FlexPie: Accelerate Distributed Inference on Edge Devices with Flexible Combinatorial Optimization

Runhua Zhang*, Hongxu Jiang✉*, Jinkun Geng+, Yuhang Ma*, Chenhui Zhu*, Haojie Wang†,
*Beihang University, +Stanford University,†Tsinghua University
{rhzhang20, jianghx}@buaa.edu.cn, gjk1994@stanford.edu,
{buaa_mayuhang, zhuch0401}@buaa.edu.cn, wanghaojie@tsinghua.edu.cn

Abstract. The rapid advancement of deep learning has catalyzed the development of novel IoT applications, which often deploy pre-trained deep neural network (DNN) models across multiple edge devices (typically 4∼6) for collaborative inference. However, conventional deployment methods frequently result in suboptimal inference times. Our study identifies that the inefficiency primarily stems from the model partitioning strategy employed by inference engines. Previous frameworks often rely on fixed partitioning schemes (e.g., one-dimensional or 2D-grid partitions) or limited optimizations (e.g., layer-wise adjustments or simple layer fusion), which fail to deliver optimal performance across varying model layers and testbed configurations. In this paper, we propose `FlexPie`, a solution to accelerate distributed inference on edge devices through *flexible combinatorial optimization.* `FlexPie` integrates an automated optimization procedure based on a data-driven cost model and dynamic programming, which efficiently finds an optimized model partition scheme in huge combinatorial spaces. Our evaluation on four commonly used DNN benchmarks demonstrates that `FlexPie` reduces inference time, achieving up to a 2.39× speedup over state-of-the-art methods.

Keywords: Edge Device, Distributed Inference, Model Partition, Combinatorial Optimization, Data-Driven, Dynamic Programming

1 Introduction

The past decade has witnessed a rapid development of deep learning, leading to a variety of novel applications in IoT (Internet of Things) scenarios, including image processing [3,14], video analysis [19], wearables [4] and so on. Due to network congestion and long physical distances, cloud services cannot meet low latency requirements [18]. Therefore, typical IoT applications tend to train the ML model in beefy clusters (e.g., cloud VMs or bare-metal machines), but the inference computation is usually offloaded to the edge device for the sake of real-time responsiveness [30,6] and data privacy [9].

While edge-based inference offers more timely results than its cloud-based counterpart, it also leads to unique challenges. Because of the growing size of

DNN models, deploying a single-edge device to load and process an entire model has become impractical. Consequently, distributed inference is required to enable collaborative execution of inference tasks. Today's edge applications [28,29,21,8] usually employ 4∼6 nodes[1] to jointly execute the inference for efficiency. Commonly, DNN feature maps are partitioned into different edge devices. Each edge device hosts a partial model and exchanges necessary parameters with others throughout the layer-based inference process.

Reviewing the existing distributed inference solutions, we can divide them into two categories. One line of prior works [29,30,20] adopt a fixed partition scheme, regardless of model shapes and testbed settings (e.g., the number of edge devices to run the inference). For instance, DeepSlicing [28] and MoDNN [20] adopt the One-dim partition scheme (i.e., partition the model by InW or InH or OutC dimension) whereas DeepThings [29] uses the 2D-grid partition scheme to run the distributed inference task. However, we consider the flexible model partition as a necessity due to two observations in our measurement study (details in §2.2): (1) When running inference computation on the same group of edge devices, different DNN layers yield their optimal inference time with different model partition schemes; (2) when running the inference computation for the same layer of DNN, the varying number of edge devices leads to different optimal model partition schemes, i.e., the optimal partition scheme obtained from one testbed will no longer be the optimal after we switch to another testbed.

The other line of works [6,8,21,30] introduce limited flexibility into the inference framework. For instance, DINA [21] and PartialDI [8] adopt layerwise optimization and allow the framework to use different partition schemes for every model layer. However, it ignores the inter-layer dependency, which can be further optimized with layer fusion. On the other hand, EdgeCI [6] and AOFL [30] explore the opportunities for layer fusion, but its fusion optimization is only applicable for a single partition scheme. While combining both layerwise optimization and fusion-based optimization seems straightforward, such a combination usually leads to a very large search space in practical deployment. The simple exhaustive search can be time-consuming and requires much expertise, which we believe is the main reason that discourages previous work from combining both *flexible layerwise partition* and *opportunistic layer fusion*.

Our goal. We develop `FlexPie`, which targets the scenarios of edge-based inference that employ multiple (4∼6) devices to jointly compute the inference results. Such scenarios have become prevalent in many practical applications [1]. `FlexPie` fully considers the optimization opportunities of layerwise optimization and inter-layer fusion to generate the desirable model partition scheme. In order to efficiently find the optimal scheme in a large design space, `FlexPie` incorporates an automatic model partition strategy, named *flexible combinatorial optimization* (FCO). In general, FCO runs the dynamic programming process and data-driven cost model to rapidly pick out the best model partition scheme

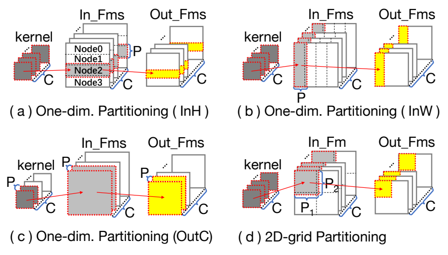
[1] Due to the practical constraints such as energy consumption, today's edge applications usually conduct small-scale (4∼6 nodes) distributed inference, which is different from the cloud-based distributed computing that employs 10s–100s nodes.

to deploy for a given model and testbed. To implement FCO, `FlexPie` consists of two main components.

