftarrow$ migrationCount $+ 1$
14: **if** migrationCount $> U$ **then return** INFEASIBLE
15: **else if** i moved from $j_{\text{old}} \neq j^*$
16: migrationCount $\leftarrow$ migrationCount $+ 1$
17: **if** migrationCount $> U$ **then return** INFEASIBLE
18: **else**
19: `ResolveResourceOverload`(i, τ)
20: migrationCount $\leftarrow$ migrationCount $+ 1$
21: **if** migrationCount $> U$ **then return** INFEASIBLE
22: **end if**
23: **if** elapsed time $> T_{\max}$ **then return** INFEASIBLE
24: **end for**
25: **if not** `allConstraintsSatisfied`$\big(\{x_{ij}(\tau)\}\big)$ **then**
26: `BacktrackForResourceViolations`(τ)
27: backtrackCount $\leftarrow$ backtrackCount $+ 1$
28: **if** backtrackCount $> U$ **then return** INFEASIBLE
29: **end if**
30: **return** Full assignment matrix $\{x_{ij}(\tau)\}$

1) Resolving Resource Overload: If a block i cannot be placed on any device without exceeding memory/compute constraints at τ, `ResolveResourceOverload` tries to migrate other blocks. For instance, to move a block from device j to k, we check that the sum of memory already on k plus $m_i(\tau)$ remains within $M_k(\tau)$. Similar checks apply for compute. If no such migration fixes the overload, we escalate to `BacktrackForResourceViolations`.

2) Backtrack for Resource Violations: When constraints remain violated after attempts to resolve overload, we use `BacktrackForResourceViolations` to reassign a minimal set of blocks that cause the violation. If it still fails, we return INFEASIBLE. This approach remains consistent with the myopic policy from Section III-G, handling each token-generation interval in a bounded-time manner.

Overall, the worst-case time complexity of Algorithm 1 at each interval τ is $\mathcal{O}(|\mathcal{B}|^2|\mathcal{V}|)$. In the most demanding scenarios, each block may require checking all devices and triggering multiple reassignments to resolve resource overloads. This complexity remains practical for small- and medium-scale deployments, but may require approximation heuristics in large-scale settings.

By iterating these steps, we obtain a mapping $\mathcal{A}(\tau)$ that aims to minimize inference and migration delays at each interval τ. In Section V, we demonstrate the evaluation of the performance of Algorithm 1.

V. EVALUATION

We evaluate our resource-aware algorithm in two distinct settings to assess both correctness (optimality gap) and scalability:

- Small-scale (3–5 devices, $N = 4$ tokens): We use an exhaustive (exact) solver to find the optimal assignment. This is feasible only for small-scale setups. We measure our heuristic's deviation from the global optimum that is found with exhaustive enumeration of the solutions, and compare against simpler baselines—Greedy, Round-Robin, Static, and Dynamic—as described below.
- Medium-scale (25 devices, up to $N = 1000$ tokens): We scale up the number of devices, examining whether our approach outperforms other methods under more realistic edge settings with higher heterogeneity. We benchmark our resource-aware approach against state-of-the-art partitioning frameworks (EdgeShard [1], Galaxy [3]) for large-scale Transformer inference across multiple devices.

We execute our resource-aware algorithm at fixed intervals of size $\lambda = 1$, meaning that one token is generated during each interval τ. In other words, the algorithm updates the block-to-device assignment $\mathcal{A}(\tau)$ once per token. This setup represents

a worst-case scenario in terms of migration overhead, as it triggers the controller to re-evaluate and potentially migrate blocks at every single decoding step. As a result, it creates the highest possible frequency of migration and placement decisions, which stresses the system and highlights the efficiency of our method under tight constraints.

