EPipe: Pipeline Inference Framework with High-quality Offline Parallelism Planning for Heterogeneous Edge Devices

Abstract

Pipeline parallelism is essential for edge computing as it effectively consolidates the limited resources of edge devices, enabling the deployment of large Deep Neural Network (DNN) models and accelerating inference processes without compromising the performance of models. Accurate computation and communication latency estimation on heterogeneous edge devices is essential for searching for a superior parallelism plan. However, existing heterogeneous pipeline inference approaches either incur substantial resource wastage during online parallelism planning, as they utilize profiling strategies that occupy physical devices; or rely on cost models with inadequate representational capabilities, leading to inaccurate predictions, thereby harming the result of pipeline planning. This paper proposes EPipe, a novel pipeline inference framework that supports high-quality offline planning in heterogeneous edge environments. EPipe integrates two core components: the Task-Device Co-analyzer (TDC) and the Multi-pipeline Parallelism Planner (MPP). TDC utilizes an undirected connected graph to depict the compatibility of DNNs across device groups and precisely estimates inference and communication latencies through fine-grained modeling. Based on TDC, MPP utilizes a dynamic programming-based genetic algorithm to explore multi-pipeline solutions, extending beyond traditional single-pipeline methods. A comprehensive experimental evaluation on an edge testbed confirms the effectiveness of EPipe, demonstrating significant speedups in inference tasks for both task streams and single tasks.

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

Keywords: edge computing, pipeline parallelism, offline parallelism planning, latency modeling

1 Introduction

Owing to the inherent advantages of edge computing environments in terms of latency, privacy, and always-on availability, Deep Neural Network (DNN) tasks are progressively shifting to edge devices. However, edge devices typically possess limited memory capacity and computational resources, which are insufficient to accommodate the increasingly large model sizes or to ensure efficient task processing. To this end, model compression techniques such as pruning [14], quantization [25], and knowledge distillation [26] have been developed to reduce model size and floating-point operations without significantly compromising model accuracy. Unlike these model-level optimizations, model parallelism techniques divide large DNNs into shards for parallel processing across multiple devices without sacrificing model performance. This system-level optimization consolidates the fragmented memory and computational resources across edge devices, supports the deployment of larger models, and yields acceleration in inference processes.

Although model parallelism techniques, such as tensor parallelism and pipeline parallelism, are well-established in cloud computing environments, applying them to edge environments is nontrivial. Tensor parallelism [19] facilitates parallel processing by dividing the workload of single operators (e.g., matrix operations) across multiple processing units, which requires frequent all-reduce communication. However, edge devices typically lack high-speed dedicated interconnects, such as NVLink or InfiniBand, which are prevalent in cloud environments. These intensive collective communications introduce considerable synchronization overhead, severely affecting parallel efficiency [30]. In contrast, pipeline parallelism divides the model into multiple sequential stages. Each stage, managed by a dedicated worker, forwards its output only to the next worker, which avoids collective communication and synchronization between all workers and facilitates the overlapping of computation and communication to mask communication latency. Nevertheless, studies of pipeline parallelism on the cloud have primarily focused on homogeneous accelerators [8, 30] or have considered only limited heterogeneity, such as heterogeneous communication topologies with homogeneous GPUs [15, 16] or heterogeneous clusters with homogeneous networks [17]. Such studies fail to meet the specific demands of edge computing, which is characterized by the coexistence of multiple heterogeneous devices interconnected through diverse configurations.

Therefore, some studies [7, 18, 27, 29] have focused on pipeline parallelism strategies that incorporate considerations of computational and communication heterogeneity in edge scenarios. At this point, accurately estimating computation and communication latencies in highly heterogeneous environments is crucial, as these calculations directly impact the outcomes of parallelism planning. Most studies [12, 17] employ the profiling strategy, wherein tasks are decomposed into smaller units and executed on various physical devices to measure corresponding latencies. The profiling process is extremely time-consuming, necessitating that the parallel planning phase be executed online. This leads to significant waste of physical resources and inefficiencies in the planning procedure. Several studies [18, 27] have introduced cost models instead of profiling. However, their cost models typically consider only the statistical or configuration information of DNNs and devices, such as parametric quantities and communication bandwidths, while our empirical studies reveal that predictions made by these cost models deviate significantly from actual results. This deviation will result in planning a load-imbalanced pipeline parallelism solution, which directly impairs the efficiency of pipeline runtime.

Addressing the challenges where existing methods are either inefficient in planning procedures or in runtime of pipelines, this paper introduces **EPipe**, a novel pipeline inference framework. EPipe supports high-quality offline parallelism planning in advance, thereby enabling seamless task switching and superior pipeline parallelism in DNN task streams. Concretely, EPipe integrates two core components, the Task-Device Co-analyzer (TDC) and the Multi-pipeline Parallelism Planner (MPP), to support high-quality offline parallelism planning in heterogeneous edge environments. Firstly, TDC abstracts the compatibility of DNNs across device groups into an undirected connected graph (UCG), where nodes can depict the inference latency of individual devices for arbitrary shards of the DNN, and edges can reflect the communication latency between

devices. A comprehensive graph representation can be derived by injecting features of tasks and devices, which serve as the foundation for effective offline parallelism planning. Secondly, MPP is introduced to plan an efficient multi-pipeline parallelism solution. Existing strategies typically offer one single-pipeline parallelism solution for each task, which may not fully utilize all available devices due to resource constraints. Multi-pipeline parallelism solutions, featuring multiple heterogeneous and independent pipelines that operate simultaneously within a device group, potentially can improve both inference efficiency and device utilization. Consequently, MPP broadens the search space to multi-pipeline parallelism solutions, which are generated by an efficient dynamic programming (DP) based genetic algorithm. Finally, EPipe provides an end-to-end automated parallelism implementation for given tasks and devices, completing the entire parallelism process automatically.

In summary, this paper makes the following contributions:

- This paper achieves a novel pipeline inference framework named EPipe, which supports high-quality offline parallelism planning for heterogeneous edge devices.
- This paper proposes TDC, abstracting the compatibility of DNNs across device groups into a UCG and utilizing fine-grained modeling to obtain accurate estimates of latencies.
- This paper designs MPP, a DP-based genetic algorithm, to expand the solution space to multi-pipeline parallelism beyond traditional single-pipeline parallelism.
- A comprehensive experimental evaluation on an edge testbed demonstrates that EPipe significantly enhances the inference task throughput for both single tasks and task streams.

2 Background and Motivation

This section demonstrates that profiling-based online parallelism planning leads to considerable resource wastage and reveals the inadequacy of existing cost models in accurately estimating the computation and communication latency of pipeline runtime.

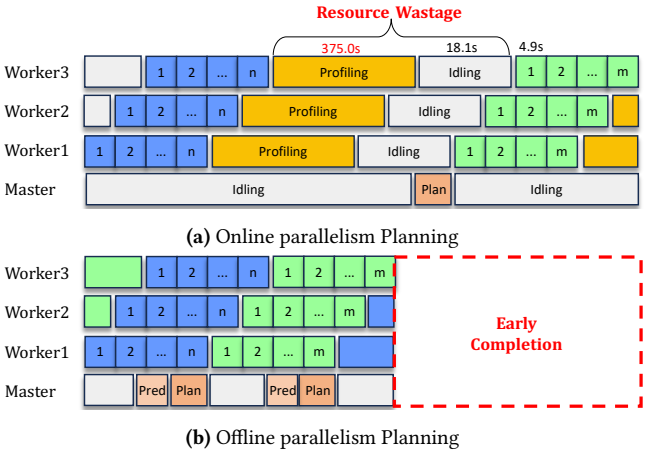
2.1 Resource Wastage in Online Parallelism Planning

Figure 1 illustrates the differences between online and offline parallelism planning[1]. Specifically, it demonstrates the inference of two distinct tasks using a 3-stage pipeline and the switching procedure between them. This scenario reflects a common requirement in real-world applications, where DNN inference tasks are subject to frequent changes.