Data-Driven Cost estimator (CE, §3.2). `FlexPie` employs Gradient Boosting Decision Tree (GBDT) models to estimate the inference time cost for the model layers. We implement the GBDT based on XGBoost [5] and collect data traces for edge-based inference under various settings, and then we use these data traces to train two GBDT models (estimators), namely i-Estimator and s-Estimator. The estimators take as input the DNN layer's metadata (including the height, width, and number of input/output channels, etc.) and the testbed information (including the bandwidth, communication topology, etc.) and then output an estimated time cost. The i-estimator estimates the time cost to complete the inference computation on this model layer; the s-estimator estimates the time cost for all the nodes in the testbed to complete the synchronization of the model layer. Estimated time costs provide useful guide information for `FlexPie` to make choices among partition schemes to minimize the overall time cost (we design a dynamic programming algorithm to achieve this).

Dynamic partition planner (DPP, §3.3). Driven by the i-Estimators and s-Estimator, we continue to design and implement a dynamic programming (DP) algorithm to decide the partition scheme for a given model and testbed. Taking the typical DNN model as an example, the DPP starts the DP process from the last layer of the DNN model. By contacting the i-Estimator, DPP knows the estimated inference time cost when it decides to adopt a specific partition scheme (e.g., InW-based, InH-based, OutC-based, 2D-grid, etc.) for one model layer of some specific shape. By contacting the s-Estimator, DPP knows the estimated cost to synchronize one model layer of some specific shape among the nodes of the cluster. Based on the i-Estimator and the s-Estimator, DPP can search and evaluate a series of partition schemes in the DP search space. DPP will keep the promising candidate partitions which can lead to the lowest overall inference time and keep traversing forwards, and finally output a desirable partition scheme. Besides, during the DP process, we have also incorporated the pruning strategy to reduce the search space and find the optimal partition scheme (i.e., the scheme yielding the lowest overall inference time) more efficiently.

Evaluation. We compare `FlexPie` with five partition schemes, i.e., InW/InH-based, OutC-based, 2D-grid, layerwise and fused-layer partition. We conduct inference with four models under different testbed settings. We find that, (1) The 2D-grid partition outperforms One-dim partition (InH/InW-based and OutC-based) when we conduct the distributed inference on the 4-node testbed. However, the InH/InW-based partition outperforms the 2D-grid and OutC-based partition when we switch to the 3-node testbed. Running on different testbeds, these typical partition schemes can all lead to an imbalance of model partition, and thus are not generally applicable to performance improvement. (2) Both layerwise optimization and data fusion can gain better inference speedup than the fixed partition scheme. However, simply using either strategy fails to yield optimal performance. By comparison, `FlexPie` combines both advantages and can adaptively choose the optimal partition scheme for different models under dif-



In_Fms
Out_Fms
kernel
Node0
Node1
Node2
Node3
P
C
(a) One-dim. Partitioning (InH)
(b) One-dim. Partitioning (InW)
(c) One-dim. Partitioning (OutC)
(d) 2D-grid Partitioning
In_Fm
P1
P2

Fig. 1: Example of parallelizing depth-wise separable convolution

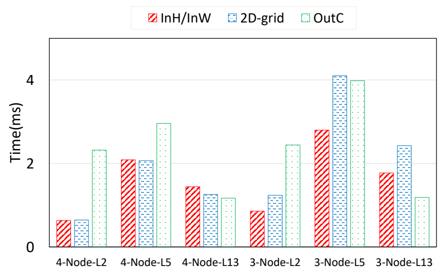


Fig. 2: Micro-bench test

ferent testbed settings. Therefore, `FlexPie` outperforms all the baselines across multiple benchmarks with a speedup of 1.10-2.39×.

2 Background and Motivation

2.1 Distributed Inference and Model Partition

Figure 1 illustrates the commonly used partition schemes in practice. Typically in a DNN model, there are two main categories of partition schemes, namely One-dim partition (Figure 1(a)-1(c)) and 2D-grid partition(Figure 1(d)). In One-dim partition, there are three commonly-used sub-categories, i.e., InH-based partition, InW-based partition and OutC-based partition, which indicate that the partition is conducted according to the height of the input feature map (InH), the width of the input feature map (InW), and the channel of the output feature map (OutC). As for the 2D-grid partition, it partitions the model on both the InH dimension and the InW dimension for the sake of load balance. Notably, the InC-based partition is not typically used to partition the model, because it introduces costly gather operations and leads to undesirable performance.

2.2 Partition Scheme Affects Performance

While One-dim and 2D-grid partition schemes are both generally applicable to inference models, we find that they lead to distinctly different completion times when we conduct the inference with different layers under different testbed settings. Figure 2 demonstrates this using a series of micro-bench tests. We conduct inference on MobileNet with different partition schemes and measure the completion inference time on different layers. In the first 3 groups of tests, i.e., 4-Node-L2, 4-Node-L5, 4-Node-L13, we run the inference with 4 edge devices, connected with SRIO at a bandwidth of 5Gb/s. The three groups of tests conduct inference with different layers, i.e., the 2nd layer (L2), the 5th layer (L5) and the 13th layer (L13). Obviously, different layers yield their optimal time with different partition schemes. L2 and L5 prefer the InH/InW-based partition and the 2D-grid partition, whereas L13 prefer the OutC-based partition. Next, we change the testbed settings from using 4 nodes to using 3 nodes, then we get the other three groups of tests, i.e., 3-Node-L2, 3-Node-L5 and 3-Node-L13. Comparing with the first 3 groups we can see that, even towards the same layer, the optimal partition schemes vary under different testbed settings. For example, L5 can yield an optimal inference time with 2D-grid partition in the 4-node setting, but the 2D-grid partition gives the worst inference time for L5 under the 3-node setting.

From the micro-bench tests, we realize that there are no *one-size-fits-all* partition schemes. Even in the same model, the different layers may prefer different partition schemes. Besides, the optimality of the partition schemes can also vary across different testbed settings. Therefore, we are motivated to design flexible partition schemes for different model layers under various testbed settings.

