MemoriaNova: Optimizing Memory-Aware Model Inference for Edge Computing

RENJUN ZHANG, Shanghai Jiao Tong University, Shanghai, China
TIANMING ZHANG, Shanghai Jiao Tong University, Shanghai, China
ZINUO CAI, Shanghai Jiao Tong University, Shanghai, China
DONGMEI LI, Beijing Institute of Microelectronics Technology, Shanghai, China
RUHUI MA, Computer Science, Shanghai Jiao Tong University, Shanghai, China
BUYYA RAJKUMAR, The University of Melbourne, Melbourne, Australia

In recent years, deploying deep learning models on edge devices has become pervasive, driven by the increasing demand for intelligent edge computing solutions across various industries. From industrial automation to intelligent surveillance and healthcare, edge devices are being leveraged for real-time analytics and decision-making. Existing methods face two challenges when deploying machine learning models on edge devices. The first challenge is handling the execution order of operators with a simple strategy, which can lead to a potential waste of memory resources when dealing with directed acyclic graph structure models. The second challenge is that they usually process operators of a model one by one to optimize the inference latency, which may lead to the optimization problem getting trapped in local optima.

We present MemoriaNova, comprising BTSearch and GenEFlow, to solve these two problems. BTSearch is a graph state backtracking algorithm with efficient pruning and hashing strategies designed to minimize memory overhead during inference and enlarge latency optimization search space. GenEFlow, based on genetic algorithms (GA), integrates latency modeling, and memory constraints to optimize distributed inference latency. This innovative approach considers a comprehensive search space for model partitioning, ensuring robust and adaptable solutions. We implement BTSearch and GenEFlow and test them on 11 deep-learning models with different structures and scales. The results show that BTSearch can reach 12% memory optimization compared with the widely used random execution strategy. At the same time, GenEFlow reduces inference latency by 33.9% in distributed systems with four-edge devices.

CCS Concepts: • **Hardware** → **Emerging tools and methodologies**; • **Computing methodologies** → **Distributed computing methodologies**; **Machine learning**; **Optimization algorithms;**

Additional Key Words and Phrases: Deep learning, edge computing, memory optimization, distributed system, inference latency optimization

This work was supported by Shanghai Key Laboratory of Scalable Computing and Systems, and National Key Laboratory of Ship Structural Safety.
Authors' Contact Information: Renjun Zhang, Shanghai Jiao Tong University, Shanghai, China; e-mail: renjun_zhang@sjtu.edu.cn; Tianming Zhang, Shanghai Jiao Tong University, Shanghai, China; e-mail: zhang_tianming@sjtu.edu.cn; Zinuo Cai, Shanghai Jiao Tong University, Shanghai, China; e-mail: kingczn1314@sjtu.edu.cn; Dongmei Li, Beijing Institute of Microelectronics Technology, Shanghai, China; e-mail: lidm772@163.com; Ruhui Ma (Corresponding author), Computer Science, Shanghai Jiao Tong University, Shanghai, China; e-mail: ruhuima@sjtu.edu.cn; Buyya Rajkumar, The University of Melbourne, Melbourne, Victoria, Australia; e-mail: rbuyya@unimelb.edu.au.

ACM 1544-3973/2025/03-ART3
https://doi.org/10.1145/3701997

ACM Reference Format:
Renjun Zhang, Tianming Zhang, Zinuo Cai, Dongmei Li, Ruhui Ma, and Buyya Rajkumar. 2025. MemoriaNova: Optimizing Memory-Aware Model Inference for Edge Computing. *ACM Trans. Arch. Code Optim.* 22, 1, Article 3 (March 2025), 25 pages. https://doi.org/10.1145/3701997

1 Introduction

The artificial intelligence paradigm has experienced significant advancement and widespread applications across various domains. **Deep learning (DL)** methods [45] have achieved state-of-the-art results in many machine learning applications [38], such as object detection, image classification, and face recognition [22]. Traditionally, the inference task [33] of DL models occurs on high-performance cloud servers, necessitating large data transfers and incurring substantial time overhead. To address this challenge, deploying models on edge devices near data sources becomes common [10]. Consequently, researchers explore distributed inference mechanisms that distribute inference workloads across multiple edge devices to mitigate latency [34]. Beyond reducing network transmission load, deploying DL models at the edge confers additional benefits [39]. These include reduced latency, enhanced privacy and security, improved reliability, and offline capability. These advantages make edge deployment an attractive option for various applications requiring real-time or near-real-time processing and decision-making capabilities.

However, inference tasks are often computationally intensive, and the limited resources of edge devices can exacerbate overall latency. For example, in a smart home, the camera processes real-time video data and recognizes visitors. Subsequently, the camera sends the visitor information to the smart speaker, which provides voice announcements based on the recognition results and performs corresponding actions as instructed by the homeowner, such as opening the door or sending an alert. Meanwhile, environmental sensors continuously monitor indoor air quality, temperature, and humidity, adjusting the operation of air conditioners or humidifiers based on the analysis results to ensure a comfortable and healthy home environment. These devices require complex deep learning models for inference, which exceeds the capabilities of a single device.

Although distributed inference [28] has attracted much attention, several challenges remain to be solved. The first challenge is addressing the memory constraint of edge devices during model distribution. Edge devices [44] such as intelligent surveillance cameras [5], intelligent door locks [11], smart TVs [9], and smart speakers [30] typically have limited memory. In contrast, several sources of memory overhead exist when conducting distributed inference. A DL model can be abstracted as a **directed acyclic graph (DAG)**, which means there may be more than one reasonable operator execution order of the model. According to Reference [40], operator execution order influences the lifetime of intermediate tensors of the model, leading to variable memory overhead. Besides, partitioning a model involves operator partition while an operator's type and partition number cause additional memory overhead. Existing methods like in References [48] and [46] only consider latency optimization, not memory constraints. HMCOS [40] reduces the memory footprint of inference tasks by adjusting operator execution order but only on a single GPU. Moreover, traversing the topological sorting of directed acyclic graphs is a **P-Complete (PC)** problem mathematically [2]. Efficiently conducting this search remains a challenging problem.

The second challenge lies in determining a suitable model partition configuration to minimize inference latency. Common distributed strategies for model partitioning encompass horizontal, vertical, and hybrid partitioning. We delve into addressing model partitioning issues under the hybrid partitioning strategy, which considers both horizontal and vertical partitioning, along with the interdependence among operators. This process involves considerations of dimension, partition number, and proportions. The partitioning of operators impacts both computing and communication time, thereby influencing overall inference latency. Moreover, the decision on operator partitioning affects the following adjacent operators. Thus, partitioning a model for reduced inference latency presents a complex optimization problem. Unfortunately, existing solutions often provide coarse-grained approximations. For instance, References [46] and [16] address the operator partition problem individually, which may not guarantee optimal results. These methods typically focus on a single operator partition dimension. Additionally, Reference [16] employs an approximation method to transform the optimization problem into a linear program, which introduces errors and diminishes effectiveness.

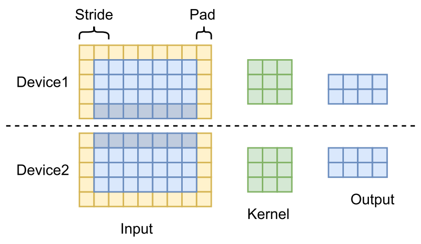
To address the challenges mentioned above during the optimization of inference latency of DL models on memory-constrained distributed edge devices, we conduct a memory-time cost analysis of operator partitioning in model parallelism and propose two optimization methods, namely BTSearch and GenEFlow. BTSearch graph state backtracking algorithm traverses all topological sorting in a DAG structure model. It guarantees to find the optimal operator execution order of a DL model. The result execution order has minimal overall memory overhead without considering operator partition, which enlarges the search space for operator partition optimization. We apply an efficient pruning strategy on BTSearch. The strategy prunes the branches with no potential for better results according to the state of the computation graph. GenEFlow is a GA-based method aiming to optimize the inference latency while satisfying the memory constraints of the edge devices. We model the partition decision of the whole model as a chromosome and consider different operator partition dimensions, thus constructing a more comprehensive search space. GenEFlow can search for the optimal solution from a global perspective through these designs. Moreover, we use constraint violation parameters to guarantee memory constraints.

Our main contributions are as follows: (1) We analyze the memory-time cost of operator partitioning and operator execution order in model parallelism. Specifically, we examine the available partitioning methods for each operator and their memory overhead, calculating the memory consumption for different partitioning methods to determine the optimal partitioning method for each operator. Additionally, we analyze the impact of operator execution order on memory, finding that adjusting the execution order under memory constraints reduces the maximum memory overhead and increases the available memory space per device. (2) We propose the BTSearch, which employs efficient pruning strategies to optimize the execution order of operators in DL models with DAG structures. BTSearch reduces the overall memory overhead and provides a more extensive search space for optimizing inference latency. (3) We introduce the GenEFlow method, which optimizes inference latency for distributed edge devices without altering the model computation results. GenEFlow models the model partition decision as a chromosome and employs GAs for optimization. GenEFlow considers two dimensions of operator partitioning and covers a more extensive search space, offering a more comprehensive search space and robust solution than traditional methods. (4) We merge BTSearch and GenEFlow into MemoriaNova and validate it on 11 deep-learning models. Our results demonstrate significant improvements in memory optimization and inference latency reduction. Specifically, BTSearch achieves up to 12% overall memory optimization, while GenEFlow reduces model inference latency by 33.9% in our distributed edge device system.

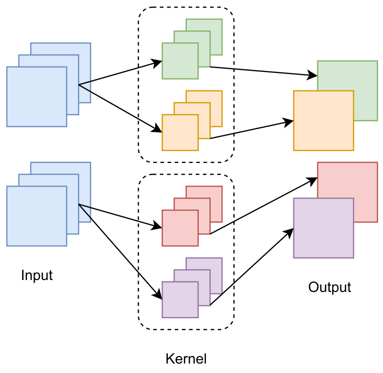
2 Background and Motivation

2.1 Operator Partition Methods and Memory Overhead Analysis

2.1.1 Operator Partition Optimization. Operator partition optimization [41] is vital for efficient model inference [4] on edge devices. It breaks down complex tasks into smaller distributable operators across multiple devices, reducing inference latency and maximizing resource utilization. Determining the correspondence among the input data, operator parameters, and output data becomes necessary to accomplish this objective. The convolution operator's partitioning along



(a) Convolution Operator Partition Along the Dimension of Feature Map Height.



(b) Convolution Operator Partition Along the Dimension of Output Channel.

