ADaPS-FC: Enhancing Performance Through Channel Division on Edge Computing Platforms

JAUME MATEU CUADRAT, Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of)
BERNHARD EGGER, Computer Science and Engineering, Seoul National University, Seoul, Korea (the Republic of) and Lucerne University of Applied Sciences and Arts, Lucerne, Switzerland

The proliferation of Artificial Intelligence (AI) applications has driven a substantial demand for deploying Neural Networks (NNs) across a wide range of device platforms. Modern networks have become so computationally intensive that even relatively simple Convolution Neural Networks (CNNs) often require costly specialized hardware for efficient execution. To mitigate this challenge, techniques that distribute inference workloads across interconnected, resource-constrained devices have become increasingly important. While existing approaches typically rely on empirical models or support only limited partitioning dimensions, we introduce ADaPS-FC, a novel framework for optimally distributing CNN inference workloads across heterogeneous embedded devices. Our analytical model partitions the height, width, and channel dimensions of 4D CNN tensors while also exploring partition configurations that enable layer fusion. In addition, it introduces weight-division strategies for fully connected layers and small output operators to improve both memory usage and latency, fully accounting for each device's computational capabilities and inter-device communication overhead.

This work extends the previously proposed ADaPS method by addressing the limitations encountered in the later, non-divisible layers of CNN models. To efficiently navigate the large search space of possible partitionings, ADaPS-FC employs a hybrid optimization algorithm that combines Alpha–Beta pruning with dynamic programming for the earlier layers, where height or width division is feasible. For the later layers, where the prior method was unable to apply spatial partitioning, the framework introducess weight (channel) division to enable further distribution opportunities. We evaluate ADaPS-FC across multiple CNN architectures deployed on interconnected heterogeneous hardware, using configurations ranging from two to four devices. Experimental results show that ADaPS-FC improves inference time by 1.3× on average over previous state-of-the-art.

CCS Concepts: • **Computing methodologies** → *Distributed artificial intelligence*; *Parallel programming languages*; **Game tree search**; **Heuristic function construction**.

Additional Key Words and Phrases: Convolutional network, data partitioning, minmax, parallelization

1 Introduction

This paper specifically addresses the needs of end-users and small-to-medium enterprises (SMEs) that require low-latency inference capabilities for relatively large CNN models on commodity hardware. In such scenarios, batch sizes are typically constrained, with minimizing inference latency as the primary concern. Existing approaches to this challenge generally fall into two broad categories: model partitioning and data partitioning.

Model partitioning involves distributing different layers of the CNN across multiple devices [2, 7, 11, 19], while data partitioning divides the input tensors within individual layers across devices [4, 8–10, 17, 21–23]. Several implementation strategies have been proposed: some approaches rely on a master node for coordination [8, 17, 23],

Extension of Conference Paper [9]. Authors' Contact Information: Jaume Mateu Cuadrat, jaume@csap.snu.ac.kr, Seoul National University, Seoul, South Korea; Bernhard Egger (corresponding author), bernhard@csap.snu.ac.kr, Seoul National University, Seoul, South Korea.

ACM 1558-3465/2026/6-ART
https://doi.org/10.1145/3820058

Table 1. Stage-wise analysis of compute-to-weight ratios and layer distribution across representative CNN architectures. Each cell reports the aggregate compute ratio followed by the percentage of layers in the corresponding stage.

Model	Early Stage	Mid Stage	Late Stage
VGG16	12.4 (31%)	6.8 (44%)	2.1 (25%)
MobileNetV2	18.7 (35%)	9.5 (47%)	3.0 (18%)
InceptionV3	14.2 (38%)	7.3 (47%)	2.5 (15%)

others perform partitioning during compilation [4, 9, 10], and more sophisticated methods take into account device heterogeneity and inter-device communication costs [21, 22].

While data partitioning can effectively reduce inference latency, determining the optimal division strategy across tensor dimensions remains a critical challenge. CNN tensors can be partitioned along four dimensions: input channels, output channels, height, and width. Existing approaches primarily focus on height and width (or also called spatial dimensions) feature map partitioning due to its general applicability and compatibility with computation–communication overlap techniques such as operator fusion.

As shown in Table 1, CNN models consistently exhibit a structured pattern in which early and mid stages are dominated by large activation tensors, while late stages show significantly reduced spatial dimensions (Height and Width dimensions) and relatively higher weight influence. This results in three characteristic regimes: (i) activation-dominant layers, (ii) balanced layers, and (iii) compact spatial layers. The proposed framework does not explicitly operate on these phases; they are introduced solely to explain the observed structural patterns across CNN architectures.