2.3 Trade-off between Computation and Communication

One-dim and 2D-grid partitions lead to different overlaps between computation (on each edge device) and communication (across multiple edge devices). Figure 1(a) illustrates the One-dim partition scheme (InH-based) when we conduct distributed inference with 4 nodes (Node0-Node3). The inference engine first uses the kernel and input feature maps (In_Fms) to inference, computing output feature maps (Out_Fms). Then the Out_Fms become the In_Fms for the next layer. From Figure 1(a) we can see that Node2 still requires some boundary data to be transferred from other nodes (e.g. Node1 and Node3) to perform a new layer inference.

Instead of fetching the data from other nodes, which incurs more communication overheads, an alternative way is to let each node conduct redundant inference computation in the previous layer. In Figure 1(a), when Node2 is conducting inference, we let it take not only the input from the dashed black rectangle but also the input from the dashed red rectangle. Therefore, Node2 is conducting more computation workload (i.e., Node2 does duplicate computation as Node1 and Node3), but it saves the communication overheads because the required In_Fms can be obtained from the local node in the next layer.

The redundant computation may not always be beneficial. For instance, when some model layers have large sizes and the inter-node bandwidth is high, trading computation overheads for communication efficiency does not help accelerate the inference. Therefore, given a specific testbed, `FlexPie` should *flexibly* decide whether or not to conduct redundant computation for each model layer (§3).

Motivated by the existing drawbacks, we develop `FlexPie` to (1) support flexible model partition for each layer, (2) conduct inter-layer optimization to yield better trade-off between computation and communication, and (3) automate the end-to-end optimization workflow to avoid human tuning effort and efficiently generate the optimal deployment scheme from a huge search space.

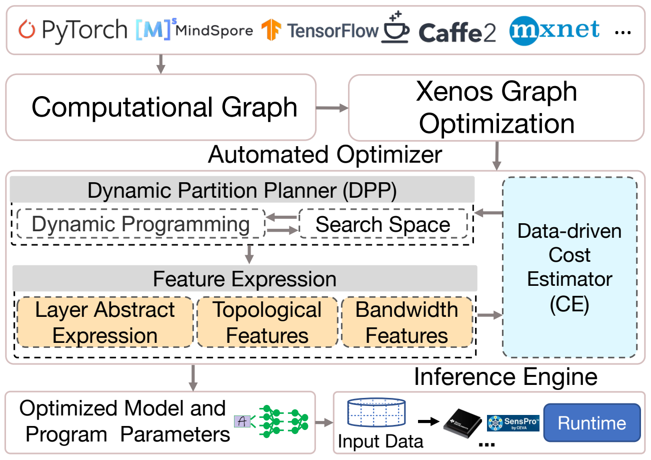


Fig. 3: The architecture of `FlexPie`

3 `FlexPie` Design

3.1 Architecture Overview

Figure 3 illustrates the architecture of `FlexPie`. `FlexPie` takes the computation graph as the general intermediate input and can support the inference computation for pre-trained models generated from multiple training frameworks (e.g., PyTorch, MindSpore, Tensorflow, etc). Besides, `FlexPie` also integrates pre-optimization strategies from Xenos [27] to optimize computation graph before it is fed into the automatic optimizer of `FlexPie`.

The automatic optimizer consists of two major modules, namely the data-driven cost estimator (CE) and the dynamic partition planner (DPP). CE (§3.2) is implemented with a Gradient Boosting Decision Tree (GBDT). The GBDT is pre-trained with more than 330K pieces of trace data, collected from running different model inference workloads under a variety of testbed settings. DPP runs a dynamic programming algorithm (§3.3) to search for the efficient model partition scheme. During runtime, DPP contacts CE to get an estimated time cost for the partition scheme in its consideration. Based on the estimated costs, DPP reserves the promising candidate schemes and continues its search. When the DP algorithm is completed, DPP will output the complete model partition scheme with the lowest estimated time cost. Then the inference engine drives multiple edge devices (nodes) to jointly execute the distributed inference computation according to the partition scheme.

3.2 Data-Driven Cost Estimator (CE)

To compare different partition schemes and make the choice between them, we need to decide an indicator to measure the cost of each scheme, so that the

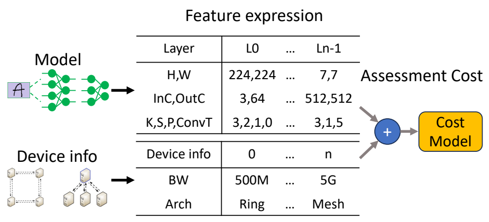


Fig. 4: Feature expression

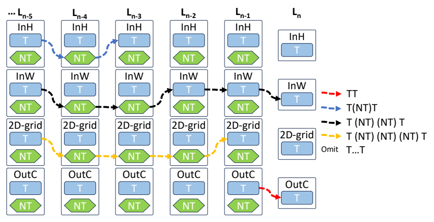


Fig. 5: Backtracking process in DPP

cost can serve as the guidance for running the dynamic programming algorithm in DPP. Instead of describing the estimated cost explicitly with some formula, which can be complicated when many variables are involved and fail to generalize, we choose a data-driven method and employ a machine learning model to predict the time cost. We use the GBDT as the cost estimator for `FlexPie` due to its simplicity and interpretability [5].

The cost estimator takes a group of features as input and outputs an estimated inference time cost to run the distributed inference for a given setting (i.e., running the inference on a specific model layer on a certain testbed). More specifically, we consider three aspects of features (illustrated in Figure 4) to predict the inference time: (1) the shape parameters of the model layer, including InH/OutH, InW/OutW, InC/OutC, K (kernel size), S (stride size), P (padding size), ConvT (convolution types); (2) the bandwidth between edge devices; (3) the communication architecture adopted by the edge cluster.[2]

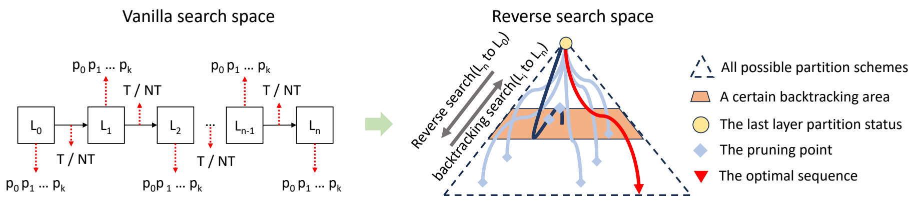
We have run multiple trials to train the i-Estimator and s-Estimator. Each trace data can be represented as a feature vector of 12 dimensions (Figure 4). While training the i-Estimator, we run the inference computation under different feature settings and use the inference time cost as the label value. While training the s-Estimator, we synchronize the model layer under different feature settings and use the communication time cost as the label value. Each estimator is trained with 330K samples, and the two estimators serve as the oracle to estimate the computation time (inference) and the communication time (synchronization) during the DP process (§3.3).