Fig. 1. Convolution operator partition along two dimensions.

the feature map's high dimension is illustrated in Figure 1(a), with no processing done on the channel dimension, remaining consistent with the original operator.

After partitioning, the output tensors are executed on different devices, each storing a copy of the operator parameters (Kernel). The input tensor is partitioned according to the convolution computation rules, resulting in a small amount of duplicate data, as shown in the gray area in Figure 1(a). Following this partitioning process, the subsequent equation provides the calculation formula for the input data range when partitioning the feature map's high dimension for the convolution operator. If the output tensor's high dimension range is $[x_s, x_e)$, then the corresponding input tensor range is given by the following:

$$[x_s \times S - P, (x_e - 1) \times S + K_h - P], \tag{1}$$

where S represents the Stride, P represents the Padding, and K_h represents the height of the convolution kernel.

2.1.2 Analysis of Memory Overhead in Operator Partitioning. Partitioning operators [27] impact computation time and memory. Concurrently, parallel execution [20] reduces computation time but may raise memory overhead. Additionally, partitioning strategy [29] and device setup determine the balance between time and memory. While parallel execution reduces the computation time by distributing the workload, it may introduce additional memory overhead due to data duplication and synchronization requirements across devices. The choice of partitioning strategy and device configuration plays a crucial role in determining the tradeoff between computation time and memory overhead. Figure 1(b) shows convolutional output channel partitioning, where kernels partition without redundant data. Each device retains a copy of the input tensor. Various partitioning methods result in different memory overheads due to input tensor and kernel memory footprints.

To determine the optimal partitioning method, we perform memory calculations for the obtained operator execution order. We consider different partitioning optimization methods from a memory perspective. The partitioning optimization methods for various types of operators and the resulting memory overhead are shown in Table 1. In the table, "cout" denotes "Channel out," and "fmh" denotes "Feature map height." "len" represents the length of the vector. The operators listed in Table 1 are the leading operators for the slicing operation. The activation

Table 1. Operator Partition Method and Memory Consumption

Operator	Partition Method	Sources of Memory Overhead
Convolution	fmh	Convolution kernel
	cout	Input tensor
Pool	fmh	None
Element-wise addition (Add)	fmh	None
Matrix multiplication (Gemm)	len	None

layer is merged into the convolution operator. Due to the direct transfer of the corresponding data to the connected device before the start of the calculation for each operator and the absence of tensor reshaping operations, operators that reorder tensor data have their execution process combined into the data communication phase. The lack of a symbol in Table 1 indicates that partitioning the operator will not incur additional memory overhead. The memory analysis and the handling of the partitioning overhead for convolutional operators are particularly beneficial, given that convolutional operators typically have a large parameter size in DL models.

Taking the convolution operator as an example, we introduce the method for determining its partitioning. Assuming there are n devices in the distributed system, each device has an available memory limit:

$$\mathcal{M} = [m_1, m_2, \ldots, m_n]. \tag{2}$$

The total available memory limit for each device is as follows:

$$M_{full} = \sum_{i=1}^{n} m_i. \tag{3}$$

For the current convolution operation *Conv*, memory allocation includes M_{in} for input, M_{out} for output, M_{kernel} for parameters, and M_{others} for intermediate tensors. The operator is partitioned into k_1 partitions along the output channel dimension, respecting device memory limits. We have the following:

$$M_{others} + k_1 * M_{in} + M_{kernel} + M_{out} \leq M_{full}. \tag{4}$$

The current convolution operator is partitioned along the height dimension of the output tensor feature map, with the number of partitions being k_2. Similarly,

$$M_{others} + M_{in} + k_2 * M_{kernel} + M_{out} \leq M_{full}. \tag{5}$$

Based on the current operator parameters and the current state of the computation graph, we can calculate the values of k_1 and k_2 and then round them down to yield the final results. When $k_1 < k_2$, we adopt output channel (cout) partitioning for the current convolution. Otherwise, we assume feature map height (fmh) partitioning.

2.2 Analysis of Operator Execution Order on Memory Overhead

In DL model inference, the operator arrangement in computational graphs impacts memory usage. Sequential execution causes fluctuating memory footprints, especially in models with multi-branch structures. Variability arises from memory allocation for tensors, parameters, and results. Memory remains constant for simpler models with one input/output tensor. However, complex models with multi-branch structures introduce memory management challenges. Different operator execution orders impact memory overhead, emphasizing the need for efficient topology sorting. Optimization can reduce memory overhead, leading to smoother inference processes. The following example illustrates this process.

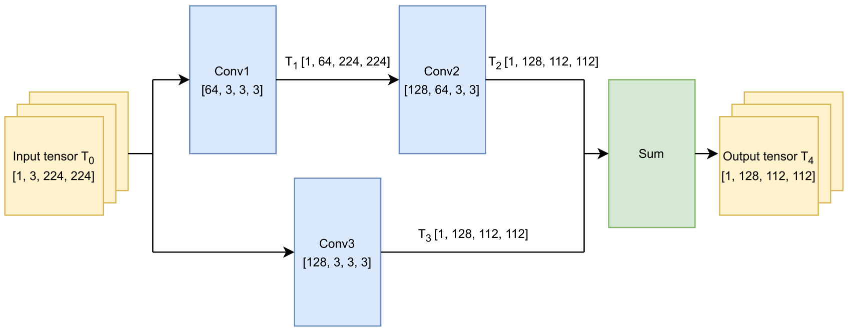
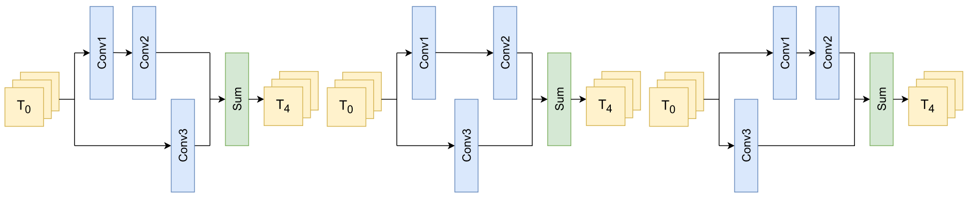


Fig. 2. Example of the impact of operator execution order on memory footprint.



(a) Execution Order1 (b) Execution Order2 (c) Execution Order3

Fig. 3. Three different execution orders of the example model.

Table 2. Memory Footprint Analysis of Different Operator Execution Orders (Metric: KB)

Execution Sequence	M_{i1}	M_{e1}	M_{i2}	M_{e2}	M_{i3}	M_{e3}	M_{i4}	M_{e4}	M_{i5}	Memory
Order1	588	13,139	13,132	19,692	6,860	13,146	12,544	12,544	6,272	19,692
Order2	588	13,139	13,132	19,418	18,816	25,376	12,544	12,544	6,272	25,376
Order3	588	7,329	6,860	19,411	18,816	25,376	12,544	12,544	6,272	25,376

The model in Figure 2 demonstrates a single-input, single-output model with four operators and two branching dataflows. There are three valid execution sequences that conform to topological sorting: Order1 = [Conv1, Conv2, Conv3, Sum]; Order2 = [Conv1, Conv3, Conv2, Sum]; Order3 = [Conv3, Conv1, Conv2, Sum]. Operator execution orders are shown in Figure 3 as (a), (b), and (c).

For a float32 data type, the memory space for tensor T_0 is calculated as follows: $Mem(T_0) = 1 \times 3 \times 224 \times 224 \times 4/1,024 = 588$KB. Similarly, the memory space occupied by tensors T_1 to T_4 is 12,544, 6,272, 6,272, and 6,272 KB, respectively. Based on the earlier analysis of memory overhead during the operator execution process, we divide the entire inference process into several execution stages and interval stages. The execution stage represents the process where an operator is actively performing computations. In contrast, the interval stage corresponds to the period when one operator has completed execution, and the execution of the next operator has not yet commenced. Memory overhead during execution and interval stages is denoted as M_e and M_i, respectively. The memory overhead analysis for all valid operator execution orders of the example model in Figure 2 is provided in Table 2. The values in the table round to the nearest whole integer.

Taking Order1 as an example, the inference process proceeds as follows: (1) Before the execution of the first operator, only the input tensor T_0 is present in memory, with a memory overhead of $M_{i1} = M(T_0) = 588$ KB; (2) during Conv1's execution, memory usage is $M_{e1} = M(T_0) + M(T_1) + M(Conv1_{kernel}) = 13,139$ KB; (3) before the execution of the second operator

Conv2, the intermediate result tensors to be stored in memory are T_0 and T_1, with a memory overhead of $M_{i2} = T_0 + T_1 = 588 + 12,544 = 13,132$ KB; (4) during the execution of the second operator Conv2, in addition to the memory space required for Conv2 computation, tensor T_0 needs to be additionally saved. The memory overhead is calculated as $M_{e2} = M(T_1) + M(T_2) + M(Conv2_{kernel}) + M(T_0) = 12,544 + 6,272 + 641,283 * 3/256 + 588 = 19,692$ KB; (5) before the execution of the third operator Conv3, the intermediate result tensors to be stored in memory are T_0 and T_2, with a memory overhead of $M_{i3} = M(T_0) + M(T_2) = 588 + 6,272 = 6,860$ KB; (6) during the execution of the third operator Conv3, in addition to the memory space required for Conv3 computation, tensor T_2 needs to be additionally saved. The memory overhead is calculated as $M_{e3} = M(T_0) + M(T_3) + M(Conv3_{kernel}) + T_2 = 588 + 6,272 + 31,283 * 3/256 + 6,272 = 13,146$ KB; (7) before the execution of the fourth operator Sum, the intermediate result tensors to be stored in memory are T_2 and T_3, with a memory overhead of $M_{i4} = M(T_2) + M(T_3) = 6,272 + 6,272 = 12,544$ KB; (8) during the execution of the fourth operator Sum, assuming an in-place addition method where the input and output tensors share the same memory space, the memory overhead is calculated as $M_{e4} = T_2 + T_3 = 6,272 + 6,272 = 12,544$ KB; and (9) after the completion of all operators' computations, the output tensor T_4 needs to be stored in memory, with a memory overhead of $M_{i5} = T_4 = 6,272$ KB.