Online parallelism planning incurs considerable resource wastage. Initially, the DNN model of the actual task must be profiled on each heterogeneous device. Subsequently, performance profiling results from all devices are collected. Parallelism planning can commence only after all profiling results are gathered, as it relies on these results to estimate the runtime overhead across various pipeline configurations. Moreover, all devices can only initiate the new pipeline after the planning is completed.

Following the latest profiling-based study [7], we execute a ViT Large [4] 3-stage pipeline with 307 million parameters on edge devices, yielding more intuitive outcomes. On the Jetson AGX Xavier with maximum performance mode, the profiling of individual layers

[1]The pipeline order may be altered after planning completion, but this does not impact the difference between online and offline parallelism planning procedures. Consequently, this situation is not depicted in the figure.



(a) Online parallelism Planning

(b) Offline parallelism Planning

Figure 1. Online vs. Offline parallelism Planning

consumes 375.0s, with an idle time of 18.1s from profiling completion until the device initiates inference. However, with a batch size of 64, the actual pipeline stage time is merely 4.9 seconds. This suggests that the resources wasted during online parallelism planning are sufficient to perform inference on approximately 5134 images!

In contrast, in offline parallelism planning, latencies are estimated not through actual profiling but through prediction. The planning procedure can be executed in advance, facilitating seamless switching between different pipelines. However, the accuracy of prediction directly impacts the efficiency of pipeline runtime. We will further discuss this subsequently.

2.2 Inaccurate Estimation by Existing Cost Models

Predicting runtime latencies is the foundation for offline parallelism planning; however, achieving precise predictions is non-trivial. Existing works have utilized the following cost models:

Computation latency: FLOPs-based methods [27, 28] rely solely on the FLOPs of DNNs for linearly latency prediction; Decision tree-based methods [18] use basic task statistics and device configuration to build a decision tree for prediction.

Communication latency: Bandwidth-based methods [7, 15, 18, 27] divide communication volume by bandwidth.

We deploy 500 different shards from real DNNs [11] to Jetson AGX Xavier and Jetson Xavier NX, using the aforementioned cost models to predict both the computation latencies on the devices and the communication latencies for intermediate outputs between

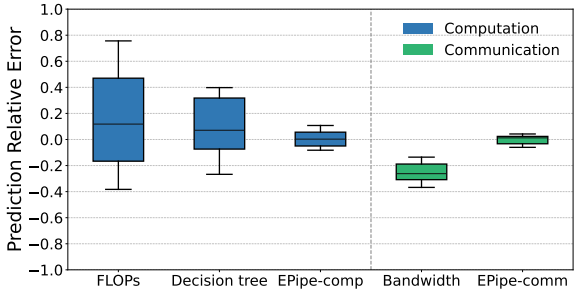


Figure 2. Existing Cost Models Evaluation

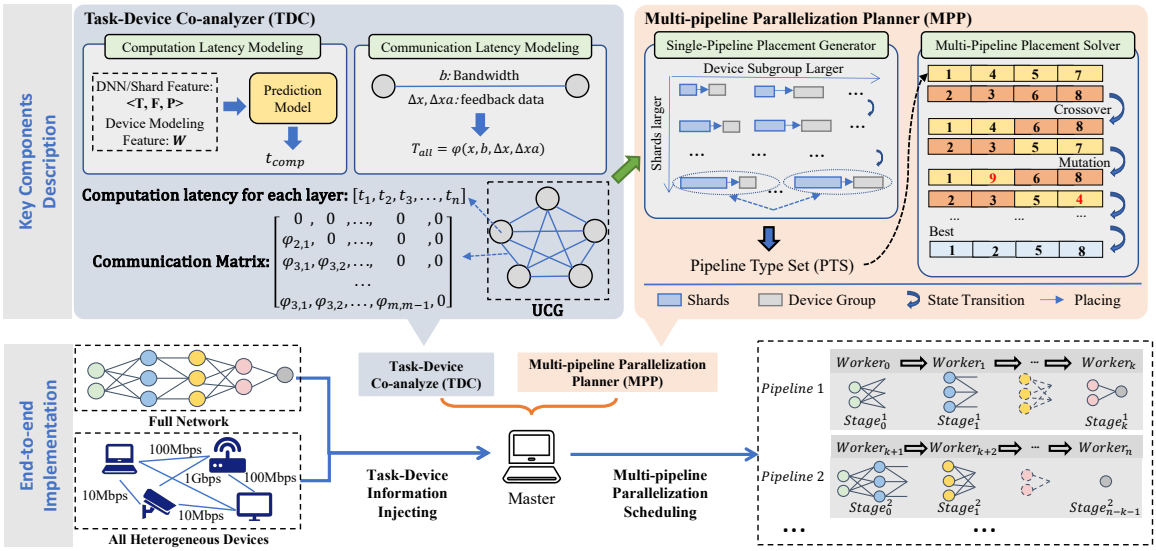


Figure 3. Overview of EPipe. The upper portion depicts two key components, where TDC ensures precise latency estimation and MPP plans the multi-pipeline parallel optimization scheme offline. The lower half of the figure highlights the end-to-end implementation of EPipe.

them. Their relative errors of latency prediction errors are illustrated in Figure 2. It is evident that there is a substantial discrepancy between the predicted and actual latency times, with some predictions deviating by more than 20%. Such errors will directly harm the results of parallelism planning.

We posit that existing cost models are inadequate for accurately predicting latency due to the diversity of DNN structures, device heterogeneity, and communication complexities. For computation latency, key sources of prediction error include: 1. Statistical metrics like FLOPS or parameter counts neither adequately distinguish between the structures of DNN shards nor provide detailed descriptions of individual layers. 2. Relying solely on hardware configurations overlooks some important factors impacting the actual computational capabilities of heterogeneous devices, such as hardware architectures or operator libraries. 3. Traditional machine learning algorithms, such as decision trees, are inadequate for characterizing the complex inference behaviors of arbitrary shards on heterogeneous devices. Regarding communication latency, transmission latencies derived solely from bandwidth calculations represent only a fraction of the total latency, overlooking other factors such as data processing latency and queuing delay.

In summary, previous profiling-based online planning methods sacrifice planning efficiency, while prediction-based methods compromise the efficiency of pipeline runtime. Building upon these insights, we introduce EPipe, an offline planning pipeline inference framework grounded in precise latency estimates.

3 Method

In this section, we formally introduce the proposed EPipe framework, as illustrated in Figure 3. Within the EPipe framework, a master node leverages information about a given task and device group to orchestrate multi-pipeline parallelism for inference and to schedule shards to the corresponding workers. Specifically, EPipe employs its core components, TDC and MPP, to plan multi-pipeline parallelism solutions offline. TDC employs a novel fine-grained method for modeling computation and communication latencies to accurately predict pipeline runtime latencies. Based on these predictions, a UCG is constructed to represent the compatibility of DNNs across device groups, thereby laying the groundwork for deriving a multi-pipeline parallelism strategy using MPP. Subsequently, MPP conducts multi-pipeline parallel planning in two phases. Initially, to simplify the solution space, MPP uses dynamic programming to generate a pipeline type set that encompasses all potentially optimal single-pipeline solutions. Finally, a genetic algorithm is utilized for deriving a high-quality multi-pipeline parallel strategy.