3.3 Dynamic Partition Planner (DPP)

To find the optimal partitioning scheme, DPP first constructs a search space and then runs a dynamic programming algorithm to search for the optimal partition scheme which can minimize the inference time.

Search Space The left side in Figure 6 sketches how DPP constructs the under-optimized search space (very large). Given a DNN with $n+1$ layers, denoted as

[2] We consider three communication architectures, i.e., ring-based, parameter server (PS)-based, and mesh-based architecture. We have transformed the architecture information into the categorical variable and fed the information to the cost estimator.



Vanilla search space
Reverse search space
L_0
L_1
L_2
L_{n-1}
L_n
T / NT
p_0 p_1 ... p_k
Reverse search(L_n to L_0)
backtracking search(L_i to L_n)
All possible partition schemes
A certain backtracking area
The last layer partition status
The pruning point
The optimal sequence

Fig. 6: DPP's search space

L_0-L_n, the search starts from L_n. For each layer L_i, DPP makes the two-step choices, and L_i is tagged with a pair $P_i = (p_i, t_i)$, $p_i \in \{\text{InH}, \text{InW}, \text{OutC}, \text{2D-grid}\}$ and $t_i \in \{\text{T}, \text{NT}\}$. DPP aims to find a sequence $S = [P_0, P_1, \cdots, P_n]$.

Step-1: DPP needs to choose the dimension to partition L_i. We have k different dimensions to consider (e.g. InH-based, InW-based, OutC-based, 2D-grid, etc) and DPP will choose one from them, and then continue to make choices in Step-2.

Step-2: DPP still needs to decide whether or not the output of L_i needs to be transmitted across edge nodes, in other words, DPP needs to choose between two modes. (1) Transmission (T) mode. The output of L_i is not sufficient to serve as the input of L_{i+1} because the input of L_{i+1} still requires the boundary data (§2.3). As a result, between the computation of L_i and L_{i+1} will be a round of the inter-node communication for nodes to transmit the necessary data between each other. (2) Non-Transmission (NT) mode. The output of L_i is sufficient to serve as the input of L_{i+1} because L_i's inference includes redundant computation. As discussed in §2.3, we need to consider this trade-off and flexibly decide whether or not to conduct redundant computation for each layer.

Since the typical DNN model can include tens or even hundreds of layers, simply enumerating every combination of partition schemes for each layer can lead to combinatorial explosion(explain in §**??**), we resort to the dynamic programming algorithm to search for the optimal partition scheme more efficiently.

Dynamic Programming Algorithm Algorithm 1 describes the workflow of the dynamic program process in `FlexPie`. As shown in the right side of Figure 6, DPP starts the search from the last layer of the model. In general, DPP is trying to evaluate each optimal substructure during the traversal and finally find out the optimal structure (solution). Specifically, during reverse search (from L_n to L_0), every time DPP considers a P_i where $t_i = \text{T}$. At this time, the algorithm will backtrack (from L_i to L_n) and generate a combined sequence, which contains multiple subsequences to be evaluated (as shown in Figure 5). For each subsequence, the starting point P_i and the ending point P_j will be T mode (i.e., $t_i = \text{T}$ and $t_j = \text{T}$). DPP will evaluate each subsequence one by one, take the optimal value and save it to the beginning t_i.

Key design-1: Reverse search. Given the search space, DPP needs to decide the search order. A key point is that the sequence S may contain multiple layers of consecutive NT cases (i.e., using redundant computation to save inter-node com-

Algorithme 1 : Dynamic Programming Algorithm for Model Partition

```
Input :
F($l_0 : l_n$) : Native network features
Nodes : The number of devices
BW : Inter-node bandwidth
Arch : Communication architecture
Output :
S : State sequence
Define:
P : Partition scheme
CS : Combined sequence
FE : Feature expression
F : Model feature
n : Number of layers
T : Transmission mode
NT : Non-Transmission mode
1  Function Main()
2      Initialize-Search-Space()
3      Initialize-State-Sequence()
4      DP()
5  Function Initialize-Search-Space()
6      for $i \leftarrow 0$ to $n$ do
7          if $i$ != $n$ then
8              P[$i$][$0 \cdots k$][$0 \cdots 1$] =
9              {{T, NT}, {T, NT}, $\cdots$}
10         else
11             P[$i$][$0 \cdots k$][$0 \cdots 1$] =
12             {{T, T}, {T, T}, $\cdots$}
13 Function Initialize-State-Sequence()
14     FE[$n$][$0 \cdots k$] =
15     Extract-Feature(P[$n$][$0 \cdots k$])
16     S[$n$][$0 \cdots k$] =
17     Estimate-Cost(FE[$n$][$0 \cdots k$])
18 Function DP()
19     for $i \leftarrow (n-1)$ to $0$ do
20         Traverse P[$i$][$0 \cdots k$][$0 \cdots 1$]
21         if P[$i$][$0 \cdots k$][$0 \cdots 1$] == T then
22             for $j \leftarrow (i+1)$ to $n$ do
23                 Traverse P[$j$][$0 \cdots k$][$0 \cdots 1$]
24                 if P[$j$][$0 \cdots k$][$0 \cdots 1$] == T then
25                     CS[$i \cdots j$][$0 \cdots k$][$0 \cdots 1$] =
                        Generate-Combined-Sequence()
26                     FE[$i \cdots j-1$][$0 \cdots k$][$0 \cdots 1$] =
27                     Extract-Feature(
28                     CS[$i \cdots j-1$][$0 \cdots k$][$0 \cdots 1$])
29                     if (Estimate-Cost(
30                     FE[$i \cdots j-1$][$0 \cdots k$][$0 \cdots 1$])
31                     + S[$j$][$0 \cdots k$]) < S[$i$][$0 \cdots k$] then
32                         S[$i$][$0 \cdots k$] = Estimate-Cost(
33                         FE[$i \cdots j-1$][$0 \cdots k$][$0 \cdots 1$])
34                         + S[$j$][$0 \cdots k$]
35                     else
36                         break;
```