In Order1, the maximum memory overhead is 19,692 KB. Order2 and Order3 are similar to Order1, and their maximum memory overhead is 25,376 KB. Hence, optimizing memory usage by adjusting the order of operator execution is crucial in limited memory scenarios. This minimizes overhead, increases memory space, and reduces computation time, especially for intensive tasks like partitioning operators. Efficient topology sorting becomes pivotal in managing memory overhead and enhancing model performance in constrained environments. Therefore, adjusting execution order impacts memory overhead, highlighting the importance of efficient topology sorting for improved performance. Even with similar maximum overhead for Order2 and Order3, differences in local memory overhead exist. Computation time improvement in inference tasks can involve sacrificing memory space via operator slicing. Additionally, the number, method, and ratio of sliced sub-operators cause computation time and additional memory overhead. Adjusting operator execution order under limited memory can reduce maximum memory overhead, increase available memory space, and reduce computation time through operator slicing.

3 Design

3.1 Overview

This section introduces MemoriaNova, a comprehensive approach designed to optimize DL models for edge devices. Within MemoriaNova, we present two core algorithms: BTSearch and GenEFlow. BTSearch focuses on exploring the computational graph of the target DL model to identify the optimal operator execution order, thereby expanding the search space for GenEFlow. Subsequently, based on hardware specifications and the determined execution order, GenEFlow utilizes the information acquired from BTSearch to optimize the model's parallel configuration. This process aims to minimize inference latency while adhering to the memory constraints of each device. Figure 4 provides an overview of our methodology, illustrating the seamless integration of BTSearch and GenEFlow to achieve enhanced DL performance.

3.2 BTSearch: A Backtracking Algorithm for Optimizing Model Operator Topological Sorting

To reduce the memory overhead from the sequence of operator executions, we bring up BTSearch. BTSearch is a graph state backtracking algorithm that aims to find an operator execution order that

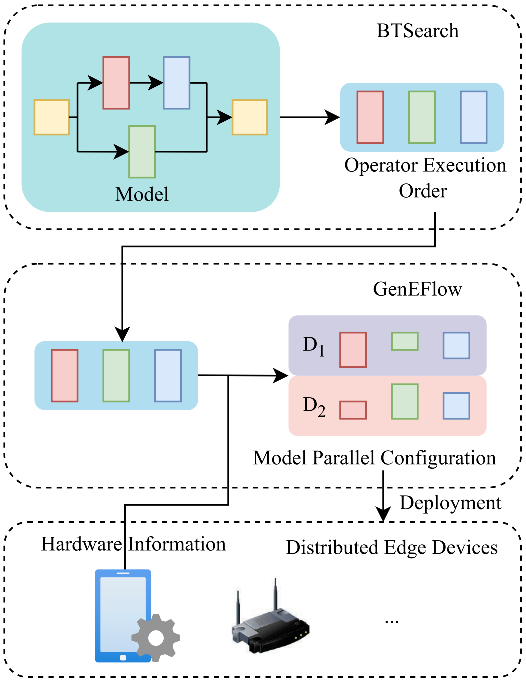


Fig. 4. System overview.

minimizes memory overhead and widens optimization opportunities for operator slicing efficiency gains.

The computational graph of a DL model can be represented as $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$, where $\mathcal{V}$ is vertices and $\mathcal{E}$ is edges. An edge $e_{ij} \in \mathcal{E}$ signifies a connection, implying op_i precedes op_j during inference. Sorting all operators ensures no path from op_j to op_i, termed topological sort. Computing sorts for a graph is a PC problem, typically requiring exponential time. In the worst-case scenario, it requires exponential time to traverse all topological sorts of a directed acyclic graph. Fortunately, multi-branch DL models typically exhibit a concatenated parallel structure, where the topological structure comprises several small-scale parallel structures. The fact results in a relatively smaller number of possible topological sorts. For ease of algorithm description, the following definitions are provided.

Definition 3.1 (Operator State). In the process of an inference task for a DL model, the state of an operator is defined as a Boolean variable, indicating whether the operator has completed its computation. For example, b_i denotes the state of the operator op_i.

Definition 3.2 (Computational Graph State). In the process of an inference task for a DL model, the states of all operators in the computational graph constitute the current state of the graph, denoted as $State_G = b_1, b_2, \ldots, b_N$ (where N is the number of operators in the computational graph).

We aim to optimize operator execution to maximize available memory during model inference, expanding efficiency optimization opportunities. The evaluation metric $Metric(Order_i)$ sums the memory overhead of each operator in a topological order: $Metric(Order_i) = \sum_{j=1}^{N}(Mem_{full} - Mem_e^{op_j})$. Smaller metric values signify better performance.

ALGORITHM 1: BTSearch

```
Data: DL model computation graph.
Result: Optimal order of operator execution for memory optimization.
1  Function Main():
       // Initialize the graph state and current local order. Initialize the state
          marking dictionary and the parsing function dictionary.
2      GraphState ← InitialState, CurrentOrder ← [], StateMark ← {}, ParseMark ← {};
3      MemMetric ← 0, BestMemMetric ← 0;
4      Recursive(GraphState, CurrentOrder);
5      return BestExecuteOrder;
6  Function Recursive(GraphState, CurrentOrder, MemMetric):
7      if All element in GraphState is true and $MemMetric > BestMemMetric$ then
8          Update BestMemMetric and ExecutionOrder;
9      end
       // Pruning.
10     if GraphState in StateMark and $MemMetric \leq StateMark[GraphState]$ then
11         return;
12     else
13         Update StateMark;
14     end
15     Executable, CurrentMem ← ParseState(GraphState);
16     foreach operator in Executable do
17         Update GraphState and CurrentOrder;
18         Recursive(GraphState, CurrentOrder, MemMetric + CurrentMem);
19         Downgrade GraphState and CurrentOrder;
20     end
```

The pseudocode for BTSearch is shown in Algorithm 1. BTSearch's input is the computation graph of a DL model, and its output is the optimal topological order under a certain metric condition. BTSearch perform a backtracking iteration on the graph that has not yet started computing. Based on the current state of the graph, all legal next states are derived and recursively processed in sequence. As the main steps, BTSearch initializes the graph state and the current local order first. And then, it calls the backtracking algorithm to obtain the optimal order.

BTSearch's backtracking recursive function first determines the recursion exit. If all operators have been executed, i.e., $State_G = true, true, \ldots, true$, then it is necessary to check whether the metric value of the currently found topological order is better. If so, then update the current best result. Then, the function returns. Parse the current graph state. Based on the current graph state, the status of each operator, the list of currently executable operators, and the tensor information stored in memory can be parsed. Loop through the current list of executable operators. For each operator in the list, assume the operator is chosen as the next to be executed, add it to the current local order list, and update the graph state. Recursively call the backtracking function with the parameters updated in the previous step. Finally, the regional order list and graph state were restored to the state before the last operator was chosen.

The graph state parsing function calculates the list of currently executable operators and the memory overhead based on the current graph state for all operators in the graph that still need to be executed loop through. Identify all directed edges that have the operator as the endpoint; for each such edge, increment the in-degree of that operator by one. Finally, Check the in-degree of all operators, adding operators with an in-degree of zero to the list of executable operators.

Additionally, the input tensors of all operators with an in-degree of zero are set as intermediate result tensors, and the memory overhead of all intermediate result tensors is calculated based on the current graph state.

3.2.1 Pruning Optimization Based on State Marking. During backtracking, repeated state transitions may lead to the same graph state. As the backtracking is depth-first, if a certain state recurs, then all subsequent iterations from that point have been processed, indicating subsequent local optimal solutions. The graph state updates with each recursive call, enabling the following optimization: Maintain a state marking dictionary outside the function to record encountered graph states and their local metric values. Before the loop, check if the state is recorded in the state-marking dictionary. If the state is recorded, then prune if the current metric exceeds the recorded value; otherwise, continue execution as usual. Finally, update the dictionary after the loop.

3.2.2 Hash Optimization for the Parsing Function. Even with previous optimization, redundant computations may occur during backtracking. Hash optimization eliminates redundant computations by recording graph states and parsing results in a dictionary. Check if parsing results exist in the dictionary; if found, return them; otherwise, compute and register the results.

3.2.3 Time Complexity Analysis. The algorithm has an exponential time complexity of $O(2^n)$ for general directed acyclic graphs. In practice, most deep learning models exhibit a topology characterized by a series-parallel graph. In such a graph, it is assumed that the structure consists of N parallel graphs concatenated, with each parallel graph containing M branches and each branch comprising K nodes. After pruning, each serial subgraph is processed only once. Best-case time complexity per subgraph is $O(M * K)$, while the worst-case is $O(K^M)$, and overall complexity is $O(N * M * K) \sim O(N * K^M)$. In practice, with limited branches and operators in serial subgraphs, the algorithm's execution time is acceptable.

3.3 GenEFlow: GA-based Model Parallel Scheduling Optimization Method

To decrease the inference latency of the target model by optimizing the model parallel schedule, we devise GenEFlow, a GA-based method, to optimize model parallel schedules to reduce inference latency. GenEFlow operates in a router-edge devices setup, considering broadcast and point-to-point communication. It abstracts model operator partition optimization as a chromosome configuration. Furthermore, GenEFlow constructs a search space, defines an objective function, and iteratively refines configurations using GAs. It ensures legal configurations and employs a GA Solution to minimize model inference latency in distributed systems.

3.3.1 Search Space Construction. We optimize the model's slicing configuration using a GA instead of optimizing operators individually. Operators execute synchronously, involving data transfer and computation stages. An operator's execution time is linearly related to its scale. By uniformly partitioning operators, parallel execution time decreases. Memory constraints guide the maximum splits per operator. Memory calculations inform optimal partitioning methods, detailed in Table 1. The computation of convolutions, typically large, benefits from efficient partitioning, reducing memory overhead.

Chromosome Encoding. Based on the current graph state, k_1 and k_2 are calculated from relevant parameters. If $k_1 < k_2$, then partitioning occurs along output channels (cout). Otherwise, it is along feature map height (fmh). Chromosome encoding for all operators' partitioning configurations is necessary to invoke GAs. For a single operator op_i, its partitioning encoding vector is defined as follows:

$$\vec{x_i} = [x_0, x_1, \ldots, x_n]. \tag{6}$$

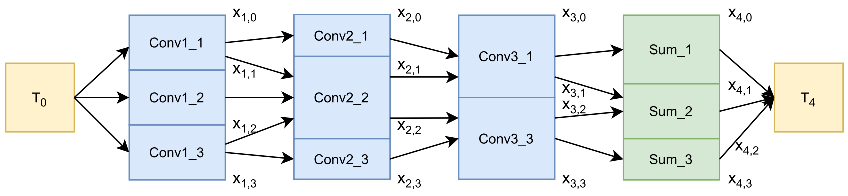


Fig. 5. Illustration of the relation between chromosome encoding and model partition configuration of the example model.