3.1 Task-Device Co-analyzer (TDC)

EPipe introduces TDC to estimate pipelines' runtime latencies accurately. Performance of the same shard can vary significantly across heterogeneous devices due to diverse hardware architectures and characteristics. Consequently, TDC employs UCG to represent the suitability of DNNs on device groups. Concretely, each node corresponds to an individual worker and includes its computation latency for each layer, while the edges form a communication matrix that contains the communication latency functions between all workers when transmitting intermediate outputs.

3.1.1 Computation Latency Modeling. Computation latency modeling is designed to complete the UCG node information. As illustrated in Figure 4, TDC employs a **transformer-based prediction model**, which utilizes **fine-grained task features** and **data-driven device characteristics embeddings** as inputs, with the computation latency of each layer on the device as the output.

Fine-grained task features: In contrast to previous methods that solely rely on statistical metrics, TDC incorporates fine-grained features to characterize DNN tasks more adequately. It employs multiple embedding vectors, each representing an individual layer and consisting of three components: layer type information, layer-specific features, and positional encoding, as detailed in Table 1. Initially, the layer type information is introduced, because variations in layer types can significantly influence their computation

Table 1. DNN Embedding Information.

	Feature Embeddings
Type	AvgPool, MaxPool, Norm, Conv, Linear
Feature	Input Size, Input Channel, Output Size, Output Channel, Parameters, MACs, Kernel Size
Position	Depth, Index, Parent Depth, Parent Index

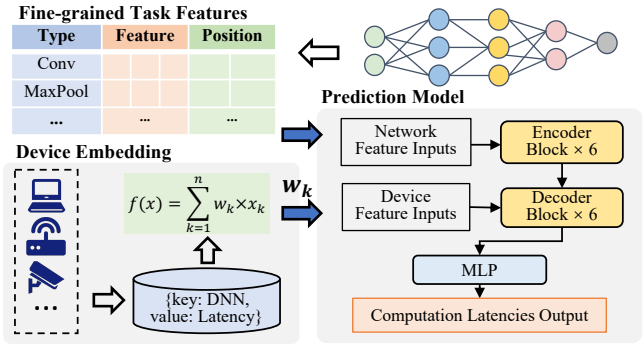


Figure 4. Computation Latency Modeling

latencies, even when the parametric quantities remain identical [13]. Subsequently, once the layer type is identified, the computation latency of an individual layer is influenced by features such as the computation amount, which can be sourced from third-party libraries such as *torchinfo*. Moreover, the data dependencies among layers in DNNs are becoming increasingly complex [24], and both the multi-branch network structure and intricate data dependencies can significantly impact the inference performance [1]. Consequently, TDC integrates positional encodings at each layer, effectively capturing sequential attributes specific to each layer, such as the number of branches (Depth) and the layer's relative position on its branch (Index). Additionally, information about the parent node is included to introduce its data dependencies.

Data-driven device characteristics embeddings: TDC implements a data-driven approach to embedding device characteristics rather than relying solely on hardware configurations. Initially, TDC constructs a dataset by testing the inference latencies of various real-world DNNs on the given device, with each sample being a key-value pair formatted as `key: DNN statistical information, value: latency`. Subsequently, a linear regression model is employed to capture the prediction relationship between these key-value pairs, aiming to condense the performance discrepancies of different DNNs on devices into a unified representation through the parameters (W_k) of the linear model. Driven by its own dataset, each device category possesses a distinct set of W_k, serving as device-specific feature embeddings. The embeddings standardize the performance metrics of heterogeneous devices within a unified, continuous hyperspace. It addresses potential issues such as inconsistency, incompleteness, and discontinuity in device configuration information, thereby facilitating comprehensive end-to-end performance characterization of heterogeneous devices.

Transformer-based prediction model: Considering the well-established representation capability of the original transformer [21], it is utilized as the backbone network of the computation latency prediction model in TDC. These fine-grained task features serve as tokens and are inputted to the encoder. Device characteristics embeddings are input into the decoder, which engages in cross-attention mechanisms with the encoder's outputs. The decoder subsequently connects to a multi-layer perceptron (MLP) layer, which generates final latency predictions.

3.1.2 Communication Latency Modeling. Communication latency modeling is designed to complete the edge information of UCG. As depicted in Figure 5, to facilitate the overlap of computation and communication, each node is equipped with a minimum of two asynchronous threads: a computation thread and a communication thread. The communication thread employs separate input and output queues to efficiently isolate its tasks from those managed by the computation thread.

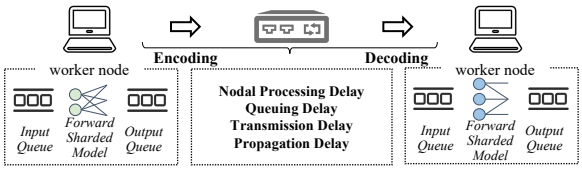


Figure 5. Communication Latency Modeling Method

Thus, communication latency is defined as the duration from the start of data transmission in the output queue to the completion of reception in the input queue. It includes data encoding and decoding delay(T_{ende}), nodal processing delay (T_n), queuing delay(T_q), transmission delay(T_t), and propagation delay(T_p), which can be formalized as:

$$T_{all} = T_{ende} + T_n + T_q + T_t + T_p \tag{1}$$

To ensure the accuracy of the communication latency estimation, TDC analyses the factors affecting each communication phase between workers separately.

- T_{ende}: To facilitate the transmission of tensors between different hosts, it is necessary to serialize tensors into a byte stream and subsequently deserialize it after receipt. Given that the efficiency of encoding and decoding operations on a device's CPU is relatively constant, the delay associated with these processes is linearly correlated with the data size (x_{plan}). Thus, this delay can be fitted by introducing a linear function, i.e., $T_{ende} = w_1 \times x_{plan} + \varepsilon_1$.
- T_n and T_p: In edge LAN environments characterized by minimal bit errors and close geographic proximity, both the nodal processing and propagation delays are stable and substantially lower than the total communication latency. A constant bias can be introduced to estimate them, i.e., $T_n + T_p = \varepsilon_2$.
- T_t: Transmission delay is determined by the ratio of the data volume to the bandwidth. It can be represented as $T_t = \frac{x_{plan}}{b}$.
- T_q: Queuing delay is influenced by the actual network congestion, which increases significantly when network traffic is high, or the device's processing capacity is relatively inadequate. Next, we launch into a detailed discussion.

The runtime queuing delay of a pipeline that is being planned offline can be approximated by measuring the current queuing delay. However, due to potential significant differences in the structure of the DNN employed in the current task compared to the DNN being planned, the network traffic within the current pipeline may differ substantially from that expected in the pipeline being planned. Hence, TDC employs a model that considers task differences to enable precise estimation of queuing delays. Specifically,

this is achieved by incorporating features that capture variations in pipeline task traffic. These features encompass the difference in data volume transferred between workers, $\Delta x = x_{now} - x_{plan}$, and the difference in the average data volume transferred across all workers, $\Delta x_a = xa_{now} - xa_{plan}$. These data volumes can be computed directly once given the DNN structure and pipelined placement plan, and if no pipelines are currently running in the device group, x_{now} and xa_{now} should be zero. Integrating these pipeline traffic difference features and the current queuing delay between workers ($\widetilde{T_q}$), we can define T_q as:

$$T_q = w_2 \times \widetilde{T_q} + w_3 \times \Delta x + w_4 \times \Delta xa + \varepsilon_3 \tag{2}$$

where w_* and ε_* are the parameters to be fitted and are related to the communication efficiency between workers.