A. Baseline Methods

We compare our approach to the following baselines, which illustrate different ways of Transformer architecture partitioning:

- Greedy: Sort blocks in descending order of resource demand and place them on the first feasible device without re-checking feasibility in subsequent steps.
- Round-Robin: Assign blocks sequentially to devices in a cyclic order, ignoring different resource requirements at different stages of the token generation process.
- Static: Do one initial assignment for all blocks and never migrate them during token generation.
- Dynamic: As in Resource-Aware algorithm, re-checks assignments at each step but treat each layer as one block.
- EdgeShard [1]: Assigns entire Transformer layers to devices. This static, layer-based partitioning does not adapt to resource changes or K/V cache growth.
- Galaxy [3]: Partitions the Transformer into contiguous layer shards for pipeline parallelism and splits each shard's large matrix multiplications across multiple devices for tensor parallelism.

B. Evaluation Setup

We built a custom Python simulator[1] in a discrete-event fashion to model each token generation step. A central controller gathers devices' memory/compute availability and links' bandwidth at every interval τ and runs our resource-aware algorithm or a baseline algorithm. The simulator's modular design supports custom device topologies, concurrency models, or partitioning policies, enabling easy adaptation to new distributed inference experiments.

a) Transformer Configurations: We focus on a single-layer decoder with h attention heads and an embedding dimension D that is the size of each token's representation. This parameter strongly influences both memory and compute usage. For a *Large* LLM model setup ($h = 32, D = 2048$), we approximate GPT-2/LLaMA scales. The initial input text length is $L_0 = 64$.

b) Device Capabilities: We sample each device's memory availability $M_j(\tau)$ (GB) and compute capacity $C_j(\tau)$ (GFLOPS) from log-normal distributions for heterogeneity (e.g., $M_j(\tau) \in [2, 8]$ GB, $C_j(\tau) \in [5, 50]$ GFLOPS) [16]. For the network, we assign each link (j, k) a bandwidth randomly drawn from $[1, 10]$ Gbps, reflecting diverse edge conditions and assuming full connectivity between devices. Table I summarizes the memory and compute usage calculation formulas per-token generation step for the considered Transformer architecture blocks, derived from the analysis in [17]; where in these formulas, b is the number of bytes for each model parameter (i.e., the numerical values learned during training that define how inputs are processed), and $d = D/h$.

Block	**Memory** ($m_i(\tau)$)	**Compute** ($b_i(\tau)$)
Attn. Head i	$3L_\tau\, d\, b + 3D\, d\, b$	$3L_\tau\, D\, d + L_\tau^2\, d$
K/V cache	$m_i^{cache}(\tau) = \tau\, D\, b$	N/A
Projection	$m_{proj}(\tau) = L_\tau\, D\, b$	$b_{proj}(\tau) = L_\tau\, D^2$
FFN	$m_{ffn}(\tau) = 4\, L_\tau\, D\, b$	$b_{ffn}(\tau) = 8\, L_\tau\, D^2$

TABLE I: Resource usage formulas assuming one token is generated per interval τ i.e., assuming that one token is generated per interval, so that $n = \tau$

c) Metrics: We consider two metrics in our evaluation, inference latency and memory usage. *Inference Latency* is total elapsed time to generate N output tokens, encompassing computation, communication, and any migration delay. *Memory Usage* can be measured as either the sum across all devices or the maximum usage on a single device, illustrating how efficiently our approach handles expanding K/V caches.

C. Small-Scale Scenario Results

Here, an *exact* search is possible and practical so as to find the optimal assignment. The ratio of each method's total latency to that of the optimal solution shows that our Resource-Aware approach remains within 15–20% of optimal, while Greedy and other baselines may lag behind by 40–60%. This underscores the impact of attention head-level partitioning for short decoding sequences for a small number of devices.