The encoding vector needs to satisfy the following constraints:

$$x_i \in \mathbb{N}, i \in [0, \ldots, n], \tag{7}$$

$$x_0 = 0, \tag{8}$$

$$x_n = length, \tag{9}$$

$$x_0 \leq x_1 \leq \ldots x_n, \tag{10}$$

where *length* is the size of the operator along the partitioning dimension, and n is the number of edge devices in the system. The partitioning encoding vector assigns tasks to devices based on output tensor indices, which are crucial for constraint calculations. Note that when $x_{i-1} = x_i$, it signifies that device d_i will not be assigned the computation task for the current operator. This characteristic is used in the subsequent calculation of constraint violation parameters.

From the partitioning vector of a single operator, where each partitioning operator corresponds to a single gene in the GA's chromosome representation, the chromosome encoding for the entire model's partitioning configuration can be obtained as

$$\vec{X} = [\vec{x_1}, \vec{x_2}, \ldots, \vec{x_N}]. \tag{11}$$

Through chromosome encoding analysis, we form a comprehensive search space. It fulfills distributed system memory needs and encompasses varied operator partitioning configurations. This space is denoted as a set as follows:

$$\{\vec{X}\} \text{ s.t. } (7),(8),(9),(10). \tag{12}$$

In Section 3.2, considering the example model, Figure 5 illustrates the relationship between chromosome encoding and model partitioning configuration. With three devices, operators execute in Order1: Conv1, Conv2, Conv3, Sum. In the figure, $x_{i,j}$ denotes the partitioning vector elements for the ith operator. The range $[x_{i,j-1}, x_{i,j})$ assigns computation to the jth device. If $x_{i,j-1} = x_{i,j}$, then it implies device j is not involved in operator i computation.

3.3.2 *Objective Function.* The GA adopted in GenEFlow is a single-objective optimization GA, and the optimization target is the single inference latency. Therefore, for any given valid chromosome encoding, it must be mapped to inference latency. This mapping is the optimization objective function.

Device Modeling Optimization and Communication. In our distributed edge device system, each of the n edge devices is linked via a router. Two communication methods are employed: point-to-point and broadcast. Point-to-point involves direct communication between two devices through the router. Broadcast sends data from one device, transmitting it to multiple devices through the router. This modeling mirrors real-world scenarios like interconnected smart home devices.

Chromosome Encoding to Inference Time Mapping. The algorithm calculates inference latency by processing operators sequentially in a deep-learning model. Its execution phase is divided

ALGORITHM 2: Optimization Objective Function

```
Data: DL model operator partitioning configuration vector $\vec{X} = [x_1, x_2 \ldots, x_N]$, model computation
      graph $\mathcal{G}$, and hardware information for distributed edge devices $\mathcal{D}$.
Result: Execution time of inference tasks under the current configuration
FinishTime← 0;
foreach $\vec{x_i} \in \vec{X}$ do
    TmpTime← 0;
    $Comm$ ←GetCommMem(i, $\vec{X}$, $\mathcal{G}$);
    foreach $d_k \in \mathcal{D}$ do
        CommTime← 0,CompTime← 0;
        CommNum← 0;
        foreach $op_j \in pred(op_i)$ do
            if Use Broadcast Mode then
                CommNum += $Comm[j][k]$;
                break;
            else
                CommNum += $Comm[j][k]$;
            end
        end
        CommTime←CommNum / $\mathcal{D}$.Bandwidth;
        CompTime← $Y_i(x_{i,k} - x_{i,k-1}, op_i)$;
        DeviceTime←CommTime + CompTime;
        if DeviceTime > TmpTime then
            $CommOpTime_i$ = CommTime;
        end
        TmpTime = max(TmpTime, DeviceTime);
    end
    FinishTime += TmpTime;
end
return FinishTime;
```

into data synchronization and computation phases. The predecessor operators of the operator op_i are defined as $pred(op_i)$. For any $op_j \in pred(op_i)$, there exists an edge e_{ji} in the computation graph $\mathcal{G}$. Similarly, the successor operators of the operator op_i are defined as $succ(op_i)$. For any $op_j \in succ(op_i)$, there exists an edge e_{ij} in the computation graph $\mathcal{G}$. During data synchronization, the algorithm determines the distribution of output tensors from predecessor operators to calculate data transfer amounts. The operator type and its partitioning method have an impact on communication mode (broadcast or point-to-point). This information is encapsulated in the chromosome $\vec{X}$ for inference latency calculation.

Calculation of Data Transfer Quantity. Algorithm 2 outlines the optimization objective function. The function GetCommMem($op_{ID}, \vec{X}$) computes the transfer parameter matrix $Comm$. $Comm[i][j]$ indicates the data amount transferred from the ith predecessor operator to device d_j. The function initializes $Comm$ with 0s and iterates over predecessor operators and devices, computing data transfer based on operator types and partitioning vectors $\vec{X}$.

Algorithm 3 computes the communication data volume for a given operator and device pair. For the predecessor operator op_j and device d_k of the current operator op_i, when $type(op_i) = Conv$, there are several cases as follows:

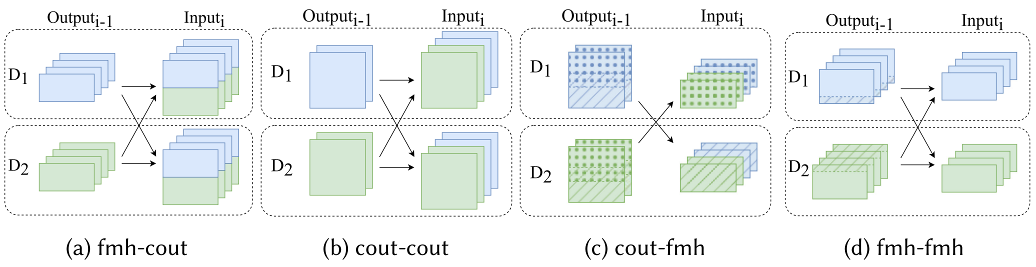


Fig. 6. Data communication of convolution operator with different partition methods. $Output_{i-1}$ is the output tensor of operator $i-1$, $Input_i$ is the input of convolution operator i. D_1 and D_2 are distributed devices.

ALGORITHM 3: GetCommMem: Function for Obtaining Communication Data Volume

```
Data: Operator ID, vector for partitioning configuration of DL model operators $\vec{X} = [x_1, x_2 \ldots, x_N]$,
      computation graph of the model $\mathcal{G}$, and hardware information for distributed edge devices $\mathcal{D}$.
Result: Communication Data Volume Matrix $Comm$
1 foreach $op_j \in pred(op_i)$ do
2     comm$\leftarrow [0] * n$;
3     foreach $d_k \in \mathcal{D}$ do
4         Calculate $M_{need}$ and $M_{hold}$ according to the $type(op_i)$ and the partition method of $op_i$ and $op_j$;
5         comm[di] += $M_{need} - M_{hold}$;
6     end
7     $Comm[j] \leftarrow$comm;
8 end
9 return $Comm$;
```

(i) If op_i adopts cout partitioning, then data from op_j is synchronized to all devices. The total transferred parameters amount to $M_{out}(op_j)$, incrementing all $Comm[j]$ elements.

(ii) If op_i uses fmh and op_j cout partitioning, then devices need partial feature map data. Data transfer is computed based on feature map indices, excluding portions saved on d_k. Transferred data amount: $M_{out}(op_j) \times C_{comm}/C^j_{full} \times (x^j_e - x^j_s)$.

(iii) If both op_i and op_j use fmh partitioning, then the data required by op_i on d_k as input and currently not held by the current device d_k needs to be transferred from other devices to d_k. As for the transferred data, it is calculated as $M_{out}(op_j) \times H_{comm}/H^j_{full}$, where $H_{comm} = \max(x^j_e - x^j_s, \max(0, x^j_e - x^j_k) + \max(0, x^j_{k-1} - x^j_s))$.

The communication volume for convolutional operators is depicted in Figure 6. Cases (a) and (b) represent Case (i), while (c) and (d) correspond to Cases (ii) and (iii). In (c) and (d), the characteristic of convolution determines that there may be duplicated data in $Input_i$, marked by shaded areas. For other scenarios, M_{need} in op_j's output and M_{hold} on d_k are computed. Data to be transferred are $M_{need} - M_{hold}$. Broadcast communication is used if data are required by multiple devices, considering a single transmission's data volume.

Communication Time and Computation Time. According Algorithm 2, the total data communication volume for op_i transmitted in device k is $CommNum+ = Comm[j][k]$, where op_j is the predecessor operator of op_i, and $Comm[j][k]$ indicates the data amount transferred from the ith predecessor operator to device d_j. Then, the total communication time of operator op_i is calculated as CommTime$\leftarrow$CommNum / $\mathcal{D}$.Bandwidth.

We assume that for a specific operator and device, the execution time is linearly related to the size of the input or output feature map. Therefore, a linear function Y_i can be used to calculate the computation time of operator op_i on device d_k as $CompTime = Y_i(x_{i,k} - x_{i,k-1}, op_i)$. Here, $x_{i,k} - x_{i,k-1}$ represents the part of the operator split on device d_i corresponding to the partitioned dimension.

Therefore, the time expense for this operator op_i on device k is $DeviceTime = CommTime + CompTime$, and the longest time spent on each device for operator op_i is the time cost $TmpTime_i$ of this operator. Summing up, all operators' time yields the total inference latency $FinishTime$. When calculating the time cost of each operator op_i, the communication time $CommOpTime_i$ generated by this segment is the communication time of that operator. Summing up the communication times of all operators gives GenEFlow the total communication time.

3.3.3 Constraint Violation Parameters. We utilize the high-performance GA library *Geatpy* [19] to implement the optimization iteration process. In the iteration process of *Geatpy*, The constraint conditions considered are Legitimacy of chromosome parameters; (ii) Legitimacy of the total memory in the distributed system; and (iii) Legitimacy of memory on each device in the distributed system.

Only the chromosomes (model partitioning configurations) that pass all three legitimacy checks are considered legal. Constraint violation parameters define the degree of violation for a specific constraint in the optimization problem. For example, assuming a constraint in the optimization problem is $a \le b$, the constraint violation parameter corresponding to this constraint is $a - b$. The larger this value, the higher the degree of constraint violation.