Combining the above delays, the planning communication latency (T_{all}) in planning pipeline tasks can be estimated as follows:

$$\begin{aligned} T_{all} = \varphi(x_{plan}, b, \Delta x, \Delta xa; w_*, \varepsilon_*) = w_1 \times x_{plan} \\ + \frac{x_{plan}}{b} + w_2 \times \widetilde{T_q} + w_3 \times \Delta x + w_4 \times \Delta xa + \varepsilon \end{aligned} \tag{3}$$

where $\varepsilon = \varepsilon_1 + \varepsilon_2 + \varepsilon_3$, $\varphi(*)$ denotes the total latency when transferring the specified amount of data x_{plan} between two workers.

At the initialization of device groups or when changes in the communication topology are made, TDC necessitates a warm-up phase for calibrating the fitting parameters in $\varphi(*; w_*, \varepsilon_*)$. This calibration involves constructing a sample set to fit $\varphi(*)$ by collecting a limited dataset of related network traffic data $(x_{plan}, b, \Delta x, \Delta xa)$ and end-to-end communication latencies between worker pairs, i.e., {key: $(x_{plan}, b, \Delta x, \Delta xa)$, value: latency}. These data pairs enable the acquisition of relatively accurate parameters. Subsequently, continuous data collection and further calibrating of parameters can be conducted during pipeline runtime.

3.1.3 Runtime Estimation. Based on the aforementioned computation and communication models, for a given task and device group, we can obtain complete UCG, which includes the computation latency for each layer at every node and the communication matrix that encompasses communication latency functions between each device pair. Thus, the runtime computation and communication latencies for arbitrary shards on any subgroup of devices can be estimated through table lookups and function calculations, supporting decision-making in parallelism planning strategies.

Moreover, given memory constraints, not all shards are feasible on certain devices. Therefore, we also follow [6] to estimate the runtime memory requirements for shards, which accounts for the weight tensor, in/out tensor, ephemeral tensor, and resident buffer[2] to prevent out-of-memory (OOM).

3.2 Multi-pipeline Parallelism Planner (MPP)

EPipe introduces MPP to produce a multi-pipeline parallelism solution. For a multi-pipeline parallelism strategy, the solution space $[\text{pipe}_1, \text{pipe}_2, ...]$ encompasses all potential pipeline quantities and pipeline types (including all various pipeline placement strategies across arbitrary device subgroups). Due to this high complexity, the maximum pipeline quantity $S_{\max} = \lfloor \sum_{i \in \mathbb{D}} m_i / P \rfloor$ can restrict the solution space, where $\sum_{i \in \mathbb{D}} m_i$ represents the total memory of all devices, and P represents the DNN size. Moreover, MPP designs a **Single-Pipeline Placement Generator** to produce optimal placement solutions for any single pipeline within a specified device subgroup (i.e., all valuable pipeline types), thus further simplifying the solution space. Based on these, the **Multi-Pipeline Placement Solver** implemented in MPP can strategize superior multi-pipeline placement solutions.

3.2.1 Single-Pipeline Placement Generator. The Generator is designed to generate arbitrary device subgroups' optimal pipeline placement solutions by implementing a DP procedure.

Concretely, during single-pipeline inference, the slowest stage directly limits the pipeline's performance. Therefore, the sub-problem of DP can be defined as finding the minimum slowest stage time of running the model's first i layers on the selected device subgroup $\mathbb{S}$ (where $\mathbb{S} \subseteq \mathbb{D}$), denoted as $H(i, \mathbb{S}, d_k)$, where d_k is the next device to be considered. For the entire DNN with L layers, the optimal placement solution on any specified device subgroup S is determined by the minimum slowest stage time $H(L, \mathbb{S}, \varnothing)$.

Due to asynchronous computation and communication on the worker, the stage time T_{st} can be formalized as:

$$T_{st}\left(i, j, d_k, d_l, P_j\right) = \max\left\{T_{cp}\left(i, j, d_k\right), T_{cm}\left(d_k, d_l, X_j\right)\right\} \tag{4}$$

where $T_{cp}\left(i, j, d_k\right)$ denotes the computation time from i-th to j-th layer at device d_k, and $T_{cm}\left(d_k, d_l, I_j\right)$ represents the communication time from device d_k to device d_l with intermediate output size of X_j. Therefore, the state transition equation can be formulated as:

$$\begin{aligned} &H\left(j, S \cup d_k, d_l\right) = \\ &\min_{0 \le i < j \le L; d_k, d_l \in \mathbb{D} \setminus \mathbb{S}} \max\left\{\left(H\left(i, \mathbb{S}, d_k\right), T_{st}\left(i, j, d_k, d_l, X_j\right)\right\}\right. \end{aligned} \tag{5}$$

Due to the constraints imposed by T_{st}, for any given device subgroup $\mathbb{S}$, the optimal placement solution $H(L, \mathbb{S}, \varnothing)$ utilizes the actual device subgroup $\mathbb{U} \subseteq \mathbb{S}$, not necessarily $\mathbb{U} = \mathbb{S}$. This relationship implies $M \le 2^{|\mathbb{D}|}$, where M represents the number of valuable pipeline types. In the DP procedure, we introduce a Pipeline Type Set (PTS) to store all valuable pipeline types, denoted as $PTS = \{(\mathbb{U}_i, H_i, R_i) \mid i \in [1, M]\}$, where i indexes the valuable pipeline type pipe $_i$. Here, $\mathbb{U}_i$ denotes the actual devices encompassed in pipe_i, H_i represents the corresponding $H(L, \mathbb{S}, \varnothing)$, and R_i outlines the specific placement solution, i.e., the mapping of DNN shards to their respective devices. The PTS is the simplified solution space.

The specific algorithmic procedure is illustrated in Algorithm 1. When identical devices with the same computation and communication capabilities exist, the computational complexity can be expressed as $O\left(\prod_{i=1}^{N}(n_i + 1) \times L^2 \times N^2\right)$. Here, N denotes the number of categories in $\mathbb{D}$, with each category i having n_i devices.

3.2.2 Multi-Pipeline Placement Solver. After simplifying solution space, the Solver utilizes a Genetic Algorithm (GA) to obtain placement solutions for multiple pipelines.

In GA, we treat each multi-pipeline solution as a chromosome, with multiple such chromosomes forming a population. Each chromosome consists of $S_{\max}$ genes, indicating that the multi-pipeline solution encompasses a maximum of $S_{\max}$ pipelines. Each gene selects a potential pipeline type index i from PTS, formalized as $g_k = i$ for $k \in [1, S_{\max}]$ and $i \in [0, M]$, where $i = 0$ indicates that the pipeline type corresponding to this gene is absent. The fitness for each chromosome C is defined as $fitness(C) = \sum_{g_k \in C} \frac{1}{H_{g_k}}$.

[2]Due to the presence of input and output queues, the estimated runtime memory for each shard directly incorporates the sum of communication volumes from the preceding and succeeding stages.