munication cost), and the impact of NT cases cascades and will be propagated forward through the layers (L_i to L_0). More specifically, if P_{i+1} has $t_{i+1} = \text{NT}$, the inference of L_i needs to perform redundant computation. Following that, if P_i has $t_i = \text{NT}$, then inference of L_{i-1} also needs to perform redundant computation, and the inference of L_{i-1} needs to perform even more redundant computation. In such cases, if the dynamic programming algorithm uses searches from L_0 to L_n, it will lead to a large number of redundant evaluations and cannot establish the optimal substructure. We realize this problem, and decide to adopt the reverse search order to run the dynamic programming algorithm: the search will be performed in order from L_n to L_0 (line 19). DPP first initializes the search space (lines 5-12) and assigns T, NT to t_i of each layer L_i, k groups per layer (lines 8-9). The last layer is different from previous layers because it must be transmitted after computation (lines 11-12).

Key design-2: Skip all NT states in reverse search. During reverse search (i.e., $L_n \rightarrow L_i$), every time DPP only considers a P_i where $t_i = \text{T}$ (line 21) and starts backtracking (line 22). This is because if DPP considers a P_i where $t_i = \text{NT}$, the reverse search becomes a reverse full traversal search. In the process of backtracking, DPP does not evaluate the substructure ended with NT state (as shown in Figure 5) and only evaluates the substructure ended with T state (details explained next in Design 3).

Key design-3: Backtrack and generate combined sequences. DPP will first consider P_n where $t_n = \mathrm{T}$. Then, DPP extracts its features and sends them to CE for estimation. Finally, DPP can get a total of k costs of P_n where $t_n = \mathrm{T}$ and save to S$[n][0 \cdots k]$ (line 13-17).

Then, during reverse search (from L_n to L_{n-1}), DPP starts backtracking at P_{n-1} (where $t_{n-1} = \mathrm{T}$) and considers fully connecting P_{n-1} (where $t_{n-1} = \mathrm{T}$) and P_n (where $t_n = \mathrm{T}$). Then, DPP generates a combined sequence (a total of k^2 subsequences) and sends them to CE for estimation. DPP can obtain k costs of P_{n-1} (where $t_{n-1} = \mathrm{T}$) and add them to the costs in S$[n][0 \cdots k]$ respectively. Based on the comparison between sequences, DPP finally obtains the final k overall costs of P_{n-1} (where $t_{n-1} = \mathrm{T}$) and save to S$[n-1][0 \cdots k]$ (lines 19-36).

Next, DPP performs reverse search again (from L_{n-1} to L_{n-2}) and starts backtracking at P_{n-2} (where $t_{n-2} = \mathrm{T}$). We find DPP can estimate the combination of P_{n-2} (where $t_{n-2} = \mathrm{T}$) and P_{n-1} (where $t_{n-1} = \mathrm{T}$), but can not estimate the combination of P_{n-2} (where $t_{n-2} = \mathrm{T}$) and P_{n-1} (where $t_{n-1} = \mathrm{NT}$). This is because we cannot estimate P_{n-2} (where $t_{n-2} = \mathrm{T}$). DPP chooses to save the information in P_{n-1} (where $t_{n-1} = \mathrm{NT}$) until DPP encounters the P_n (where $t_n = \mathrm{T}$). Then, DPP evaluates P_{n-2} (where $t_{n-2} = \mathrm{T}$) and P_{n-1} (where $t_{n-1} = \mathrm{NT}$), and then sums up the cost with P_n (where $t_n = \mathrm{T}$). Based on the comparison between other sequences, DPP finally obtains the final k costs of P_{n-2} (where $t_{n-2} = \mathrm{T}$) and save to S$[n-2][0 \cdots k]$.

Why skip NT states? In our case, DPP only evaluates the substructure S_{sub} that starts with P_i where $t_i = \mathrm{T}$. Otherwise, if S_{sub} that starts with P_i where $t_i = \mathrm{NT}$, the time cost of S_{sub} is indeterminate because the computation workload of L_i will vary depending on whether the previous layer L_{i-1} chooses the Transmission mode or Non-transmission mode (i.e., whether $t_{i-1} = \mathrm{T}$ or $t_{i-1} = \mathrm{NT}$).[3] Therefore, DPP skips the sub-sequence starting with P_i where $t_i = \mathrm{NT}$. For every S_{sub} that starts with P_i where $t_i = \mathrm{T}$, DPP records the current time cost of S_{sub}. The S_{sub} will be used to compare with the other candidate sub-sequences and compose longer sub-sequences. Finally, the optimal structure (solution) will be established with a sequence S.

Piecing together. In summary, DPP adopts multiple pruning strategies to reduce the search space and further improve the search efficiency. (1) During reverse search, DPP can improve search efficiency by ignoring evaluating P_i where $t_i = \mathrm{NT}$ (line 21). (2) DPP uses S$[j][0 \cdots k]$ to reduce the number of evaluations during the backtracking process (lines 32-34). (3) DPP incrementally generates the combination sequences and avoids much ineffective backtracking in the planning process (lines 22-36, through dynamic thresholds).

Based on our algorithm design, we have the following theorem. The proof is included in our technical report [26].