This structural evolution directly impacts partitioning feasibility. In early and mid stages, spatial feature map partitioning is generally feasible and preferred due to sufficiently large spatial dimensions that allow balanced workload distribution. This preference is motivated by its lower communication and synchronization overhead, as spatial partitioning avoids inter-device partial sum exchanges in convolution operations. However, as spatial dimensions size decreases in later stages, height and width feature map partitioning becomes infeasible or inefficient when tensor dimensions for the height and width approach or fall below the number of available devices. In these cases, channel-based partitioning provides a structurally consistent alternative, as channel dimensions remain available across all layers.

Importantly, ADaPS-FC does not treat spatial and channel partitioning as a global trade-off. Spatial partitioning is applied when feasible and preferred due to lower synchronization overhead and to avoid the introduction of additional partial sums. To efficiently navigate this complex solution space of height/width partitioning, the framework employs advanced search techniques inspired by Minmax [14] and Alpha-Beta pruning [5] algorithms. When spatial partitioning is not feasible, channel partitioning is considered and selected only if a hardware-aware cost model indicates lower cost model score. Unlike spatial partitioning, which requires combinatorial exploration of spatial split configurations due to varying input/output feature map dimensions and the resulting trade-offs in communication and load balancing, channel partitioning exhibits a significantly reduced design space. In channel partitioning, the split is primarily governed by channel allocation across devices, which is constrained by hardware capacity and tensor channel dimensionality rather than spatial structure. As a result, the decision space is largely structured and can be evaluated directly through a cost model without exhaustive exploration of multiple partition candidates.

This design is orthogonal to ADaPS [9], where channel partitioning is applied only on layers where spatial partitioning is not feasible. In contrast to prior work such as DeeperThings [17], where the division is static ours is based on hardware capabilities and network characteristics.

Table 2. Comparison of related work. *H*, *W*, and *C* denote partitioning in the *H*eight, *W*idth, and *C*hannel dimension.

Paper	Search Type	Heterogeneous Devices	Partitioning Directions	Communication	Layer Fusion
MoDNN [8]	Layer by layer	No	H or W	Centralized	No
DeepThings [23]	Grouped layers	No	H or W	Centralized	Yes
DeeperThings [17]	Grouped layers	No	C	Centralized	Yes
CoEdge [21]	Layer by layer	Yes	H or W	Centralized	No
DeepSlicing [22]	Grouped layers	Yes	W	Centralized	Yes
EdgeFlow [4]	Analytical model	Yes	H	Decentralized	Yes
BBGraP [10]	Fixed partition	No	H or W or C	Decentralized	No
ADaPS [9]	Analytical model	Yes	H or W	Decentralized	Yes
ADaPS-FC (this work)	Analytical model	Yes	H or W and C	Decentralized	Yes

To demonstrate the practical applicability of ADaPS-FC in real-world heterogeneous edge environments, we evaluate its performance across configurations involving two to four devices, consisting of various Raspberry Pi boards.

The main contributions of this work are as follows:

- A flexible technique for partitioning inference workloads along the height and width dimensions of a tensor across heterogeneous edge device configurations.
- An efficient search space exploration method using a specialized tree structure and pruning techniques that significantly reduce optimization time.
- A novel approach for layer fusion across partitions, considering the end-to-end characteristics of the CNN architecture to maximize performance.
- The introduction of channel input and output division to the existing method, enabling partitioning when height and width divisions are not possible.

The remainder of this paper is structured as follows: Section 2 reviews related work. Section 3 introduces the key concepts underlying our approach. Section 4 provides a detailed description of the ADaPS-FC framework and its operation. Section 5 evaluates our approach against state-of-the-art methods, and Section 6 presents the conclusions of the work.

2 Related Work

Recent research has focused on improving inference latency through three primary approaches: data partitioning, model partitioning, and optimization techniques (see Table 2). Model partitioning has emerged as the predominant strategy, with significant contributions from works such as Combining [11], Coin [19], and other notable systems [2, 7]. Data partitioning techniques include approaches like MoDNN [8], DeepThings [23], DeeperThings [17], CoEdge [21], DeepSlicing [22], EdgeFlow [4], BBGraP [10], and ADaPS [9]. These methods focus on distributing the computational workload across multiple devices, particularly for resource-constrained platforms where traditional inference methods are impractical due to limitations in memory and computational power.

2.1 Centralized vs. Decentralized Approaches

Distributed CNN inference across multiple devices requires careful coordination to ensure correct execution and data synchronization. Coordination mechanisms generally fall into two categories:

Centralized approaches rely on a master device to orchestrate synchronization among all participating devices. The master dynamically creates tasks and distributes them to idle devices, ensuring balanced workload distribution and optimal resource utilization. While this approach simplifies coordination, it introduces a potential bottleneck and a single point of failure at the master device.