Algorithm 1 DP-based Single-Pipeline Placement Strategy

Require: $\mathbb{T}$: Model with L layers, intermediate output set X, and memory requirements set M; $\mathbb{D}$: available devices with memory m_d for $d \in \mathbb{D}$;
Ensure: PTS
Initialize $h(i, \mathbb{S}, d_k) \leftarrow +\infty$ for all $i \in L, \mathbb{S} \subseteq D, d_k \in D$;
Initialize $h(0, \emptyset, \varnothing) \leftarrow 0$, sets $\mathbb{I}, PTS \leftarrow \emptyset$, dictionary $Z \leftarrow \{\}$;
for $i = 0$ to $L - 1$ **do**
  for each subgroup $S \subseteq D$ **do**
    for each $d_k \in D \setminus S$ **do**
      for $j = i + 1$ to L **do**
        if $\sum_{k=i}^{j} M_k > m_u$ **then**
          Break
        $C \leftarrow$ Calculate Eq.5
        if $j == L$ **then**
          Append $(i, \mathbb{S}, d_k, H(L, \mathbb{S} \cup d_k, \varnothing))$ to $\mathbb{I}$
        else
          for each $d_l \in \mathbb{D} \setminus \mathbb{S} \setminus \{d_k\}$ **do**
            if $C < h(j, \mathbb{S} \cup \{d_k\}, d_l)$ **then**
              $h(j, \mathbb{S} \cup \{d_k\}, d_l) = C$
              $Z \leftarrow \{(j, \mathbb{S} \cup \{d_k\}, d_l) : (i, d_k)\}$
for each $i \in \mathbb{I}$ **do**
  Append $(\mathbb{U}, H, R) =$ Backtrace(i, Z) to PTS
return PTS

To ensure a performance lower bound for the generated multi-pipeline solution, the initial population includes a chromosome $\left[\text{index}(\text{pipe}_{\text{best}}), 0, 0, \ldots\right] \in \mathbb{N}_0^{S_{\max}}$, which represents the optimal single-pipeline solution for the entire device group within PTS. The other chromosomes are initialized randomly. In each generation, the algorithm employs a fitness proportionate selection strategy to choose two chromosomes, which then undergo crossover and mutation to form two new chromosomes, repeating the process until a new population is established. After many generations, the chromosome with the highest fitness throughout the GA procedure is chosen as the best solution, as illustrated in Algorithm 2.

Algorithm 2 GA-based Multi-Pipeline Placement Strategy

Require: PTS, S: population size, P_c: crossover probability, P_m: mutation probability, S_{max}: maximum pipeline quantity, E: number of generations
Ensure: Multi-Pipeline parallelism solution $Answer$.
Initialize population $\mathbb{P}$ with S_{max} genes per chromosome,$\mathbb{P}_{next}$ an empty set
for each iteration in E **do**
  Calculate $fitness(C)$ for all $C \in \mathbb{P}$
  Record the chromosome C_{best} with the best fitness.
  while $size(\mathbb{P}_{next}) < S$ **do**
    Select two chromosomes (C_i, C_{i+1}) using fitness proportionate selection
    Generate a random probability p_1.
    if $P_c > p_1$ **then**
      Randomly choose a position k and exchange the genes:
      $C'_i = [C_i[1:k], C_{i+1}[k+1:S_{max}]]$
      $C'_{i+1} = [C_{i+1}[1:k], C_i[k+1:S_{max}]]$
    Generate a random probability p_2.
    if $P_m > p_2$ **then**
      Randomly change one gene in C'_i and C'_{i+1}
    Add C'_i, C'_{i+1} into $\mathbb{P}_{next}$.
  $\mathbb{P} = \mathbb{P}_{next}, \mathbb{P}_{next} = \emptyset$
Calculate $fitness(C)$ for all $C \in \mathbb{P}$ and update C_{best}
Generate $Answer$ using PTS and C_{best}
return $Answer$

3.3 Implementation

EPipe uses PyTorch with Gloo as the communication backend. The master is responsible for planning and dispatching the complete scheduling strategy to every worker involved in each pipeline, and each pipeline is managed independently by different threads. All workers receive the full parallelism solution specific to their respective pipelines. To realize the offline characteristic of EPipe, the master employs `isend()` to asynchronously dispatch commands, whereas workers maintain a task queue to asynchronously receive `irecv()` these directives. Task sequence numbers are encoded as a global logical clock to ensure task order consistency.

Each worker is an individual process incorporating one computation thread for task inference and two communication threads for managing input and output queues, respectively. Building on this, EPipe employs `send()` and `recv()` functions to facilitate synchronous communication between each stage of the single pipeline, ensuring the consistency and orderly processing of data within the pipeline stages.

4 Experiment

4.1 Experiment Setup

Testbed setup: All experiments are conducted in a practical edge LAN testbed. The experimental network is supported by a *TP-LINK TL-SG2024D switch* including 24 RJ45 ports configurable to 10/100/1000Base-T. The testbed integrates two kinds of heterogeneous devices: *NVIDIA Jetson AGX Xavier*: features a Volta GPU with 512 CUDA cores and 64 Tensor cores, an 8-core Carmel ARMv8.2 CPU, and 16GB of 256-bit LPDDR4x memory with 137GB/s bandwidth. *NVIDIA Jetson Xavier NX*: equipped with a Volta GPU with 384 CUDA cores and 48 Tensor cores, a 6-core Carmel ARMv8.2 CPU, and 8GB of 128-bit LPDDR4x memory with 59.7GB/s bandwidth. Both are configured with JetPack 5.1.3, featuring Ubuntu 20.04, CUDA 11.4, Python 3.8, and PyTorch 1.8.

Compared to cloud environments, edge devices exhibit greater diversity in computation, memory, and communication capabilities. To facilitate a more comprehensive comparison, we intentionally constrained the performance of device groups and enhanced heterogeneity, as depicted in Table 2, which presents four configurations. Cases 1 and 2 exhibit memory heterogeneity and limitations, Cases 2 and 3 show computation heterogeneity and limitations, and Cases 3 and 4 demonstrate communication heterogeneity and limitations.

DNN Models: Following [7], to evaluate the effectiveness of EPipe in tasks across the computer vision (CV) and natural language processing (NLP) domains, three widely used models are selected: ViT-Large [4], ViT-Huge, and BERT-Large [3]. The ViT-Large model comprises 24 Transformer layers with approximately 300 million parameters. The ViT-Huge model includes 32 Transformer layers,

Table 2. Configurations with Increasing Heterogeneity

Case	Devices	Power budget & Online CPU	Memory	Bandwidth
1	3 × AGX	n/a & 8	16 GB	1 Gbps
	3 × NX	15W & 6	8 GB	1 Gbps
2	3 × AGX	n/a & 8	4/2/1 GB	1 Gbps
	3 × NX	15W & 6	4/2/1 GB	1 Gbps
3	1 × AGX	n/a & 8	4 GB	1 Gbps
	1 × AGX	15W & 4	2 GB	1 Gbps
	1 × AGX	10W & 2	1 GB	1 Gbps
	1 × NX	15W & 6	4 GB	1 Gbps
	1 × NX	15W & 2	2 GB	1 Gbps
	1 × NX	10W & 2	1 GB	1 Gbps
4	1 × AGX	n/a & 8	4 GB	1 Gbps
	1 × AGX	15W & 4	2 GB	100 Mbps
	1 × AGX	10W & 2	1 GB	10 Mbps
	1 × NX	15W & 6	4 GB	1 Gbps
	1 × NX	15W & 2	2 GB	100 Mbps
	1 × NX	10W & 2	1 GB	10 Mbps

[1] 'n/a' denotes MAXN mode with no power limitations.