D. Medium-Scale Scenario Results

Next, we simulate a network of 25 devices (2–8 GB memory, 5–50 GFLOPS compute) with N up to 1000 tokens. We also inject background tasks to emulate fluctuating compute load. We compare Resource-Aware with the EdgeShard and Galaxy frameworks.

a) Inference Latency vs. Token Generation Step n: Figure 3 shows that while all methods see rising latency as more tokens accumulate, EdgeShard eventually surpasses 1000 seconds at the last generated token, and Galaxy slows to about 400–600 seconds. In contrast, our fine-grained, attention head-level partitioning dynamically reallocates attention heads to prevent severe overload, keeping latency under 200 seconds at the last generated token.

b) Memory Usage vs. Token Generation Step n: As shown in Figure 4, EdgeShard and Galaxy exceed 7 GB by $n = 100$, while Resource-Aware remains near 6 GB. The disparity grows further for larger n, partly due to the inflexibility of layer-level partitioning under expanding K/V caches.

c) Scalability with Increasing Number of Devices: Although allocating attention heads among more devices can accelerate inference by spreading the workload, larger networks impose additional coordination and scheduling overhead. As the number of devices grows, the complexity of finding a good placement or migration increases.

[1]The full codebase of our implementation is publicly available in [15]

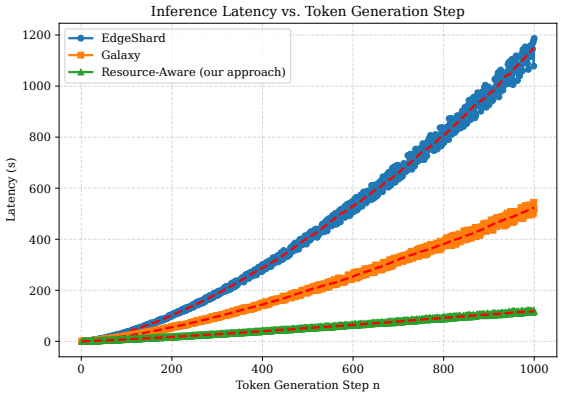

Fig. 3: Inference latency vs. generated token step n on 25 devices. Our approach (Resource-Aware) avoids steep growth.

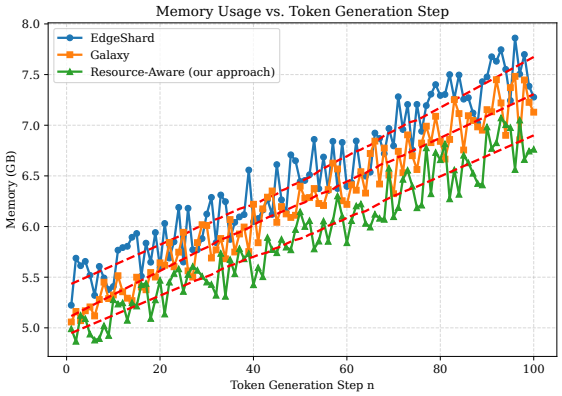

Fig. 4: Total memory usage vs. generated token step n in the 25-device setup. Our resource-aware approach mitigates memory growth more effectively.

E. Summary of Results

Overall, in the small-scale scenario, our method remains within 15–20% of the optimal solver's latency, outperforming simpler heuristics such as the Greedy and Round-Robin baselines by 40–60%. In the medium-scale setting, our approach scales well to 25 devices and $N = 1000$ tokens, resulting in latency and memory overhead considerably better than EdgeShard or Galaxy (up to 9–10 times speedup). This highlights the advantage of *attention head-level* partitioning plus *dynamic* partitioned blocks assignment for autoregressive LLM inference.

VI. Conclusion and Future Work

We introduced a resource-aware approach for partitioning decoder-only Transformers under the tight memory, compute, and link constraints. Our key contribution is to treat each attention head block along with its associated key/value cache as a distinct block that can be allocated to a certain device. The ability to execute attention head blocks in parallel across devices is the central strength of our approach, significantly reducing inference delay. By explicitly modeling how K/V caches expand at each decoding step, we can dynamically reassign these fine-grained blocks across devices to balance workloads.