Decentralized approaches, by contrast, distribute synchronization and correctness responsibilities across the individual devices. This eliminates the overhead associated with a master device but increases system complexity. The compiler must incorporate performance models to calculate a partitioning scheme that balances execution across devices while considering device capabilities and network bandwidth. Recent decentralized methods include EdgeFlow [4], BBGraP [10], and ADaPS [9].

2.2 Search Space Exploration and Layer Fusion

Partitioning CNN workloads across multiple devices presents a computationally intensive optimization challenge, with an exponentially growing search space. Efficient search strategies are essential for identifying the optimal partitioning that balances the computational load and minimizes communication overhead, especially as the number of devices and network complexity increases.

Earlier works, such as MoDNN [8] and CoEdge [21], optimize each layer individually and synchronize after each division at the master device. More advanced approaches, including DeepThings [23] and DeeperThings [17], minimize data synchronization within layers while still synchronizing between layers.

In distributed inference, layer fusion is critical to maximizing performance by minimizing inter-device data movement. Existing methods, however, have significant limitations:

BBGraP [10] implements data partitioning layer by layer, dividing the output based solely on the number of available devices, but fails to account for device-specific parameters or capabilities.

DeepSlicing [22] advances the state-of-the-art by searching across multiple fused layers, though it is limited to a fixed number of layers rather than optimizing the partitioning strategy itself.

EdgeFlow [4] addresses data transfer overhead by solving a linear optimization problem that considers only the previous layer to minimize inter-layer data transfers. However, this approach introduces restrictions on output patterns, as it prohibits the data overlaps essential for convolutional operations, where stride and width parameters naturally create input overlaps. Consequently, if the previous layer cannot produce overlapped outputs, EdgeFlow cannot perform layer fusion, resulting in increased data transfers between devices.

ADaPS-FC addresses these limitations by implementing a comprehensive approach to layer fusion, considering the entire CNN architecture while supporting flexible partitioning across multiple tensor dimensions. ADaPS-FC dynamically optimizes partitioning strategies based on computational capabilities and communication constraints, enabling more efficient execution plans on heterogeneous hardware configurations.

2.3 Fully Connected and Small Output Layers

Many of the methods mentioned above face challenges when partitioning the later layers or small output layers, particularly when the output of these layers is too small to be partitioned by height or width. In such cases, the layer is assigned to a single device for execution.

DeeperThings [17] addresses this issue by using channel input and output division for the later layers. It applies a formula to determine which layers to divide, based on the weight and feature map memory footprint: when the weight memory footprint exceeds that of the feature map, a channel division is assigned to the operator. However, this approach is data transfer-intensive and introduces partial sums into the graph (Section 3.4), which can negatively impact inference time.

Our method, by contrast, focuses on dividing only the layers that were not previously handled by ADaPS partitioning. We introduce a decision-making process that determines whether dividing a layer is beneficial or

Table 3. Data partitions types and their operators

Type	Allowed Partitions	Operators
Type 1	H,W,Ci, Co	Conv, GeMM
Type 2	H,W,C	DWConv, Pool, AvgPool, MaxPool
Type 3	H,W,C	Mul, Add, HardSwish, HardSigmoid, BatchNorm, Relu
Type 4	C	ReduceMean, Dropout
Type 5	None	Reshape, Flatten, Concat, Crop

whether it is better to keep it on a single node. This decision is based on the same cost model used during search space exploration in prior work [9], balancing performance against data transfer overhead.

3 CNN Computational Graph Partitioning

This section introduces the core concepts and formal definitions underpinning our partitioning framework. We begin by categorizing CNN operators based on their partitioning characteristics and then present the mathematical formulations for determining the input requirements corresponding to partitioned outputs.

3.1 Operator Classification

When partitioning CNN workloads, operators exhibit different behaviors that influence how they can be efficiently divided. Building upon formulations from EdgeFlow [4] and BBGraP [10], we classify operators into five distinct types based on their mathematical properties and partitioning constraints. Operator classification is used only to define partitioning feasibility constraints and does not influence the cost model or optimization objective. This classification is summarized in Table 3.