Legitimacy of Chromosome Parameters. For the chromosome $\vec{X} = [\vec{x_1}, \vec{x_2}, \dots, \vec{x_N}]$, where $\vec{x_i} = [x_{i,0}, x_{i,1}, \dots, x_{i,n}]$, it corresponds to the partitioning configuration of the ith operator in the execution sequence. The parameters in it need to satisfy the constraint conditions given by (7), (8), (9), and (10). The constraint condition (7) is ensured by specifying that the parameters within the chromosome are integers when defining the optimization problem, and there is no need to add it to the constraint violation parameters. Constraint condition (8) corresponds to two constraint violation parameters,

$$cv_{1,i} = x_{i,0},\ cv_{2,i} = -x_{i,0}. \tag{13}$$

Similarly, constraint condition (8) corresponds to two constraint violation parameters,

$$cv_{3,i} = x_{i,n} - length,\ cv_{4,i} = length - x_{i,n}, \tag{14}$$

where $length$ is the size of op_i in the corresponding partitioning dimension. Constraint condition (10) corresponds to n constraint violation parameters,

$$cv_{5,ij} = x_{i,j-1} - x_{i,j}, j \in [1, 2, \dots, n]. \tag{15}$$

Legitimacy of the Total Memory in the Distributed System. In Section 3.3.1, we discussed the impact of the total available memory in the distributed system on the upper limit of the number of partitions in the model partitioning configuration. Assuming the upper limit of the number of partitions for op_i is k_{max}, then op_i corresponds to a constraint violation parameter,

$$cv_{6,i} = n - \sum_{j=1}^{n} \mathcal{I}_j - min(n, k_{max}), i \in [1, 2, \dots, N], \tag{16}$$

where

$$\mathcal{I}_j = \begin{cases} 0 & x_{i,j-1} = x_{i,j-1} \\ 1 & x_{i,j-1} \neq x_{i,j-1} \end{cases}. \tag{17}$$

Legitimacy of Memory on Each Device in the Distributed System. During inference, each operator's execution on devices must adhere to device memory limits. Given the fixed execution order and result tensor storage on devices, memory consumption per operator on each device is derived from model partitioning configuration $\vec{X}$. Assume device memory limits as $M_{limits} = [M_{l1}, M_{l2}, \ldots, M_{ln}]$, where M_{li} denotes the ith device's available memory. For the operator op_i, it has a total of n constraint violation parameters on various devices,

$$cv_{7,ij} = M_{e,ij} + M_{o,ij} - M_{lj}, j \in [1, 2, \ldots n]. \tag{18}$$

The memory consumption during operator execution on device d_j is denoted as $M_{e,ij}$, and $M_{o,ij}$ represents the memory consumed by other tensors on d_j during op_i execution. Based on the analysis in Section 3.2, it can be inferred that the memory overhead of operators during execution is always greater than or equal to that of the intermediate stages. Therefore, it is only necessary to ensure that the execution phase complies with the memory constraints.

The vector representing the constraint violation parameters for a single operator is given by

$$\begin{aligned}\vec{cv}_i = [&cv_{1,i}, cv_{2,i}, cv_{3,i}, cv_{4,i}, cv_{5,i1},\\ &\ldots, cv_{5,in}, cv_{6,i}, cv_{7,i1}, \ldots, cv_{7,in}].\end{aligned} \tag{19}$$

The vector representing the constraint violation parameters for the entire model is as follows:

$$\vec{CV} = [cv_1, cv_2, \ldots, cv_N]. \tag{20}$$

3.3.4 GA Solving. The model parallel scheduling problem seeks to minimize inference latency by optimizing partition vectors for each operator in the distributed system. GAs are well suited for this nonlinear optimization task. However, conventional crossover operations may disrupt superior chromosomes, affecting overall performance. Therefore, we adopt a single-objective GA with an elite preservation strategy. This approach initializes a large population and computes fitness based on latency. A new population is generated through crossover and mutation operators, preserving privileged individuals. The process continues until convergence or a specified generation limit is reached, resulting in optimized model parallel scheduling.

4 Evaluation

This section mainly presents the experimental results and analysis of the previously mentioned methods, divided into six parts. In Section 4.1, we introduce the configurations and settings of both the simulated and real environments. In Section 4.2, we select multiple DNN models and **large language models (LLMs)** to evaluate the memory optimization effectiveness of BTSearch compared to other methods. In Section 4.3, we compare the inference latency optimization of GenEFlow with other methods under the same configuration. The experiments assess the model inference efficiency of these methods without considering memory constraints. In Section 4.4, we set different device memory limitations to validate the minimum memory requirements for model inference optimization and evaluate the optimization effects of various methods. In Section 4.5, we evaluate the inference latency of GenEFlow across multiple models by altering the number of devices and heterogeneous configurations, analyzing how these factors impact model inference latency. In Section 4.6, we compare the inference latency optimization of GenEFlow with other methods in a real environment.

4.1 Experimental Setup

Experiment Platforms. The parameters of the experimental platform and simulation configuration are shown in Table 3. Our experiments are conducted in two distinct environments. The first scenario is a simulated environment using a local PC (CPU*8 @2.5GHz, 32GB RAM) to mimic

Table 3. Hardware Information Used in the Simulation Environment and Simulation Configuration

Simulation Configuration		**Hardware Information**		
Parameter	Value	Hardware	Model Information	CFLOPS
Number of Devices	4	PC	CPU*8 @2.5GHz 32GB	0.24
Memory (MB)	[50, 50, 50, 50]	Jetson TX2	GPU*1 @1.12GHz, CPU*6 @1.4GHz 8GB	0.50
Bandwidth (Mbps)	2000	RPi4	CPU*4 @1.5GHz 4GB	0.80
CFLOPS	[1.0, 1.0, 0.8, 0.8]	RPi3	CPU*4 @1.2GHz 1GB	1.00

edge devices with varying performance levels by limiting the number of **CPU cores and floating-point computational performance (CFLOPS)**. In this setup, four simulated devices are configured with a communication bandwidth of 2,000 Mbps, each having 50 MB of memory, with CFLOPS values set to [1.0, 1.0, 0.8, 0.8]. The second scenario is a real environment where GenEFlow optimization experiments are performed on a platform consisting of one desktop PC, one Jetson TX2, one Raspberry Pi 3B (RPi3), and one Raspberry Pi 4B (RPi4). An SE109 (2.5 Gbps) is used for wired connections and configuration, with the communication bandwidth limited to 2000 Mbps.

Experiment Models. We select VGG13 [35], ResNet50 [13], InceptionV3 [37], MobileNetV3 [14], SqueezeNet [18], GoogLeNet [36], and RegNet [31] as the models. The models are pre-trained models sourced from PyTorch.hub. They are converted to the .onnx format using the *torch.onnx.export*() command from PyTorch. Moreover, we also evaluate our framework on three LLMs, BERT [7], GPT-2 [23], and Qwen2 [24]. For running CNN models, the input data shape is [1, 3, 224, 224], and for LLMs, the input data shape is [1, 128].

4.2 Memory Optimization Analysis during Inference Process

This experiment aims to validate the memory optimization method proposed in Section 3.2. The comparison of the method with different baselines is shown in Table 4.

The adopted baselines are as follows: (i) Random, which randomly selects an executable operator each time; (ii) PEFT [1], a heuristic algorithm optimizing for inference efficiency; and (iii) Greedy [21], which selects the operator with the largest input tensor to execute each time, aiming to minimize memory consumption as much as possible.

BTsearch consistently achieves optimal results across all models. All methods yield the same for VGG13 and GPT-2 with a single valid topological order. Similarly, models like MobileNetV3, SqueezeNet, and EfficientNet-50, despite having branching structures, result in identical outcomes due to simplified operators. However, ResNet-50, InceptionV3, GoogLeNet, BERT, and Qwen2 variations occur. PEFT optimizes execution time, favoring larger-scale operators early in the order. Greedy selects operators based on input tensor size, outperforming PEFT. BTSearch guarantees optimal results by exploring all legal topological orders. Compared to random selection, BTSearch achieves up to a 12% improvement. To illustrate BTSearch's efficacy, we use GoogLeNet to compare memory overheads under Random and BTSearch. As shown in Figure 8, while initial stages show minimal optimization due to fixed orders, subsequent multi-branch DAG structures benefit from optimized execution, reducing memory usage and expanding optimization possibilities for inference latency.

Table 4. Comparison of Cumulative Memory Overhead during the Execution Process of Each Operator (MB)

Model	Random	PEFT	Greedy	BTSearch
VGG13	194.17	194.17	194.17	194.17
ResNet50	395.35	394.97	390.37	390.37
InceptionV3	483.10	471.36	460.07	437.22
MobileNetV3	27.78	27.78	27.78	27.78
SqueezeNet	70.41	70.41	70.41	70.41
EfficientNet-b0	236.88	236.88	236.88	236.88
GoogLeNet	159.58	156.27	151.02	139.71
RegNet	694.39	698.51	695.92	692.86
GPT-2	1000.88	1000.88	1000.88	1000.88
BERT	703.91	701.91	673.03	646.03
Qwen2	20590.18	20481.18	20179.93	19224.75

Table 5. Comparison of Efficiency of Memory Optimization Methods

Model	Op Num	Random (ms)	PEFT (ms)	Greedy (ms)	BTSearch		
					Time (ms)	Pruned	Searched
ResNet50	71	1.03	2.02	2.03	5.98	20	20
InceptionV3	108	7.01	6.98	7.01	3435.12	1,387,509	1,529
GoogLeNet	71	2.99	2.99	2.99	1530.76	336,654	2,666
RegNet	94	3.00	2.99	3.44	4.99	28	3
GPT-2	159	39.41	39.47	39.37	50.84	0	1
BERT	173	23.48	23.04	22.49	130.57	11,021	46
Qwen2	283	293.99	302.92	302.83	2351.12	496,743,478	225

Next, we analyze the efficiency of the BTSearch algorithm. In models with multiple valid operator execution orders, compare the execution times of different methods. In addition, a comparison is made between the pruning frequency of the BTSearch algorithm and the total number of complete topological orderings searched. The comparative data are shown in Table 5.

From the table, Random, PEFT, and Greedy optimize memory quickly, with time complexity O(N) for N model operators. BTSearch, despite higher time complexity, completes optimization and reaches the millisecond level of 10^3 ms, which is acceptable for fixed hardware environments and single inference tasks. Because the BTSearch method aims to optimize memory consumption, GenEFlow is provided with a broader search space to support more complex models and computational tasks. The "Pruned" and "Searched" columns in BTSearch show pruned and total searched orderings, respectively. BTSearch efficiently prunes orders that do not meet requirements based on graph states. Pruning reduces the search space significantly, considering fewer complete orderings and is especially effective when executed before many DAG operators start. Due to the lack of complex branching structures in GPT-2, the number of pruned orderings by BTSearch is 0. For the BERT and Qwen2 models, due to their complexity and the large number of operators, BTSearch prunes and searches a more significant number of orderings, resulting in better optimization. This approach ensures BTSearch navigates a manageable number of orderings, enhancing efficiency for complex models like InceptionV3, GoogLeNet, BERT, and Qwen2.