[3] If $t_{i-1} = \mathrm{NT}$, the inference of L_i needs to conduct more computation to construct its input feature map (including the boundary data illustrated in §2.3). By comparison, if $t_{i-1} = \mathrm{T}$, the inference of L_i conducts less computation because the boundary data is obtained from the other edge nodes instead of redundant computation.

4 Experimental Evaluation

Testbeds: Our testbed includes 4 TMS320C6678[2] devices connected through SRIO. We have evaluated under different communication topologies, including Ring-based, PS-based and Mesh-based topologies.[4] In addition, we run tests with various settings of inter-device bandwidth: 5Gbps, 1Gbps and 500Mbps.

Benchmarks: We use four typical models as benchmarks to evaluate `FlexPie`, namely MobileNet [13], ResNet18 [12], ResNet101 [12] and Bert [15].

Baselines: We compare `FlexPie` with five baselines. Specifically, we compare `FlexPie` with 3 fixed partition schemes, i.e., Xenos[27](One-dim (OutC)), MoDNN[20] and DeepSlicing[28](One-dim (InH/InW)), DeepThings[29](2D-grid). Besides, we also consider two other baselines used by recent works. We refer to DINA [21] and PartialDI [8] and implement the layerwise partition, which uses different partition schemes for different model layers. We refer to AOFL [30] and EdgeCI [6] and incorporate data fusion into the fixed model partition scheme.

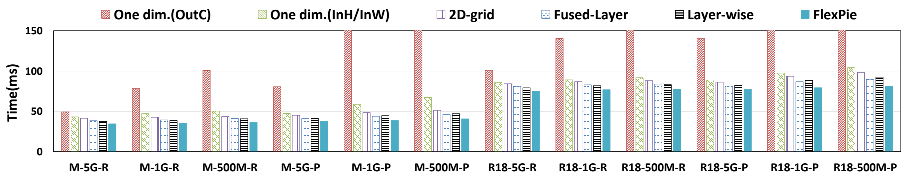
Metrics: We study the inference time cost and the DPP search time cost in our evaluation. We run each inference workload and each generated optimal sequence 1000 times and reported the average. Besides, in order to make an overall comparison between `FlexPie` and the baselines across all the test cases, we define a summative metric called *performance score*, to quantify the performance of each solution. *performance score* is defined as follows. For a given model benchmark and a given testbed setting, we run both `FlexPie` and the four baselines and obtain their inference time cost. Among the five time costs, denoted as $t_1 - t_5$, we choose the smallest one $\min\{t_1, t_2, t_3, t_4, t_5\}$ as the reference, and the *performance score* of each solution is computed as $Score_i = \frac{\min\{t_1,t_2,t_3,t_4,t_5\}}{t_i}$.

The value of each score will be a value between 0 and 1.0. If a partition strategy incurs a larger inference time, then its score will be smaller. The best partition strategy that yields the smallest inference time will be scored 1.0.

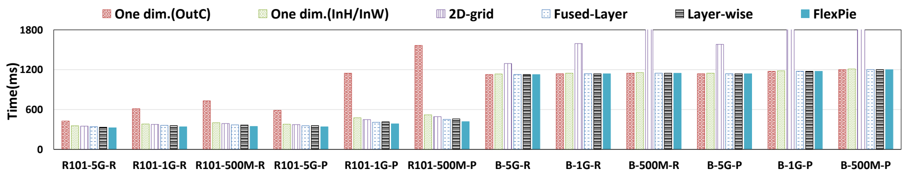
4.1 Comparison on 4-node Testbed

We compare the inference time cost of different partition schemes on the 4-node testbed. Figure 7 shows that, the 2D-grid partition performs best among the three baselines (OutC, InH/InW and 2D-grid) without layer-wise and fused-layer strategy. The OutC-based partition introduced non-trivial communication overheads. As illustrated in Figure 1(c), with OutC-based partitioning, each node has to fetch the input feature maps from all the other three nodes to suffice its inference computation, so the performance is the lowest. The performance of one-dim.(InH/InW) is between 2D-grid partition and OutC-based partition. However, not all layers perform optimally with 2D-grid partition (as shown in Figure 2), because 2D-grid partitions will cause varying degrees of imbalanced

[4] Our evaluation shows similar performance results for Ring-based and Mesh-based topologies. Due to space limits, we omit the results of Mesh-based topology.



(a) MobileNet, ResNet18 on Ring and PS



(b) ResNet101, Bert on Ring and PS

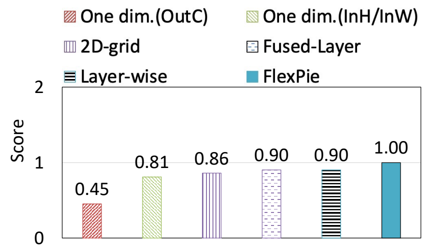
Fig. 7: Comparison on 4-node testbed

calculation. For example, in MobileNet, when the input feature map is 14x14x512 or 7x7x512, the 2D-grid has unbalanced calculations because neither the InH dimension (14) nor the InW dimension (14) is a multiple of the number of nodes (4). The OutC-based partitioning may not be optimal for overall model partitioning, but it can achieve balanced computation (i.e., dividing the 512 channels evenly among the 4 nodes).

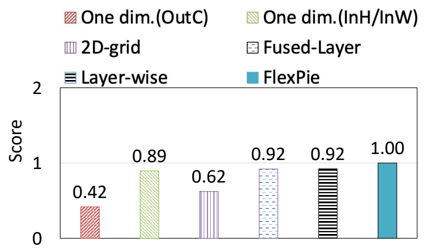
Compared with the One-dim and 2D-grid partition schemes, the layerwise partition introduces more flexibility and adopts different partition schemes for each layer. As a result, the layerwise partition yields better performance than the fixed One-dim and 2D-grid baselines. Different from layerwise partition, the layer fusion optimization accelerates the inference from the other direction. By fusing some model layers, the inference workflow can avoid redundant communication costs and in turn gain speedup. However, the previous works (e.g., DINA, EdgeCI) fail to incorporate both optimizations due to the explosive search space. Therefore, simply adopting layerwise optimization or fusion optimization cannot yield the best performance. By contrast `FlexPie` can efficiently find the proper partitioning scheme through its dynamic optimization strategies (§3.3). Therefore, `FlexPie` distinctly reduces the inference time for a wide range of models under different typologies.