Type 1 and **Type 2** Operators include operators that involve weights, padding, or stride parameters, such as convolutional and pooling layers. These operators are characterized by neighborhood dependencies, meaning that the output elements depend on multiple adjacent input elements. Partitioning these operators by height or width requires special handling to account for these dependencies. **Type 1** operators require two types of channel division, depending on which feature map (input or output) is being partitioned. However, **Type 2** operators, along with **Type 3** and **Type 4**, exhibit simpler behavior when dividing the channel dimension, as the input and output dimensions remain consistent after partitioning. **Type 3** Operators behave similarly to **Type 2** when partitioning along the height and width dimensions, with similar constraints and requirements. **Type 4** and **Type 5** Operators transform tensor dimensions or aggregate information across the height and width dimensions. In our framework, operators of these types are not partitioned along the height and width dimensions. Additionally, **Type 5** operators also cannot be divided by the channel dimension, as the same constraints that apply to height and width partitioning also affect the channel dimension in these cases.

The classification provides a structured approach to understanding how different types of operators behave under partitioning, allowing us to design a framework that can efficiently partition CNN workloads while maintaining computational integrity.

3.2 Partition Representation and Tensor Indexing

Our framework adopts the NCHW tensor representation convention, where N represents the batch dimension, C denotes channels, and H and W specify the height and width dimensions, respectively. The origin (index 0) is located at the top-left corner of the tensor. Partitions are defined using interval notation across these dimensions.

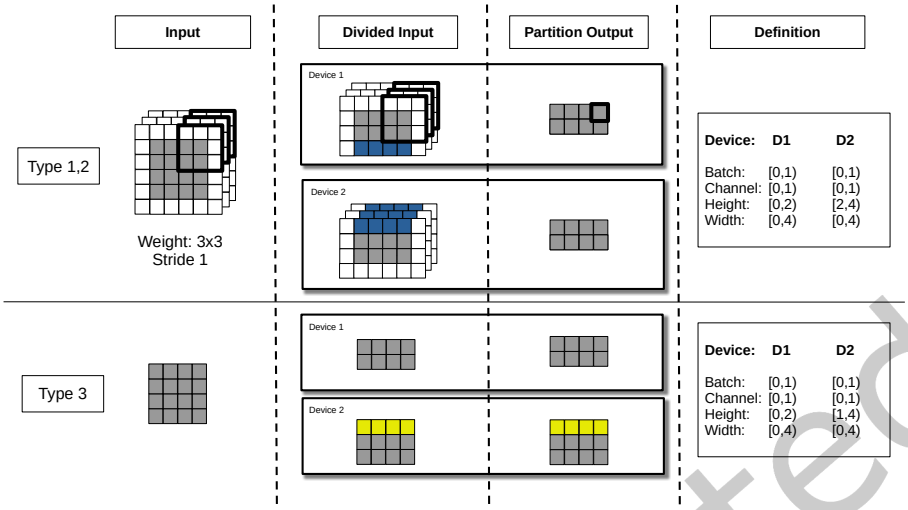


Fig. 1. Illustration of partitioning effects across different operator types. Type 1 and Type 2 operators (top) benefit from computational overlaps by reducing data dependencies (blue regions), while Type 3 operators (bottom) experience only redundant computation (yellow regions) without communication benefits.

In this representation, the origin (index 0) is located at the top-left corner of the tensor. Partitions within the tensor are defined using interval notation across one or more dimensions.

Each partition is represented as a collection of intervals in one or more dimensions. For example, a height partition denoted as H:[0,2)[2,4), which indicates two intervals: the first spanning height indices 0 to 2, and the second covering indices 2 to 4. The left value in each interval specifies the starting index (inclusive), while the right value indicates the ending index (exclusive). When partitioning occurs along a single dimension, each interval is mapped to a specific device.