4.3 Acceleration Optimization Analysis during Inference Process

The experiment evaluates the GenEFlow algorithm for model inference efficiency, excluding memory constraints. Inter-device bandwidth is limited to 2000 Mbps, and memory limits per device

Table 6. GA Search Space Upper Bound Calculation

Model	$\mathcal{D}$	$lg(K_{\mathbf{fmh}})$	$lg(K_{\mathbf{cout}})$	$lg(K_{\mathbf{len}})$	$lg(S)$
VGG13	4	142.0	74.9	0.0	216.9
ResNet50	4	351.0	576.0	0.0	927.0
InceptionV3	4	601.0	1100.0	0.0	1701.0
MobileNetV3	4	352.0	394.0	0.0	746.0
SqueezeNet	4	246.0	117.0	0.0	363.0
EfficientNet-b0	4	537.0	728.0	0.0	1270.0
GoogLeNet	4	647.9	409.76	0.0	1057.6
RegNet	4	256.0	647.0	0.0	904.0
GPT-2	4	0.0	0.0	2318.1	2318.1
BERT	4	0.0	0.0	2522.2	2522.2
Qwen2	4	0.0	0.0	4125.9	4125.9

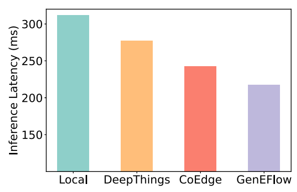
Here K_{fmh} represents $\prod_{i=1}^{N_{\text{fmh}}}(k_{\text{fmh}_i}+1)^{\mathcal{D}-1}$, K_{cout} represents $\prod_{j=1}^{N_{\text{cout}}}(k_{\text{cout}_j}+1)^{\mathcal{D}-1}$, and K_{len} represents $\prod_{l=1}^{N_{\text{len}}}(k_{\text{len}_l}+1)^{\mathcal{D}-1}$.

are set to 5000 MB, eliminating memory impact. GenEFlow parameters include a single-objective GA, elite preservation, 250,000 population size, 50 max iterations, 1e-6 convergence threshold, and 10 max convergence generations. These settings aim to optimize model partitioning for efficient distributed inference.

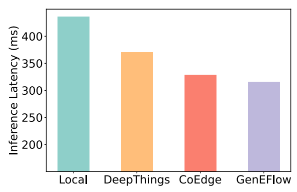
The GA search space upper bound S, as shown in Table 6, can be expressed as $S = \prod_{i}^{N_{\text{fmh}}}(k_{\text{fmh}_i}+1)^{\mathcal{D}-1} \times \prod_{j}^{N_{\text{cout}}}(k_{\text{cout}_j}+1)^{\mathcal{D}-1} \times \prod_{l}^{N_{\text{len}}}(k_{\text{len}_l}+1)^{\mathcal{D}-1}$, where k_{fmh_i} represents the output tensor size of the operators split by feature map height (fmh), k_{cout_j} represents the output channel size of the operators split by output channels (cout), k_{len_l} represents the tensor size of the operators split by output length (len), and $\mathcal{D}$ represents the number of distributed devices. Additionally, N_{fmh} represents the number of operators split by feature map height (fmh), N_{cout} represents the number of operators split by output channels (cout), and N_{len} represents the number of operators split by output length (len). The table shows that the GPT-2, BERT, and Qwen2 models have a large search space due to their higher number of operators (Op Number). Consequently, the upper bounds of the search space for these models are much higher compared to models like VGG13 and ResNet50.

The comparison in Figure 7 illustrates GenEFlow's superior inference latency without memory constraints. It outperforms CoEdge [46] by up to 33.9%. GenEFlow incurs minimal computational overhead and partitions each layer individually, enhancing its efficiency. In contrast, CoEdge optimizes layers individually, yielding inferior results holistically. Model size strongly correlates with inference latency. GenEFlow excels in optimizing complex models but produces similar results to CoEdge for smaller models like SqueezeNet. The slight dip in GenEFlow's performance for InceptionV3 may stem from longer chromosome encoding and inadequate population size, leading to local optima. The Efficient-b0 model, with minimal computational overhead, favors DeepThings, which achieves marginally better results than GenEFlow.

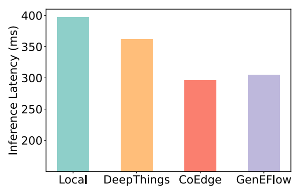
As shown in Figure 9, it compares the data transfer volume in the final operator scheduling obtained by the EfficientNet-b0 model under the GenEFlow and CoEdge methods. Compared to CoEdge, GenEFlow notably reduces communication by analyzing data transfer volumes, which is attributed to its holistic optimization objective encompassing computation and communication processes. The GA fosters offspring with lower latency, indirectly minimizing data communication during distributed inference.



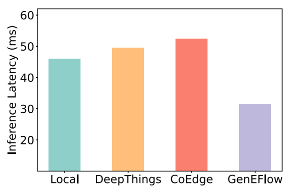
(a) VGG13



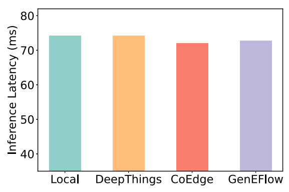
(b) ResNet50



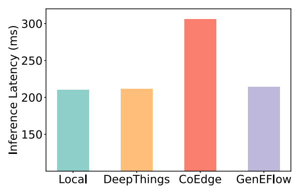
(c) InceptionV3



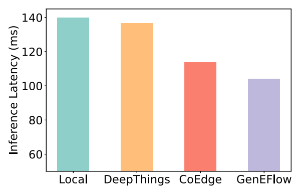
(d) MobileNetV3



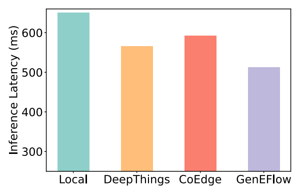
(e) SqueezeNet



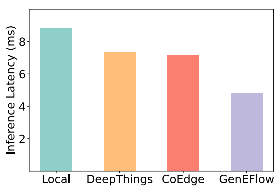
(f) EfficientNet-b0



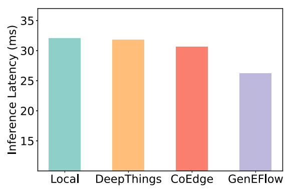
(g) GoogLeNet



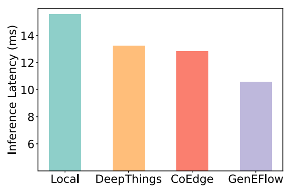
(h) RegNet



(i) BERT

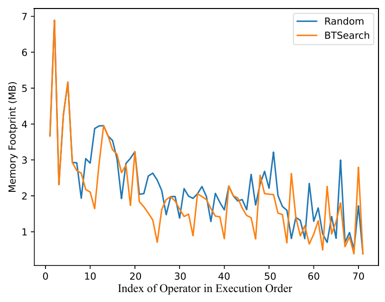


(j) GPT-2



(k) Qwen2

Fig. 7. Comparison of inference latency among different models.



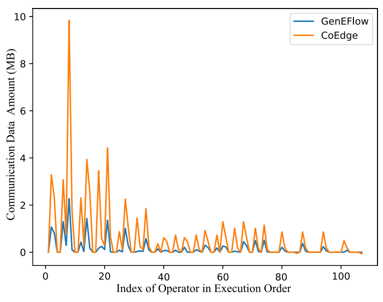


Fig. 8. Memory footprint trace of operators.

Fig. 9. Comparison of data communication.

Table 7. Comparison of Time Consumption of the Optimization Process (s)

Model	DeepThings	CoEdge	GenEFlow
VGG13	1.05	9.95	6371.59
ResNet50	1.28	9.97	20123.58
InceptionV3	2.26	9.97	21380.50
MobileNetV3	1.43	8.55	12351.91
SqueezeNet	1.02	7.65	12442.12
EfficientNet-b0	0.99	7.33	38615.78
GoogLeNet	2.43	10.50	18248.82
RegNet	1.90	9.93	25876.56
BERT	1.05	48.41	108210.81
GPT-2	1.99	50.03	131025.18
Qwen2	5.98	60.02	232352.55

Table 7 compares the optimization time for each method in this experiment. Except for GenEFlow, all methods optimize the operators sequentially, resulting in faster optimization speeds at the second level. In contrast, the GenEFlow algorithm takes significantly longer, ranging from 1.7 to 36.4 hours. This is mainly due to using a genetic algorithm, which involves a lot of computation. In this experiment, the number of distributed devices is fixed at 4, so the chromosome encoding length in the GenEFlow algorithm is proportional to the number of model operators. Therefore, models with a more significant number of operators require more time for population initialization and individual fitness evaluation within the population.

4.4 Optimization Effect Analysis under Memory Limitation Conditions

We aim to validate the optimization effects of different methods on model inference efficiency while considering memory constraints. We set various device memory limitations to verify whether the optimization methods meet the specified memory constraints. If the memory requirements are met, then the inference acceleration effects of each model under memory constraints are analyzed as shown in Table 8.

Prioritizing the adjustment of operator partitioning, GenEFlow optimizes inference memory overhead, facilitating efficient task execution even under stringent memory constraints. Memory thresholds are directly linked to the scale of model operators. CoEdge and GenEFlow minimize computational overhead, significantly reducing memory consumption compared to local and DeepThings' deployment methods. Tight memory constraints limit partitioning methods, reducing GenEFlow's search space and potential acceleration. GenEFlow adapts to varying memory constraints by considering device memory limits during GA application. Other methods lack memory consideration and remain fixed at specific thresholds, limiting their applicability and latency reduction even with increased memory availability.