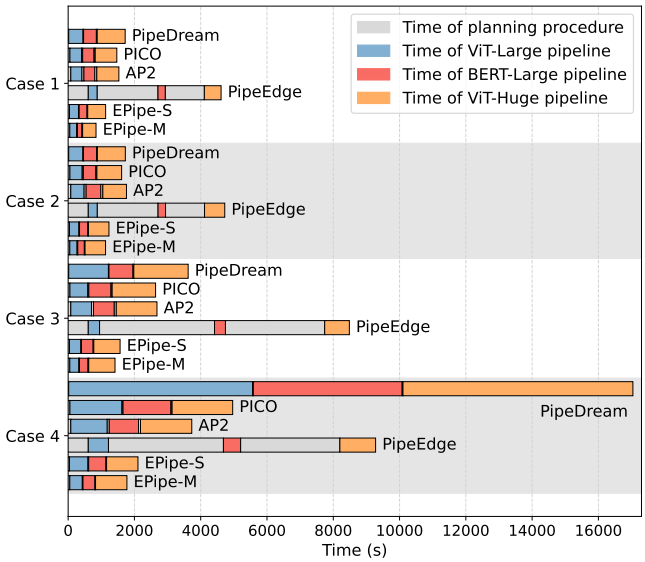


Figure 6. Duration comparison of task streams.

totaling about 632 million parameters. Input images are from ImageNet [2] batch size of 64 and resized after the embedding layer with a uniform input dimension to the models. The BERT-Large model consists of 24 Transformer layers and 340 million parameters. Input texts are from SST-2 dataset [22] batch size of 48 with texts processed to a maximum sequence length of 512.

Baselines: To verify the effectiveness of EPipe, we tested the single-pipelined (**EPipe-S**) and multi-pipelined (**EPipe-M**) solutions in the four cases mentioned above. In addition, four representative baselines are introduced. **PipeDream** [15]: A classic pipeline parallelism scheme extensively employed in homogeneous cloud environments. **PipeEdge** [7]: An optimal single pipeline parallelism strategy designed for heterogeneous environments, based on profiling. **PICO**[27] and **AP^2**[18] represent the latest cost model-based approaches for heterogeneous environments, both utilizing bandwidth-based methods for communication latency predictions. Specifically, PICO employs a FLOPs-based method, while AP^2 utilizes a decision tree method for computation latency predictions.

4.2 Comparison Results

4.2.1 Comparison in Task Streams. We conducted evaluations of EPipe against all baseline methods during a streaming DNN inference task, as detailed in Figure 6. Concretely, it presents the time taken by the different methods to continuously process 3000 items each for ViT-Large, Bert-Large and ViT-Huge in different heterogeneous configurations. Clearly, PipeDream's pipeline efficiency dramatically declines as device heterogeneity increases due to its uniform shard division based on assumptions of device homogeneity. Although PipeEdge shows lower pipeline runtime overhead, the substantial resource wastage from profiling and online planning during task switching substantially degrades its overall efficiency. PICO, AP^2, and EPipe facilitate offline pipeline parallelism planning without significant task switching overheads. However, the inaccurate prediction of latency by PICO and AP^2 leads to a loss of runtime efficiency, preventing the realization of optimal pipeline runtime performance. Additionally, this loss in efficiency worsens as resource constraints become more severe. In contrast, EPipe not only enables seamless task switching but also enhances pipeline runtime efficiency, thereby yielding the best performance. Moreover, it can be noted that compared to single-pipeline parallelism solutions (EPipe-S), multi-pipeline parallelism solutions (EPipe-M) achieve greater overall efficiency.

4.2.2 Comparison in Single Tasks. To further examine EPipe's performance in pipeline runtime, we conducted a detailed comparison of all methods across single inference tasks, as illustrated in Figure 7. Again, it can be seen that PipeDream lacks awareness of device heterogeneity, which results in its worst throughput in heterogeneous scenarios. AP^2 and PICO, relying on inaccurate predictions, do not perform as well as PipeEdge which is profiling-based and can find the theoretical optimal single-pipeline parallelism solution through its strategy. Nevertheless, EPipe's TDC engages in more detailed extraction and modeling of both task and device characteristics, allowing its single-pipeline solution to deliver performance on par with PipeEdge. Furthermore, by expanding to include the multi-pipeline parallelism solutions, EPipe-M effectively utilizes devices to achieve superior throughput compared to other methods.

In the heavily heterogeneous and constrained communication environment of Case 4, EPipe-S surpasses PipeEdge in performance. This can be attributed to PipeEdge's reliance on single-point measurements, which fail to account for fluctuations between current and pipeline runtime network traffic. In contrast, EPipe-S improves accuracy by incorporating pipeline traffic difference features, providing a more reasonable estimation of communication latencies than simply profiling current bandwidth. Additionally, in the task of ViT-Huge, storage constraints make it infeasible for EPipe-M to identify a more optimal multi-segment pipeline configuration. At this point, the performance of EPipe-M converges to that of EPipe-S, yet the efficiency remains very high.

4.3 Ablation Analysis

We conducted detailed ablation studies on the EPipe framework, focusing on its key components: TDC and MPP. Given the essential nature of these components, we redefined the scope of this ablation study: TDC is shifted to the profiling strategy, and MPP is adjusted to operate under a single-pipeline parallel strategy. Therefore, PipeEdge serves as our baseline. The experiment uses the same setup as the comparison in task streams, with results shown in Table 3. Notably, TDC's substitution of profiling with latency prediction significantly reduced the time for latency awareness, achieving up to a 2.57× speed up across all cases. Although MPP, which enables multi-pipeline inference, yields improvements over the baseline, these gains are relatively less pronounced due to the extended durations required for profiling. Ultimately, by combining TDC and MPP, EPipe achieved up to a 3.22× speed up for task streaming, further validating the effectiveness of these components.

4.4 Overhead Analysis

EPipe's overhead stems from two phases: initialization training and planning procedure. The initialization training phase occurs once before EPipe deployment on the device group and is responsible for training the latency prediction models in the TDC. Its overhead involves data collection, encompassing computation and communication tasks that range from 30 to 50 minutes for all involved devices; and predictive model training, where parameters are fitted using the collected data, typically converging in under

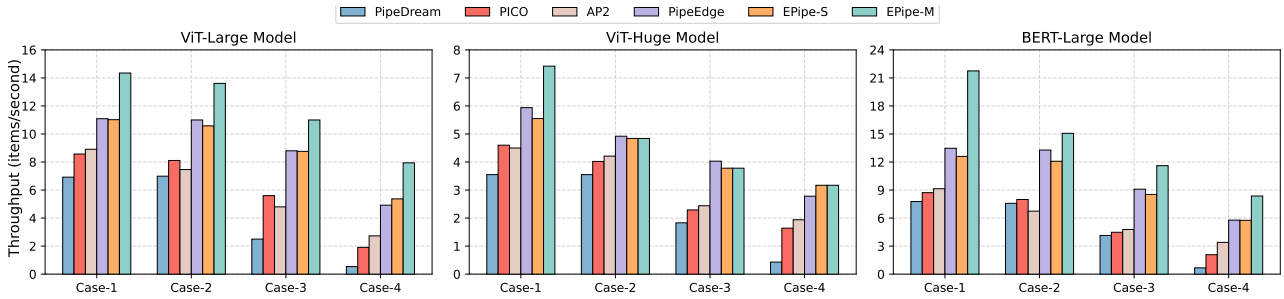


Figure 7. Throughput comparison in single tasks.