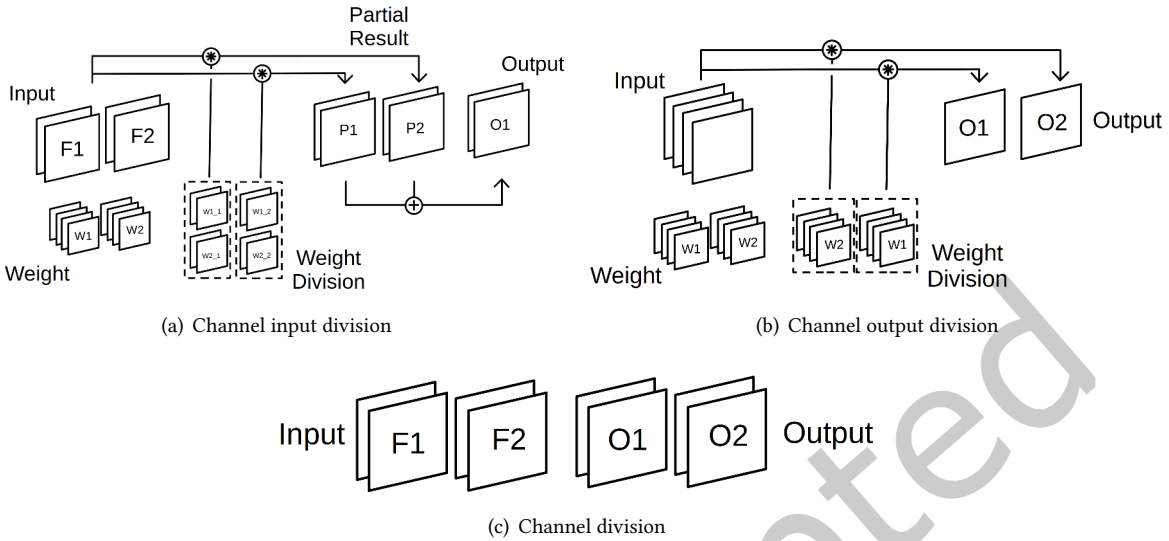
This partitioning representation ensures that the framework can efficiently allocate workloads to different devices while preserving the integrity of the computations and minimizing data transfer overhead.

3.3 Partitioning Concepts: Height and Width

Figure 1 illustrates two core concepts in our partitioning approach using representative examples from both Type 1 and Type 2 (top) and Type 3 (bottom) operators: dependencies and computation overlaps. The Type 1 and Type 2 example shows a convolutional operation with padding, while the Type 3 example demonstrates an element-wise operation.

Dependencies (highlighted in blue in Figure 1) represent the values (output activations from the preceding layer) that must be transferred between devices, creating synchronization points in the execution flow. These dependencies directly affect communication overhead and serve as a primary optimization target in our framework.

Computation overlaps (shown in yellow in Figure 1) occur when identical computations are redundantly performed across multiple devices. These overlaps represent a trade-off between computational redundancy and communication cost. For Type 1 and Type 2 operators, overlapping computations can reduce communication requirements by eliminating data dependencies. In contrast, for Type 3 operators, computation overlaps typically increase the computational load without providing any communication benefits.



(a) Channel input division

(b) Channel output division

(c) Channel division

Fig. 2. Types of Channel division depending on the operator type (Table 3): (a) and (b) correspond to Type 1, (c) correspond to Type 2,3,4. The numbers correspond to a device index while the letters represent *F* for Feature map, *O* for Output, *W* for Weight and *P* for Partial sum.

3.4 Partitioning Concepts: Channel

Figure 2 illustrates the different types of channel division based on the operator being partitioned (as summarized in Table 3). Figures 2(a) and 2(b) are related to Type 1 operators, where the weight tensor modifies the dimensions of both the input and output feature maps and requires partial summation.

Figure 2(a) provides a step-by-step illustration of channel input division, where both the weight tensor's input channels and the input feature map channels are divided. To maintain consistency, the number of channels in the weight tensor must match the number of channels in the input feature maps (F1 and F2). Although the number of channels in the input matches the output feature map channels, the operation is incomplete at this stage. As shown by P1 and P2, the resulting tensors only represent partial sums of the operation. Thus, an additional summation operator is required to accumulate these partial results and produce the final output feature map (O1).

In channel output division (Figure 2(b)), the weight tensor's output channels and the output feature map channels are divided. Unlike in channel input division, the partial sum step is avoided, as the summation occurs during the convolution operation itself. However, the entire input feature map must be provided for the division of the output.

For Type 2 operators, the main distinction is that partial summation is not required after the operation. Consequently, the input and output dimensions remain consistent throughout the operation.

Figures 9(b) and similar diagrams illustrate the behavior of Type 2, Type 3, and Type 4 operators. These operators maintain consistent input and output sizes, and no partial summation is required. This scenario represents the ideal case since, when connected to other operators of the same type, no data transfer is needed, thus optimizing efficiency.

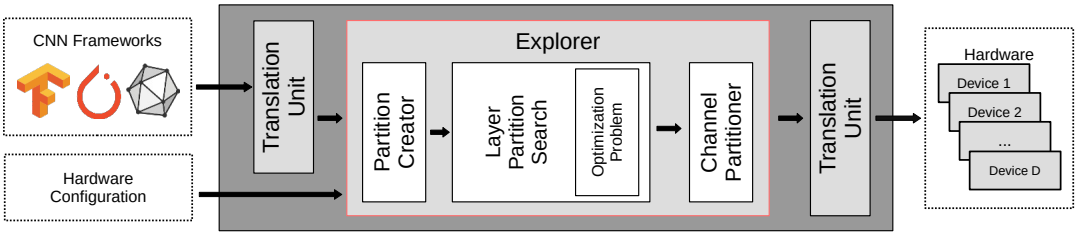


Fig. 3. ADaPS architecture highlighting the search space exploration module (in red) within the optimization pipeline.