4.5 Heterogeneous Device Scalability and Inference Latency Analysis

We evaluate GenEFlow's inference latency on VGG13, ResNet50, MobileNetV3, and EfficientNet-b0 models by changing the number of devices and heterogeneous device configurations. The communication bandwidth is 2000 MB/s, the memory limit for each device is 5000 MB, the number of distributed devices is four, and the CFLOPS of the devices for each device is set to 0.5. Figure 10 shows that as the number of devices increases, the inference latency of the models

Table 8. Comparison of Inference Latency of Different Models under Memory Constraints (ms)

Model	Memory	Local	DeepThings	CoEdge	GenEFlow	Model	Memory	Local	DeepThings	CoEdge	GenEFlow
VGG13	10MB	×	×	242.48	217.26	ResNet50	5MB	×	×	×	315.54
	20MB	×	×	-	-		10MB	×	370.83	329.04	-
	30MB	×	277.23	-	-		15MB	×	-	-	-
	40MB	311.80	-	-	-		20MB	435.98	-	-	-
InceptionV3	5MB	×	×	×	304.52	MobileNetV3	0.5MB	×	×	52.52	31.38
	7MB	×	×	296.12	301.60		1MB	×	×	-	-
	10MB	×	361.52	-	-		1.5MB	×	49.62	-	-
	15MB	397.52	-	-	-		2MB	46.10	-	-	-
SqueezeNet	2.5MB	×	×	72.08	72.78	EfficientNet-b0	5MB	×	×	305.96	218.38
	5MB	×	×	-	72.37		10MB	×	×	-	214.40
	7.5MB	×	74.18	-	-		15MB	×	211.80	-	-
	10MB	74.23	-	-	-		20MB	210.18	-	-	-
GoogLeNet	2MB	×	×	×	105.40	RegNet	10MB	×	×	×	512.33
	3MB	×	×	113.81	103.99		20MB	×	565.79	592.79	-
	5MB	×	136.63	-	-		30MB	×	-	-	-
	7MB	140.03	-	-	-		40MB	651.06	-	-	-
BERT	1MB	×	×	×	5.84	GPT-2	5MB	×	×	30.65	26.23
	1.5MB	×	×	×	-		10MB	×	×	-	-
	2MB	×	7.33	7.15	-		15MB	×	31.79	-	-
	2.5MB	8.83	-	-	-		18MB	32.06	-	-	-
Qwen2	10MB	×	×	×	10.59						
	20MB	×	×	12.82	-						
	30MB	×	13.26	-	-						
	40MB	13.59	-	-	-						

The memory limit is applied to each device. × indicates that the inference task cannot be completed under this memory limit. - indicates that increasing memory will not improve the optimization effect.

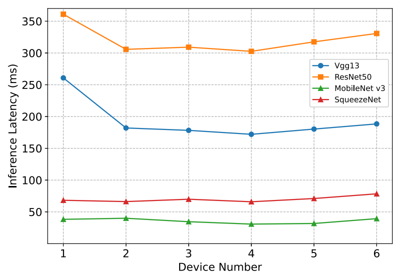


Fig. 10. Compare the inference with different numbers of devices.

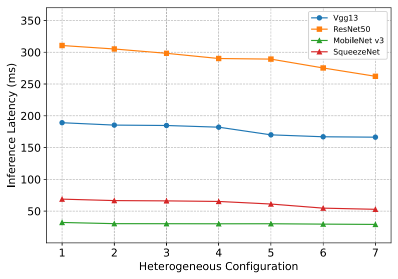


Fig. 11. Compare the inference with heterogeneous configuration.

first increases and then decreases, reaching the lowest latency when the number of distributed devices grows to four. When the number of distributed devices exceeds four, the inference latency gradually increases because the increase in communication time between devices outweighs the reduction in computation time due to distributed inference. We then fixed the number of devices to four and varied the CFLOPS values of each device to test the inference latency of models under heterogeneous device configurations. As shown in Figure 11. "Heterogeneous Configuration" refers to the CFLOPS settings of the four devices. Configurations 1 through 7 correspond to the following CFLOPS settings: 1 (0.8, 0.8, 0.8, 0.8), 2 (0.8, 0.8, 0.8, 0.5), 3 (0.8, 0.8, 0.5, 0.5), 4 (0.8, 0.5, 0.5, 0.3), 5 (0.5, 0.5, 0.5, 0.3), 6 (0.5, 0.5, 0.3, 0.3), and 7 (0.3, 0.3, 0.3, 0.3). As the CFLOPS values decrease, the overall inference time of the models tends to decrease, particularly for the VGG13 and ResNet50 models, where the reduction in latency is most significant in Configurations 6 and 7.

4.6 Analysis of Inference Acceleration on Heterogeneous Edge Devices

In a real environment, we compare the inference acceleration effects of different baselines on the models InceptionV3, ResNet50, Vgg19, SqueezeNet, and MobileNetV3. Each operator of these

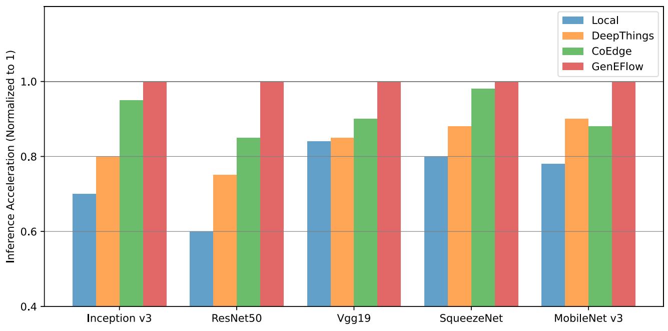


Fig. 12. Comparison of inference acceleration on heterogeneous edge devices.

models can be executed individually in all hardware configurations. The baseline methods include the following: (1) Local, where inference tasks are performed individually on each core and the average is taken as the result; (2) DeepThings, as described previously; and (3) CoEdge, as described previously. Figure 12 shows the inference acceleration effects under different hardware configurations, with the results normalized to GenEFlow. From the results, it can be seen that GeneFlow is able to achieve optimal inference latency optimization in most cases. Therefore, on heterogeneous edge devices, the GeneFlow method can significantly enhance the inference performance of the models.

5 Related Work

Optimizing DL models for deployment on edge devices. Deploying DL models on edge devices poses challenges due to limited resources. Lightweight models like MobileNets [15], Single Shot Detector [26], YOLO [32], and SqueezeNet [18] are designed for edge deployment, utilizing techniques such as filter decomposition and specialized convolution filters to reduce computations while maintaining accuracy. Model compression methods, including parameter quantization, pruning, and knowledge distillation, aim to minimize accuracy loss in existing models. DeepIoT [43] offer pruning methods for IoT devices, enabling immediate deployment on edge devices. Knowledge distillation trains smaller models to mimic larger ones, while Fast Exiting provides approximate classification results by utilizing initial layer computations. Techniques like AdaDeep [25], and DeepMon [17] combine compression methods to meet the accuracy and resource constraints. These methods aim to reduce model complexity for efficient inference on distributed edge devices while preserving computational integrity.

Distributed DL inference optimization. Deploying DL inference tasks across distributed systems involves optimizing efficiency through various strategies [47]. In the simplest approach, individual model operators are distributed across devices for sequential execution [3], enhancing throughput via pipeline formation. Guo et al.[12] adopt hierarchical optimization, employing a GA to vertically partition models and reduce pipeline latency.

In scenarios where devices execute tasks serially, pipeline design aims to enhance computational throughput [6]. Alternatively, parallel execution partitions models into sub-models deployed across devices, leveraging internal parallelism for improved resource utilization and reduced latency.

DeepThings [48], proposed by Zhao et al., adopt a classic model horizontal partitioning approach, dividing the model into independent sub-models to fully utilize each device's computational resources without inter-device data transmission overhead.

Zeng et al. introduce CoEdge [46], which partitions model operators without overlap to minimize computational overhead. CoEdge employs inter-device communication for overlapping input data, dividing each operator execution phase into data transfer and execution phases. However, while efficient, CoEdge's optimization process may lead to suboptimal solutions due to its greedy approach. EdgeFlow [16] extends the theoretical analysis for heterogeneous distributed edge devices by considering data transfer and computation phases during operator partitioning. It converts partitioning problems into linear programming and adopts a greedy approach to achieve local optimality. However, EdgeFlow may need to pay more attention to the impact of operator execution orders on performance, potentially leading to suboptimal solutions.

Optimization utilizing a model's directed acyclic graph structure. The optimization method utilizing DAG structures organizes tasks or dependencies into directed acyclic graphs to streamline computational processes efficiently. IOS [8] allowed parallel execution of operators within stages, employing a dynamic programming algorithm to find optimal execution schedules. However, its coarse optimization granularity limits scalability to distributed devices. HMCOS [40] optimized memory usage by simplifying DAG structures through a hierarchical perspective, reducing memory overhead during inference. AGO [42] partitioned computation graphs into subgraphs, optimizing operator execution efficiency for specific convolution operators. While effective, it is limited to certain convolution types and complex DAG handling. PEFT [1] scheduled DAG tasks onto heterogeneous devices, considering earliest start times and device completion times. However, it must address task division possibilities and may not fully optimize memory usage. Applying PEFT directly to DL models' DAG structure may underutilize device resources and provide suboptimal memory optimization results.

6 Conclusion and Future Work

We propose MemoriaNova, a framework that includes two innovative algorithms, BTSearch and GenEFlow, for optimizing memory and inference latency in distributed deep learning on edge devices. The BTSearch method optimizes the cumulative memory overhead of models structured as DAGs. Through meticulous exploration of the operator execution order, BTSearch effectively minimizes memory usage during model inference. This application significantly enhances memory efficiency and enlarges the latency optimization search space. Our experimental results demonstrate that BTSearch achieves up to a remarkable 12% reduction in memory overhead. GenEFlow targets the optimization of communication latency in distributed inference tasks from a holistic model perspective. It strategically configures operator placements by leveraging GAs to minimize communication delays across distributed edge devices and offering a comprehensive search space for model partitioning. Our empirical evaluations indicate that GenEFlow achieves impressive results, with a 33.9% reduction in inference latency. With the popularity of large language models, our future work will consider how to deploy large language models with higher memory requirements in memory-constrained edge devices and optimize their inference performance.

References

[1] Hamid Arabnejad and Jorge G. Barbosa. 2014. List scheduling algorithm for heterogeneous systems by an optimistic cost table. *IEEE Trans. Parallel Distrib. Syst.* 25, 3 (2014), 682–694.

[2] Graham Brightwell and Peter Winkler. 1991. Counting linear extensions. *Order* 8, 3 (1991), 225–242.

[3] Zinuo Cai, Zebin Chen, Zihan Liu, Quanmin Xie, Ruhui Ma, and Haibing Guan. 2024. RIDIC: Real-time intelligent transportation system with dispersed computing. *IEEE Trans. Intell. Transport. Syst.* 25, 1 (2024), 1013–1022. DOI: https://doi.org/10.1109/TITS.2023.3303877

[4] Zinuo Cai, Zebin Chen, Ruhui Ma, and Haibing Guan. 2023. SMSS: Stateful model serving in metaverse with serverless computing and GPU sharing. *IEEE J. Select. Areas. Commun.* 42, 3 (December 2023), 799–811. DOI: https://doi.org/10.1109/JSAC.2023.3345401

[5] Antonio Carlos Cob-Parro, Cristina Losada-Gutiérrez, Marta Marrón-Romera, Alfredo Gardel-Vicente, and Ignacio Bravo-Muñoz. 2021. Smart video surveillance system based on edge computing. *Sensors* 21, 9 (2021).