Table 3. Ablation study on each component of EPipe.

Case	Variations	TDC	MPP	Task Stream Throughput (item/second)
1	Baseline			1.44 (×1.00)
	& TDC	✓		2.77 (×1.92)
	& MPP		✓	1.63 (×1.13)
	EPipe(ours)	✓	✓	3.90 (×2.71)
2	Baseline			1.37 (×1.00)
	& TDC	✓		2.41 (×1.76)
	& MPP		✓	1.41 (×1.03)
	EPipe(ours)	✓	✓	2.84 (×2.08)
3	Baseline			0.80 (×1.00)
	& TDC	✓		2.06 (×2.57)
	& MPP		✓	0.82 (×1.02)
	EPipe(ours)	✓	✓	2.32 (×2.90)
4	Baseline			0.66 (×1.00)
	& TDC	✓		1.59 (×2.41)
	& MPP		✓	0.75 (×1.13)
	EPipe(ours)	✓	✓	2.13 (×3.22)

3 minutes. After initializing the TDC, EPipe executes the offline parallel planning procedure on the master node, independently of the worker nodes. The procedure incurs costs during the predicting and planning phases: prediction for Vit-Huge across six heterogeneous devices is completed in 0.6 seconds, and planning on these devices takes no more than 10 seconds.

5 Related Work

In recent years, with the expansion of model sizes, various model parallelism techniques have become the focal point of intense research in the cloud domain, such as tensor parallelism [19, 23], pipeline parallelism [8, 20], and their hybrid parallelism [5, 9, 30]. However, considering the limited resources and extensive heterogeneity of edge devices, these methodologies can not be directly applicable in such scenarios. Parallel computation of tensors often necessitates extensive collective communication, which can introduce considerable synchronization overhead. Pipeline parallelism can substantially accelerate inference by overlapping communication with computation. However, most studies (e.g., [8, 10, 15, 16, 20]) in cloud environments rely on homogeneous assumptions to simplify the operator placement problem. In contrast, accounting for high heterogeneity is critical in edge scenarios.

Recently, some studies [7, 12, 18, 27, 28] have focused on designing pipeline parallelism specifically for heterogeneous edge computing environments. To estimate computation and communication latencies within the runtime for parallelism planning purposes, existing research can be broadly categorized into two groups: *Profiling-based methods* [7, 12, 17] decompose DNNs into smaller units and execute them on various physical devices to measure corresponding latencies. HeterPS [12] employs profiling and incorporates Amdahl's Law to estimate overheads, then uses reinforcement learning to create cost-efficient parallelism scheduling plans that meet throughput requirements. HetPipe [17] and PipeEdge [7] both directly employ profiling strategies and integrate linear programming or dynamic programming approaches to optimize pipeline solutions, respectively. The profiling process necessitates occupying physical devices, thereby incurring substantial resources wastage during planning. *Cost model-based methods* [18, 27, 28] rely on predictive cost models to estimate latencies without actual execution on physical devices. Both PICO [27] and DeepSlicing [28] employ FLOPs and bandwidth metrics to predict latency. Building upon this foundation, PICO [27] leverages graph partitioning and many-to-many mapping algorithms to search for the optimal single-pipeline solution, and DeepSlicing [28] utilizes a greedy strategy for task allocation and dynamic adjustment. AP^2 [18] utilizes a Gradient Boosted Decision Trees (GBDT) model to estimate latencies based on task statistics and device configuration, subsequently employing a greedy-based scheduling algorithm to search the pipeline solution. However, as demonstrated in Figure 2, these cost models fail to predict latencies accurately, thereby compromising pipeline runtime efficiency. Unlike previous approaches, this study introduces EPipe, facilitating seamless pipeline switching and superior parallelism planning. Initially, the compatibility of DNNs across device groups is abstracted into a UCG, where node and edge information are enriched through fine-grained modeling. Subsequently, EPipe designs a dynamic programming-based genetic algorithm to generate a multi-pipeline parallelism solution.

6 Conclusion

In this paper, we have introduced EPipe, a novel pipeline inference framework designed to enhance the efficiency and scalability of DNN model deployment on heterogeneous edge devices through high-quality offline parallelism planning. EPipe integrates the Task-Device Co-analyzer (TDC), which enables detailed and accurate estimations of computation and communication latencies, along with the Multi-pipeline Parallelism Planner (MPP), paving the way for deriving an effective multi-pipeline parallelism solution. Through extensive empirical evaluations, EPipe has demonstrated its capability to significantly improve inference task throughput in diverse and realistic edge environments. A comprehensive experimental evaluation on an edge testbed confirms the effectiveness of EPipe, demonstrating significant improvements in inference task throughput for both task streams and single tasks.

References

[1] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Q. Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. 2018. TVM: An Automated End-to-End Optimizing Compiler for Deep Learning. In *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, Andrea C. Arpaci-Dusseau and Geoff Voelker (Eds.). USENIX Association, 578–594.

[2] Jia Deng, Wei Dong, Richard Socher, Li-Jia Li, Kai Li, and Li Fei-Fei. 2009. ImageNet: A large-scale hierarchical image database. In *2009 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2009), 20-25 June 2009, Miami, Florida, USA*. IEEE Computer Society, 248–255.

[3] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, NAACL-HLT 2019, Minneapolis, MN, USA, June 2-7, 2019, Volume 1 (Long and Short Papers)*, Jill Burstein, Christy Doran, and Thamar Solorio (Eds.). Association for Computational Linguistics, 4171–4186.

[4] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. 2021. An Image is Worth 16x16 Words: Transformers for Image Recognition at Scale. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net.

[5] Shiqing Fan, Yi Rong, Chen Meng, Zongyan Cao, Siyu Wang, Zhen Zheng, Chuan Wu, Guoping Long, Jun Yang, Lixue Xia, Lansong Diao, Xiaoyong Liu, and Wei Lin. 2021. DAPPLE: a pipelined data parallel approach for training large models. In *PPoPP '21: 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, Virtual Event, Republic of Korea, February 27- March 3, 2021*, Jaejin Lee and Erez Petrank (Eds.). ACM, 431–445.

[6] Yanjie Gao, Yu Liu, Hongyu Zhang, Zhengxian Li, Yonghao Zhu, Haoxiang Lin, and Mao Yang. 2020. Estimating GPU memory consumption of deep learning models. In *ESEC/FSE '20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020*, Prem Devanbu, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1342–1352.

[7] 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 *25th Euromicro Conference on Digital System Design, DSD 2022, Maspalomas, Spain, August 31 - Sept. 2, 2022*. IEEE, 298–307.

[8] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Xu Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V. Le, Yonghui Wu, and Zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism. In *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (Eds.). 103–112.

[9] Fanxin Li, Shixiong Zhao, Yuhao Qing, Xusheng Chen, Xiuxian Guan, Sen Wang, Gong Zhang, and Heming Cui. 2023. Fold3D: Rethinking and Parallelizing Computational and Communicational Tasks in the Training of Large DNN Models. *IEEE Trans. Parallel Distributed Syst.* 34, 5 (2023), 1432–1449.