4 Implementation

ADaPS-FC introduces a novel search space exploration methodology that automatically identifies optimal partitioning strategies across heterogeneous devices. The framework considers both computational capabilities and communication constraints, enabling effective workload distribution while minimizing data transfer overhead.

During its search, ADaPS-FC considers each device's processing speed, bandwidth constraints, and data transfer considerations. Unlike related work, ADaPS-FC recursively explores optimal partitioning for layer fusion across the entire CNN, not just adjacent layers. Finally, the Channel Partitioner Module (shown in Figure 3) divides the layers that were not previously divided if the inference was to improve.

4.1 Search Space Exploration and Channel Partitioner

Previous works have shown significant limitations in adaptability. For instance, BBGraP [10] employs naive partitioning without considering hardware specifications, while EdgeFlow [4] restricts its search to immediate layer pairs and prohibits overlapped output partitions. In contrast, ADaPS-FC addresses these limitations with a sophisticated search strategy that takes the heterogeneous hardware environment into account.

ADaPS-FC extends the partitioning to include fully connected (FC) and small-output layers, a capability not considered by prior approaches such as DeeperThings [17], which only considers feature map and weight sizes. ADaPS-FC, however, uses a performance model to evaluate whether the remaining layers—those that were not previously partitioned—will benefit from channel division and improve inference time. While channel division can reduce the memory footprint of weights, it often comes with increased data transfer costs, making it important to evaluate whether the benefits in memory efficiency outweigh the added complexity. Unlike other methods, ADaPS-FC only divides layers that were not previously partitioned, ensuring that this operation is orthogonal to the initial search space.

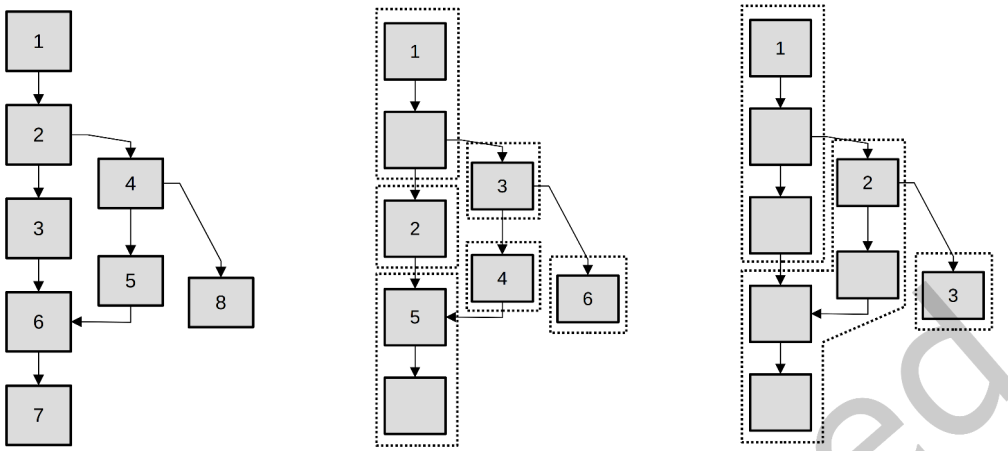
The search space exploration engine and channel partitioner in ADaPS-FC consist of five integrated components:

(1) **Path Analysis**: This component handles networks with multiple paths and shortcuts by breaking down complex graphs into manageable sub-blocks. It transforms multi-path problems into a series of single-path segments, ensuring partition consistency across the network.

(2) **Partition Generation**: Instead of exhaustively enumerating all possible combinations of partitions, this component uses a procedural approach to dynamically generate specific partition configurations as needed. ADaPS stores only the essential partitioning parameters and can reconstruct any partition in linear time.

(3) **Performance Evaluation**: For each candidate partition, a performance model calculates the expected execution time by considering computational workload, communication volume, and memory requirements. This allows for a quantitative comparison of different partitioning strategies.

(4) **Layer Fusion Optimization**: This component searches for optimal partitioning across consecutive layers, dynamically adjusting previously determined partitions. The algorithm revises earlier partitioning decisions



(a) Initial network graph (b) Results after path splitting (c) Final result after path fusion

Fig. 4. Path decomposition. Dotted rectangles represent identified sub-blocks, the indexes are the network processing order.

when downstream layers reveal more efficient global configurations. It determines when the benefits of layer fusion (such as reduced communication overhead) outweigh the associated costs.