For future work, we plan to extend and validate our approach on multi-layer decoder-only Transformers. We also aim to incorporate limited foresight in the decision-making process i.e., to predict resource availability ahead of time and make decisions based on these predictions. We also aim to deploy real-world testbeds that add factors such as energy constraints or inference request load forecasts, in a real environment.

References

[1] M. Zhang, J. Cao, X. Shen, and Z. Cui, "Edgeshard: Efficient llm inference via collaborative edge computing," *arXiv preprint arXiv:2405.14371*, 2024.

[2] A. Mudvari, Y. Jiang, and L. Tassiulas, "Splitllm: Collaborative inference of llms for model placement and throughput optimization," *arXiv preprint arXiv:2410.10759*, 2024.

[3] S. Ye, J. Du, L. Zeng, W. Ou, X. Chu, Y. Lu, and X. Chen, "Galaxy: A resource-efficient collaborative edge ai system for in-situ transformer inference," in *IEEE INFOCOM 2024 - IEEE Conference on Computer Communications*, pp. 1001–1010, 2024.

[4] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, *et al.*, "Language models are few-shot learners," *Advances in Neural Information Processing Systems*, vol. 33, pp. 1877–1901, 2020.

[5] One6G, "6g technology overview," 2024.

[6] D. Kafetzis and I. Koutsopoulos, "Dnn partitioning and inference task offloading in 6g resource-constrained networks," in *2024 Joint European Conference on Networks and Communications 6G Summit (EuCNC/6G Summit)*, pp. 753–758, 2024.

[7] S. Tuli, G. Casale, and N. R. Jennings, "Splitplace: Ai augmented splitting and placement of large-scale neural networks in mobile edge environments," *IEEE Transactions on Mobile Computing*, vol. 22, p. 5539–5554, Sept. 2023.

[8] Z. Zhang, Q. Li, L. Lu, D. Guo, and Y. Zhang, "Joint optimization of the partition and scheduling of dnn tasks in computing and network convergence," *IEEE Networking Letters*, vol. 5, no. 2, pp. 130–134, 2023.

[9] S. Yuan, Z. Zhang, Q. Li, W. Li, and Y. Zhang, "Joint optimization of dnn partition and continuous task scheduling for digital twin-aided mec network with deep reinforcement learning," *IEEE Access*, vol. 11, pp. 27099–27110, 2023.

[10] Y. Huang, Y. Cheng, A. Bapna, O. Firat, M. X. Chen, D. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, and Z. Chen, *GPipe: efficient training of giant neural networks using pipeline parallelism*. Red Hook, NY, USA: Curran Associates Inc., 2019.

[11] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, and M. Zaharia, "Pipedream: generalized pipeline parallelism for dnn training," in *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, SOSP '19, (New York, NY, USA), p. 1–15, Association for Computing Machinery, 2019.

[12] W. Liu, Z. Lai, S. Li, Y. Duan, K. Ge, and D. Li, "Autopipe: A fast pipeline parallelism approach with balanced partitioning and micro-batch slicing," in *2022 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 301–312, 2022.

[13] D. Narayanan, A. Phanishayee, K. Shi, X. Chen, and M. Zaharia, "Memory-efficient pipeline-parallel dnn training," in *International Conference on Machine Learning*, pp. 7937–7947, PMLR, 2021.

[14] H. Zheng, P. Liang, Y. Tang, Y. Shi, L. Qiao, and D. Li, "3d parallelism for transformers via integer programming," in *ICASSP 2024 - 2024 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pp. 6440–6444, 2024.

[15] D. Kafetzis, "Distributed Transformer Inference Simulator." https://github.com/Dimitrios-Kafetzis/DistributedTransformer InferenceSimulator, 2025.

[16] C. Reiss, J. Wilkes, and J. L. Hellerstein, "Google cluster-usage traces: format+ schema," Tech. Rep. ISTC-CC-TR-12-101, Google Inc., 2012.

[17] A. Vaswani, N. M. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, "Attention is all you need," in *Neural Information Processing Systems*, 2017.