[6] Jacqueline M. Cole. 2020. A design-to-device pipeline for data-driven materials discovery. *Accounts Chem. Res.* 53, 3 (2020), 599–610.

[7] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. Bert: Pre-training of deep bidirectional transformers for language understanding. *CoRR* abs/1810.04805, (2018). Retrieved from https://arxiv.org/abs/1810.04805

[8] Yaoyao Ding, Ligeng Zhu, Zhihao Jia, Gennady Pekhimenko, and Song Han. 2021. Ios: Inter-operator scheduler for cnn acceleration. *Proc. Mach. Learn. Syst.* 3 (2021), 167–180.

[9] Khasim Vali Dudekula, Hussain Syed, Mohamed Iqbal Mahaboob Basha, Sudhakar Ilango Swamykan, Purna Prakash Kasaraneni, Yellapragada Venkata Pavan Kumar, Aymen Flah, and Ahmad Taher Azar. 2023. Convolutional neural network-based personalized program recommendation system for smart television users. *Sustainability* 15, 3 (2023), 2206.

[10] Mohammad Goudarzi, Marimuthu Palaniswami, and Rajkumar Buyya. 2022. Scheduling IoT applications in edge and fog computing environments: A taxonomy and future directions. *Comput. Surv.* 55, 7 (2022), 1–41.

[11] Jalalu Guntur, S. Srinivasulu Raju, T. Niranjan, Sai Kiran Kilaru, Rakesh Dronavalli, and N. Surya Seshu Kumar. 2023. IoT-Enhanced smart door locking system with security. *SN Comput. Sci.* 4, 2 (2023), 209.

[12] Xiaotian Guo, Andy D. Pimentel, and Todor Stefanov. 2023. Hierarchical design space exploration for distributed CNN inference at the edge. In *Machine Learning and Principles and Practice of Knowledge Discovery in Databases*. Springer Nature Switzerland, Cham, 545–556.

[13] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep residual learning for image recognition. *CoRR* abs/1512.03385, (2015). Retrieved from http://arxiv.org/abs/1512.03385

[14] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, Quoc V. Le, and Hartwig Adam. 2019. Searching for MobileNetV3. *CoRR* abs/1905.02244, (2019). Retrieved from http://arxiv.org/abs/1905.02244

[15] Andrew G. Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. 2017. MobileNets: Efficient convolutional neural networks for mobile vision applications. *CoRR* abs/1704.04861, (2017). Retrieved from http://arxiv.org/abs/1704.04861

[16] Chenghao Hu and Baochun Li. 2022. Distributed inference with deep learning models across heterogeneous edge devices. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'22)*. 330–339.

[17] Loc N. Huynh, Youngki Lee, and Rajesh Krishna Balan. 2017. Deepmon: Mobile GPU-based deep learning framework for continuous vision applications. In *Proceedings of the 15th Annual International Conference on Mobile Systems, Applications, and Services*. 82–95.

[18] Forrest N. Iandola, Matthew W. Moskewicz, Khalid Ashraf, Song Han, William J. Dally, and Kurt Keutzer. 2016. SqueezeNet: AlexNet-level accuracy with 50x fewer parameters and <1MB model size. *CoRR* abs/1602.07360, (2016). Retrieved from http://arxiv.org/abs/1602.07360

[19] Jazzbin et al. 2020. Geatpy: The genetic and evolutionary algorithm toolbox with high performance in Python.

[20] Amanda Jayanetti, Saman Halgamuge, and Rajkumar Buyya. 2024. Multi-agent deep reinforcement learning framework for renewable energy-aware workflow scheduling on distributed cloud data centers. *IEEE Trans. Parallel Distrib. Syst.* (April 2024), 1–12. DOI: https://doi.org/10.1109/TPDS.2024.3360448

[21] Dieter Jungnickel. 2013. *The Greedy Algorithm*. Springer, Berlin, 135–161.

[22] Yassin Kortli, Maher Jridi, Ayman Al Falou, and Mohamed Atri. 2020. Face recognition systems: A survey. *Sensors* 20, 2 (2020).

[23] Jieh-Sheng Lee and Jieh Hsiang. 2019. Patent claim generation by fine-tuning OpenAI GPT-2. *CoRR* abs/1907.02052, (2019). Retrieved from http://arxiv.org/abs/1907.02052

[24] Zehan Li, Xin Zhang, Yanzhao Zhang, Dingkun Long, Pengjun Xie, and Meishan Zhang. 2023. Towards general text embeddings with multi-stage contrastive learning. Retrieved from https://arxiv.org/abs/2308.03281

[25] Sicong Liu, Yingyan Lin, Zimu Zhou, Kaiming Nan, Hui Liu, and Junzhao Du. 2018. On-demand deep model compression for mobile devices: A usage-driven model selection framework. In *Proceedings of the 16th Annual International Conference on Mobile Systems, Applications, and Services (MobiSys '18)*. Association for Computing Machinery, Munich, Germany, 389–400. DOI: https://doi.org/10.1145/3210240.3210337

[26] Wei Liu, Dragomir Anguelov, Dumitru Erhan, Christian Szegedy, Scott Reed, Cheng-Yang Fu, and Alexander C. Berg. 2016. SSD: Single shot multibox detector. In *Proceedings of the European Conference on Computer Vision (ECCV'16)*, Bastian Leibe, Jiri Matas, Nicu Sebe, and Max Welling (Eds.). Springer International, Cham, 21–37.

[27] Yura Malitsky and Matthew K. Tam. 2023. Resolvent splitting for sums of monotone operators with minimal lifting. *Math. Program.* 201, 1 (2023), 231–262.

[28] Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. 2020. Distributed inference acceleration with adaptive DNN partitioning and offloading. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'20)*. IEEE, 854–863.

[29] Xiaonan Nie, Xupeng Miao, Zhi Yang, and Bin Cui. 2022. Tsplit: Fine-grained gpu memory management for efficient dnn training via tensor splitting. In *Proceedings of the IEEE 38th International Conference on Data Engineering (ICDE'22)*. IEEE, 2615–2628.

[30] Jeongeun Park, Donguk Yang, and Ha Young Kim. 2023. Text mining-based four-step framework for smart speaker product improvement and sales planning. *J. Retail. Consum. Serv.* 71 (2023), 103186.

[31] Ilija Radosavovic, Raj Prateek Kosaraju, Ross Girshick, Kaiming He, and Piotr Dollar. 2020. Designing network design spaces. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition (CVPR'20)*.

[32] Joseph Redmon and Ali Farhadi. 2017. YOLO9000: Better, faster, stronger. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'17)*.

[33] Wei-Qing Ren, Yu-Ben Qu, Chao Dong, Yu-Qian Jing, Hao Sun, Qi-Hui Wu, and Song Guo. 2023. A survey on collaborative DNN inference for edge intelligence. *Mach. Intell. Res.* 20, 3 (2023), 370–395.

[34] 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. Select. Areas Commun.* 41, 7(2023), 2041–2056. DOI: https://doi.org/10.1109/JSAC.2023.3280970

[35] Karen Simonyan and Andrew Zisserman. 2015. Very deep convolutional networks for large-scale image recognition. Retrieved from https://arxiv.org/abs/1409.1556

[36] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. 2015. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'15)*.

[37] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jon Shlens, and Zbigniew Wojna. 2016. Rethinking the inception architecture for computer vision. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR'16)*.

[38] Athanasios Voulodimos, Nikolaos Doulamis, Anastasios Doulamis, Eftychios Protopapadakis, and Diego Andina. 2018. Deep learning for computer vision: A brief review. *Intell. Neurosci.* (January 2018). DOI: https://doi.org/10.1155/2018/7068349

[39] Zhiyu Wang, Mohammad Goudarzi, Mingming Gong, and Rajkumar Buyya. 2024. Deep reinforcement learning-based scheduling for optimizing system load and response time in edge and fog computing environments. *Future Gener. Comput. Syst.* 152 (2024), 55–69.

[40] Zihan Wang, Chengcheng Wan, Yuting Chen, Ziyi Lin, He Jiang, and Lei Qiao. 2022. Hierarchical memory-constrained operator scheduling of neural architecture search networks. In *Proceedings of the 59th ACM/IEEE Design Automation Conference (DAC'22)*. Association for Computing Machinery, New York, NY, 493–498.

[41] Yuanjia Xu, Heng Wu, Wenbo Zhang, and Yi Hu. 2022. EOP: Efficient operator partition for deep learning inference over edge servers. In *Proceedings of the 18th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'22)*. Association for Computing Machinery, 45–57.

[42] Zhiying Xu, Hongding Peng, and Wei Wang. 2023. AGO: Boosting mobile AI inference performance by removing constraints on graph optimization. In *Proceedings of the IEEE Conference on Computer Communications (INFOCOM'23)*. 1–10.

[43] Shuochao Yao, Yiran Zhao, Aston Zhang, Lu Su, and Tarek F. Abdelzaher. 2017. Compressing deep neural network structures for sensing systems with a compressor-critic framework. *CoRR* abs/1706.01215, (2017). Retrieved from http://arxiv.org/abs/1706.01215

[44] Abbas Yazdinejad, Behrouz Zolfaghari, Ali Dehghantanha, Hadis Karimipour, Gautam Srivastava, and Reza M Parizi. 2023. Accurate threat hunting in industrial internet of things edge devices. *Digit. Commun. Netw.* 9, 5 (2023), 1123–1130.

[45] Chuanlong Yin, Yuefei Zhu, Jinlong Fei, and Xinzheng He. 2017. A deep learning approach for intrusion detection using recurrent neural networks. *IEEE Access* 5 (2017), 21954–21961.

[46] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2021. CoEdge: Cooperative DNN inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Trans. Netw.* 29, 2 (2021), 595–608.

[47] Rui Zhang, Xuesen Chu, Ruhui Ma, Meng Zhang, Liwei Lin, Honghao Gao, and Haibing Guan. 2022. OSTTD: Offloading of splittable tasks with topological dependence in multi-tier computing networks. *IEEE J. Select. Areas Commun.* 41, 2 (2022), 555–568.

[48] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. DeepThings: Distributed adaptive deep learning inference on resource-constrained IoT edge clusters. *IEEE Trans. Comput.-Aid. Des. Integr. Circ. Syst.* 37, 11 (2018), 2348–2359.

Received 1 March 2024; revised 10 August 2024; accepted 19 September 2024