(5) **Fully Connected Layers Division**: In this final stage, the compiler attempts to divide previously undivided fully connected layers along the channel dimension. This step is applied to layers that were not divisible in earlier stages and aims to optimize their partitioning for improved inference performance using channel divison.

These components enable ADaPS-FC to systematically explore a vast search space while maintaining computational efficiency. The framework identifies near-optimal solutions for distributing CNN workloads across heterogeneous devices, balancing computation, memory usage, and communication overhead.

In the following sections, we describe each of these components in detail.

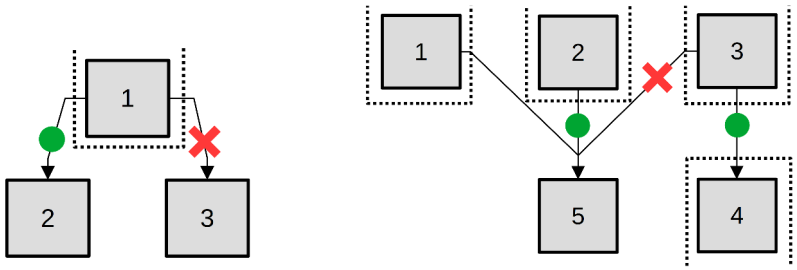
4.2 Path Analysis

Modern neural networks frequently feature complex topologies with multiple execution paths, skip connections, and merge points. This presents a significant challenge for partitioning optimization, as naively traversing the graph could lead to inconsistent partitioning decisions or inefficient search patterns. ADaPS addresses these challenges by decomposing the network graph into manageable sub-blocks before calculating the partitions.

The decomposition of the graph is divided into two steps: **splitting** and **fusing**. Figure 4 shows a complete example of both steps.

4.2.1 First Step: Splitting. In this step, we create sub-blocks that can already be evaluated independently, although they are not yet fully optimized. To generate these sub-blocks, we follow the processing order of the initial graph. When we encounter a node with two outputs, we close the current sub-block and start a new one; for example, *sub-block 2* in Figure 4(a).

Another condition for closing a sub-block occurs when one of the connected operators has multiple inputs, as in the case of *sub-block 3, 5, and 6* in Figure 4(a).



(a) One input (b) Multiple inputs

Fig. 5. Examples for the type of fusion depending on the number of inputs for the fusing step. The indexes are the network processing order. Green dots are the fused path and red crosses are blocked paths.

4.2.2 Second Step: Fusing. In this final step, we fuse all possible sub-blocks based on the inputs of the original graph. Although the previously generated sub-blocks can already be evaluated independently, some of them can be further fused to achieve better results through potential layer fusion.

This step considers two different conditions depending on the number of inputs of each sub-block:

- **One input**: When a sub-block has a single input, it is fused with the first available index, following a left-to-right order. Figure 5(a) shows an example of this case, where *sub-block 2* is fused with *sub-block 1*. However, since *sub-block 3* has no available options (as *sub-block 1* is already assigned), it must remain as a separate sub-block.
- **Multiple inputs**: In the case of multiple inputs, the fusion order is reversed, starting from the right, as shown in Figure 5(b). In this example, *sub-block 3* is already assigned to *sub-block 4*. Therefore, *sub-block 5* must be assigned to the next available node from the right, which is *sub-block 2*.

While our decomposition approach effectively reduces the complexity of the partitioning search problem, it does not incorporate cost-based optimization for the sub-block boundaries themselves. This simplification may occasionally result in sub-optimal decomposition patterns. However, the substantial reduction in search space complexity outweighs potential inefficiencies at sub-block boundaries. It is important to note that this decomposition does not constrain layer fusion opportunities within each sub-block. The subsequent optimization phases may still identify and implement fusion across any layers within a sub-block, regardless of their position or connectivity pattern. Furthermore, while layers at sub-block boundaries are not candidates for cross-boundary fusion, their individual partitioning strategies remain fully optimizable.

4.3 Partition Creation

The search space of one layer's partitioning grows exponentially with the number of devices. If no constraints are applied, there are $O(M^D)$ possible partitionings, where M represents the output feature map dimension size and D is the number of deployed devices. Layers are independent, yielding a search space complexity of $O(M_l^D)$ $\forall l \in Layers$. Exhaustively evaluating all possible partitions is impossible; therefore, we employ an adaptive thresholding approach to constrain the search space while preserving the exploration of the most promising partitioning configurations.