[10] Shigang Li and Torsten Hoefler. 2021. Chimera: efficiently training large-scale neural networks with bidirectional pipelines. In *International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2021, St. Louis, Missouri, USA, November 14-19, 2021*, Bronis R. de Supinski, Mary W. Hall, and Todd Gamblin (Eds.). ACM, 27.

[11] Ying Li, Yifan Sun, and Adwait Jog. 2023. Path Forward Beyond Simulators: Fast and Accurate GPU Execution Time Prediction for DNN Workloads. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*. ACM, 380–394.

[12] Ji Liu, Zhihua Wu, Danlei Feng, Minxu Zhang, Xinxuan Wu, Xuefeng Yao, Dianhai Yu, Yanjun Ma, Feng Zhao, and Dejing Dou. 2023. HeterPS: Distributed deep learning with reinforcement learning based scheduling in heterogeneous environments. *Future Gener. Comput. Syst.* 148 (2023), 106–117.

[13] Weihong Liu, Jiawei Geng, Zongwei Zhu, Yang Zhao, Cheng Ji, Changlong Li, Zirui Lian, and Xuehai Zhou. 2024. Ace-Sniper: Cloud-Edge Collaborative Scheduling Framework With DNN Inference Latency Modeling on Heterogeneous Devices. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.* 43, 2 (2024), 534–547.

[14] Zichang Liu, Jue Wang, Tri Dao, Tianyi Zhou, Binhang Yuan, Zhao Song, Anshumali Shrivastava, Ce Zhang, Yuandong Tian, Christopher Ré, and Beidi Chen. 2023. Deja Vu: Contextual Sparsity for Efficient LLMs at Inference Time. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 22137–22176.

[15] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R. Devanur, Gregory R. Ganger, Phillip B. Gibbons, and Matei Zaharia. 2019. PipeDream: generalized pipeline parallelism for DNN training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles, SOSP 2019, Huntsville, ON, Canada, October 27-30, 2019*, Tim Brecht and Carey Williamson (Eds.). ACM, 1–15.

[16] Deepak Narayanan, Amar Phanishayee, Kaiyu Shi, Xie Chen, and Matei Zaharia. 2021. Memory-Efficient Pipeline-Parallel DNN Training. In *Proceedings of the 38th International Conference on Machine Learning, ICML 2021, 18-24 July 2021, Virtual Event (Proceedings of Machine Learning Research, Vol. 139)*, Marina Meila and Tong Zhang (Eds.). PMLR, 7937–7947.

[17] Jay H. Park, Gyeongchan Yun, Chang M. Yi, Nguyen T. Nguyen, Seungmin Lee, Jaesik Choi, Sam H. Noh, and Young-ri Choi. 2020. HetPipe: Enabling Large DNN Training on (Whimpy) Heterogeneous GPU Clusters through Integration of Pipelined Model Parallelism and Data Parallelism. In *2020 USENIX Annual Technical Conference, USENIX ATC 2020, July 15-17, 2020*, Ada Gavrilovska and Erez Zadok (Eds.). USENIX Association, 307–321.

[18] Hongjian Shi, Weichu Zheng, Zifei Liu, Ruhui Ma, and Haibing Guan. 2023. Automatic Pipeline Parallelism: A Parallel Inference Framework for Deep Learning Applications in 6G Mobile Communication Systems. *IEEE J. Sel. Areas Commun.* 41, 7 (2023), 2041–2056.

[19] 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. *CoRR* abs/1909.08053 (2019).

[20] Jaeyong Song, Jinkyu Yim, Jaewon Jung, Hongsun Jang, Hyung-Jin Kim, Youngsok Kim, and Jinho Lee. 2023. Optimus-CC: Efficient Large NLP Model Training with 3D Parallelism Aware Communication Compression. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2023, Vancouver, BC, Canada, March 25-29, 2023*, Tor M. Aamodt, Natalie D. Enright Jerger, and Michael M. Swift (Eds.). ACM, 560–573.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2017. Attention is All you Need. In *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, Isabelle Guyon, Ulrike von Luxburg, Samy Bengio, Hanna M. Wallach, Rob Fergus, S. V. N. Vishwanathan, and Roman Garnett (Eds.). 5998–6008.

[22] Alex Wang, Amanpreet Singh, Julian Michael, Felix Hill, Omer Levy, and Samuel R. Bowman. 2019. GLUE: A Multi-Task Benchmark and Analysis Platform for Natural Language Understanding. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net.

[23] Haoran Wang, Chong Li, Thibaut Tachon, Hongxing Wang, Sheng Yang, Sébastien Limet, and Sophie Robert. 2021. Efficient and Systematic Partitioning of Large and Deep Neural Networks for Parallelization. In *Euro-Par 2021: Parallel Processing - 27th International Conference on Parallel and Distributed Computing, Lisbon, Portugal, September 1-3, 2021, Proceedings (Lecture Notes in Computer Science, Vol. 12820)*, Leonel Sousa, Nuno Roma, and Pedro Tomás (Eds.). Springer, 201–216.

[24] Yu Wang, Gu-Yeon Wei, and David Brooks. 2019. Benchmarking TPU, GPU, and CPU Platforms for Deep Learning. *CoRR* abs/1907.10701 (2019).

[25] Guangxuan Xiao, Ji Lin, Mickaël Seznec, Hao Wu, Julien Demouth, and Song Han. 2023. SmoothQuant: Accurate and Efficient Post-Training Quantization for Large Language Models. In *International Conference on Machine Learning, ICML 2023, 23-29 July 2023, Honolulu, Hawaii, USA (Proceedings of Machine Learning Research, Vol. 202)*, Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett (Eds.). PMLR, 38087–38099.

[26] Yi Xiong, Wenjie Zhai, Xueyong Xu, Jinchen Wang, Zongwei Zhu, Cheng Ji, and Jing Cao. 2023. Ability-aware knowledge distillation for resource-constrained embedded devices. *J. Syst. Archit.* 141 (2023), 102912.

[27] Xiang Yang, Zikang Xu, Qi Qi, Jingyu Wang, Haifeng Sun, Jianxin Liao, and Song Guo. 2024. PICO: Pipeline Inference Framework for Versatile CNNs on Diverse Mobile Devices. *IEEE Trans. Mob. Comput.* 23, 4 (2024), 2712–2730.

[28] Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. 2021. DeepSlicing: Collaborative and Adaptive CNN Inference With Low Latency. *IEEE Trans. Parallel Distributed Syst.* 32, 9 (2021), 2175–2187.

[29] Shixiong Zhao, Fanxin Li, Xusheng Chen, Xiuxian Guan, Jianyu Jiang, Dong Huang, Yuhao Qing, Sen Wang, Peng Wang, Gong Zhang, Cheng Li, Ping Luo, and Heming Cui. 2022. vPipe: A Virtualized Acceleration System for Achieving Efficient and Scalable Pipeline Parallel DNN Training. *IEEE Trans. Parallel Distributed Syst.* 33, 3 (2022), 489–506.

[30] Lianmin Zheng, Zhuohan Li, Hao Zhang, Yonghao Zhuang, Zhifeng Chen, Yanping Huang, Yida Wang, Yuanzhong Xu, Danyang Zhuo, Eric P. Xing, Joseph E. Gonzalez, and Ion Stoica. 2022. Alpa: Automating Inter- and Intra-Operator Parallelism for Distributed Deep Learning. In *16th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2022, Carlsbad, CA, USA, July 11-13, 2022*, Marcos K. Aguilera and Hakim Weatherspoon (Eds.). USENIX Association, 559–578.