Limitation: `FlexPie` provides little speedup when conducting inference with Bert. With a deeper dive, we realize that Bert model uses much matrix multiplication, but does not involve significant convolution computation as the other three benchmarks. Therefore, Bert already enjoys much easy parallelism in its nature. Neither layerwise flexibility nor inter-layer fusion brings distinct acceleration, leading to very close performance among different partition schemes.

Performance score: `FlexPie` achieves the highest performance score among all 5 solutions, and it outperforms the baselines with a speedup of 1.10-2.21×.



(a) Score on 4node



(b) Score on 3node

Fig. 8: Performance score comparison

4.2 Comparison on 3-node Testbed

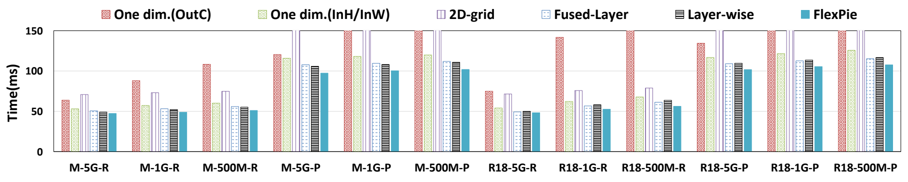
We then change the testbed setting and only use 3 nodes to run the inference tasks. In the 3-node test, the partition strategies including One-dim (InH/InW), One-dim (OutC) and 2D-grid still lead to imbalanced computation. But different from 4-node test, the 2D-grid partition strategy becomes the worst case when running on the 3-node testbed. This is because, one node needs to undertake much more (i.e., twice as much) computation workload as the other two nodes.

Based on the comparative evaluation in both 3-node testbed and 4-node testbed, we also demonstrate that, there is no "one-size-fits-all" fixed partition scheme to provide general performance improvement. More specifically, One-dim (OutC) and 2D-grid partition may yield the best performance under different bandwidths or different architectures. Since such statistic partition schemes can only reach their "sweet spots" under specific settings, they hardly provide general performance benefits to distributed inference under varying testbed configurations. By contrast, DPP enables `FlexPie` to adapt to different testbeds and automatically search for the appropriate partition strategies to yield high performance, making `FlexPie` more desirable for practical deployment.

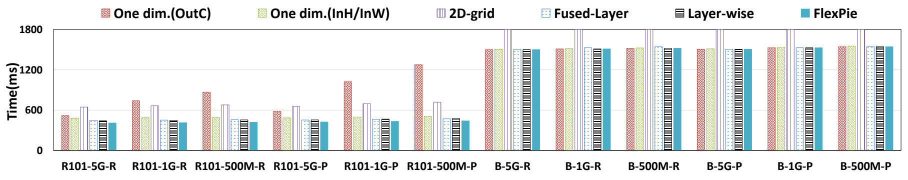
Performance score: `FlexPie` achieves the highest performance score among all 5 solutions, and it outperforms the baselines with a speedup of 1.08-2.39×.

5 Related Work

Distributed DNN Inference Distributed DNN inference can be categorized into *cloud-to-edge devices* [24,16], *edge server-to-edge device* [21,23], *cloud-to-edge server-to-edge device* [22,25,17,7] and *edge device-to-edge device* [29,28,20,30]. Among them, *edge device-to-edge device* deploys the DNN on the local terminal device and performs DNN inference entirely in a local collaborative manner. This paradigm focuses on inference latency and energy consumption, but it can be applied to high mobility scenarios or some remote and harsh environments. In this paper, we focus on how to obtain lower inference latency through flexible combinatorial optimization in device-device collaborative inference scenarios.



(a) MobileNet, ResNet18 on Ring and PS



(b) ResNet101, Bert on Ring and PS

Fig. 9: Comparison on 3-node testbed

Model Partitioning in Distributed Inference MoDNN [20] uses the one-dim to minimize the non-parallel data transmission time, but ignores the computational imbalance problem caused by the one-dim scheme. Deepslicing [28] achieves a trade-off between computation and synchronization through the Proportional Synchronization Scheduler. But it also adopts one-dim scheme, which constrains its optimization opportunities. DeepThings[29] uses a 2D-grid to perform model partitioning and implements a distributed work-stealing approach to enable dynamic workload distribution and balancing during inference runtime. However, it still suffers from the imbalanced distribution of computation tasks. In addition, many model partition strategies have been developed and applied in distributed training (e.g., [11,10]), and it will be an interesting future direction to study these partition strategies in the scenario of model inference.

6 Conclusion

This paper presents `FlexPie`, a new inference work to employ multiple edge devices to jointly execute the model inference task. `FlexPie` incorporates the flexible combinatorial optimization to automatically decide the partition scheme, which can reduce inference time cost, and yield desirable performance (up to 2.39×) under various benchmark settings.

7 Acknowledgment

This work is supported by the National Key Research and Development Program of China (No. 2021ZD0110202).

References

1. Nio-innovation. https://www.nio.cn/innovation.
2. TI TMS320C6678. https://www.ti.com/product/TMS320C6678.
3. Salma Abdel Magid, Francesco Petrini, and Behnam Dezfouli. Image classification on iot edge devices: profiling and modeling. *Cluster Computing*, 23:1025–1043, 2020.
4. Chongguang Bi and Guoliang Xing. Ramt: Real-time attitude and motion tracking for mobile devices in moving vehicle. *Proceedings of the ACM on Interactive, Mobile, Wearable and Ubiquitous Technologies*, 3(2):1–21, 2019.
5. Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.
6. Yanming Chen, Tong Luo, Weiwei Fang, and Neal N Xiong. Edgeci: Distributed workload assignment and model partitioning for cnn inference on edge clusters. *ACM Transactions on Internet Technology*, 24(2):1–24, 2024.
7. Swarnava Dey, Jayeeta Mondal, and Arijit Mukherjee. Offloaded execution of deep learning inference at edge: Challenges and insights. In *2019 IEEE International Conference on Pervasive Computing and Communications Workshops (PerCom Workshops)*, pages 855–861. IEEE, 2019.
8. Swarnava Dey, Arijit Mukherjee, Arpan Pal, and Balamuralidhar P. Embedded deep inference in practice: Case for model partitioning. In *Proceedings of the 1st Workshop on Machine Learning on Edge in Sensor Systems*, pages 25–30, 2019.
9. Argha Chandra Dhar, Arna Roy, Subrata Biswas, and Bashima Islam. Studying the security threats of partially processed deep neural inference data in an iot device. In *Proceedings of the 20th ACM Conference on Embedded Networked Sensor Systems*, pages 845–846, 2022.
10. Jinkun Geng and et al. Elasticpipe: An efficient and dynamic model-parallel solution to dnn training. In *ScienceCloud'19*.
11. Jinkun Geng and et al. Fela: Incorporating flexible parallelism and elastic tuning to accelerate large-scale dml. In *ICDE'20*.
12. Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770–778, 2016.
13. Andrew G Howard, Menglong Zhu, Bo Chen, Dmitry Kalenichenko, Weijun Wang, Tobias Weyand, Marco Andreetto, and Hartwig Adam. Mobilenets: Efficient convolutional neural networks for mobile vision applications. *arXiv preprint arXiv:1704.04861*, 2017.
14. Ling Hu and Qiang Ni. Iot-driven automated object detection algorithm for urban surveillance systems in smart cities. *IEEE Internet of Things Journal*, 5(2):747–754, 2017.
15. Ganesh Jawahar, Benoît Sagot, and Djamé Seddah. What does bert learn about the structure of language? In *ACL 2019-57th Annual Meeting of the Association for Computational Linguistics*, 2019.
16. Hyuk-Jin Jeong, Hyeon-Jae Lee, Chang Hyun Shin, and Soo-Mook Moon. Ionn: Incremental offloading of neural network computations from mobile devices to edge servers. In *Proceedings of the ACM symposium on cloud computing*, 2018.
17. Chang-You Lin, Tzu-Chen Wang, Kuan-Chih Chen, Bor-Yan Lee, and Jian-Jhih Kuo. Distributed deep neural network deployment for smart devices from the edge to the cloud. In *Proceedings of the ACM MobiHoc workshop on pervasive systems in the IoT era*, pages 43–48, 2019.

18. Yu Liu, Yingling Mao, Zhenhua Liu, Fan Ye, and Yuanyuan Yang. Joint task offloading and resource allocation in heterogeneous edge environments. *IEEE Transactions on Mobile Computing*, 2023.
19. Changchun Long, Yang Cao, Tao Jiang, and Qian Zhang. Edge computing framework for cooperative video processing in multimedia iot systems. *IEEE Transactions on Multimedia*, 20(5):1126–1139, 2017.
20. Jiachen Mao, Xiang Chen, Kent W Nixon, Christopher Krieger, and Yiran Chen. Modnn: Local distributed mobile computing system for deep neural network. In *Design, Automation & Test in Europe Conference & Exhibition (DATE), 2017*, pages 1396–1401. IEEE, 2017.
21. Thaha Mohammed, Carlee Joe-Wong, Rohit Babbar, and Mario Di Francesco. Distributed inference acceleration with adaptive dnn partitioning and offloading. In *IEEE INFOCOM 2020-IEEE Conference on Computer Communications*, pages 854–863. IEEE, 2020.
22. Pei Ren, Xiuquan Qiao, Yakun Huang, Ling Liu, Calton Pu, and Schahram Dustdar. Fine-grained elastic partitioning for distributed dnn towards mobile web ar services in the 5g era. *IEEE Transactions on Services Computing*, 2021.
23. Nanliang Shan, Zecong Ye, and Xiaolong Cui. Collaborative intelligence: Accelerating deep neural network inference via device-edge synergy. *Security and Communication Networks*, 2020:1–10, 2020.
24. Mingcong Song, Kan Zhong, Jiaqi Zhang, Yang Hu, Duo Liu, Weigong Zhang, Jing Wang, and Tao Li. In-situ ai: Towards autonomous and incremental deep learning for iot systems. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 92–103. IEEE, 2018.
25. Min Xue, Huaming Wu, Ruidong Li, Minxian Xu, and Pengfei Jiao. Eosdnn: An efficient offloading scheme for dnn inference acceleration in local-edge-cloud collaborative environments. *IEEE Transactions on Green Communications and Networking*, 6(1):248–264, 2021.
26. Runhua Zhang, Hongxu Jiang, Jinkun Geng, Yuhang Ma Ma, Chenhui Zhu, and Haojie Wang. Flexpie: Accelerate distributed inference on edge devices with flexible combinatorial optimization [technical report]. `https://github.com/Happyrhzhang/FlexPie/blob/main/FlexPie_Technical_Report.pdf`, 2025.
27. Runhua Zhang, Hongxu Jiang, Fangzheng Tian, Jinkun Geng, Xiaobin Li, Yuhang Ma, Chenhui Zhu, Dong Dong, Xin Li, and Haojie Wang. Xenos: Dataflow-centric optimization to accelerate model inference on edge devices. In *International Conference on Database Systems for Advanced Applications*, pages 535–545. Springer, 2023.
28. Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. Deepslicing: collaborative and adaptive cnn inference with low latency. *IEEE Transactions on Parallel and Distributed Systems*, 32(9):2175–2187, 2021.
29. Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 37(11):2348–2359, 2018.
30. Li Zhou, Mohammad Hossein Samavatian, Anys Bacha, Saikat Majumdar, and Radu Teodorescu. Adaptive parallel execution of deep neural networks on heterogeneous edge devices. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing*, pages 195–208, 2019.