4.3.1 Bound Determination for Partition Sizes. To effectively prune the search space, we first establish minimum and maximum partition sizes for each device. The minimum partition size for device d is calculated using Equation 1, where out_size represents the output feature map dimension to be partitioned, and $speed_d$ denotes

the processing capability of device d relative to all devices in the system.

$$min\_size_d = \lfloor out\_size \cdot \frac{speed_d}{\sum_{i=1}^{D} speed_i} \rfloor \tag{1}$$

The processing speed metrics ($speed_d$) are derived from empirical measurements on real devices, following the methodology used in EdgeFlow [4] and DeepSlicing [22]. We construct device-specific linear regression models using benchmarking data collected with PyTorch profiling tools [13], capturing the relationship between tensor size and execution time for each device and partition direction.

This linear approximation is justified by the observed near-linear scaling behavior of execution time with respect to tensor size in one-direction partitioning scenarios. The resulting models provide a reliable estimate of relative processing speeds across heterogeneous devices without requiring explicit hardware modeling as previous methods also demonstrated.

The minimum size constraint ensures a computational load distribution proportional to each device's processing capability in an ideal scenario that ignores communication overhead. Partition sizes smaller than this minimum threshold would likely create performance imbalances, resulting in suboptimal execution as detailed in the optimization formulation in Section 4.4.

The maximum size of a partition is identical for all devices and ensures that no single device monopolizes the entire workload.

$$max\_size_d = out\_size - (D - 1) \tag{2}$$

For example, in a scenario with output size 224 and four devices, the first device cannot process the entire range from 0 to 224, as this would leave no meaningful work for the remaining devices. Similarly, the last device cannot start its partition at index 0 for the same reason.

4.3.2 Search Space Pruning with Adaptive Thresholding. After the minimum and maximum bounds are established, we implement adaptive thresholding according to Equation (3):

$$search\_size_d = (max\_size_d - min\_size_d) \cdot thrs \tag{3}$$

Where $thrs$ represents a tunable threshold parameter between 0 and 1 that controls the aggressiveness of search space pruning. This approach allows us to restrict the search on the most promising region between the minimum size (based on computational balance) and maximum size (constrained by work distribution requirements).

The threshold mechanism provides a configurable trade-off between exploration thoroughness and computational efficiency. Higher threshold values explore a larger portion of the partition size range, potentially discovering more optimal configurations at the cost of increased search time. Lower threshold values concentrate on partitions closer to the computational balance point, accelerating the search process while potentially missing some optimization opportunities.

Importantly, this thresholding approach preserves the potential for layer fusion by maintaining a reasonable range of partition sizes rather than forcing a single fixed partitioning strategy. Exploring diverse partitioning configurations remains essential for identifying fusion opportunities across multiple layers.

4.4 Optimization Problem

This subsection describes our methodology for identifying optimal partitioning strategies through a formalized optimization problem. The approach incorporates three key factors: computational workload distribution, inter-device data transfer, and management of overlapped computational regions. The following notations are used during this subsection:

- Indices s and e denote the start and end of partition intervals (e.g., for interval $[3, 7)$, $s = 3$ and $e = 7$)

- $p_{l,d,s/e}$: l layer index, d device, s start/e end of partition.

4.4.1 Computational Workload Estimation. The computational workload distribution across heterogeneous devices is calculated by Equation (4) by computing a vector of estimated execution times $\mathbf{t}$ for each device d, based on the assigned output size and device-specific processing capabilities. We utilize the same $speed_d$ values as introduced in Section 4.3.

$$t_d = \lfloor out\_size / speed_d \rfloor \quad \forall d \in [1, D] \tag{4}$$

Equation (5) identifies the computational bottleneck by determining t_{comp}, the maximum execution time among all devices. This represents the slowest device that others must wait for to maintain synchronization.

$$t_{comp} = \max(\mathbf{t}) \tag{5}$$

Example 4.1. Consider two partitioning strategies: $p_{l,1} = [0, 3)[3, 6)[6, 9)$ and $p_{l,2} = [0, 2)[2, 3)[3, 9)$, where all devices have equal processing capabilities. In $p_{l,2}$, two devices ($device_1$ and $device_2$) complete significantly earlier but remain idle while waiting for $device_2$. Conversely, in $p_{l,1}$, the slowest device finishes earlier than in $p_{l,2}$. Our model penalizes $p_{l,2}$ to avoid excessive idle time across devices.

4.4.2 Data Transfer Overhead. After determining the computational costs, we calculate the data transfer overhead between adjacent layers.

Equations (6) and (7) calculate the data transfer requirements at the start and the end of each partition interval, applying constraints to prevent illogical negative transfer values.

$$data_{l,d,s} = \begin{cases} 0 & \text{if} \quad p_{l+1,d,s} > p_{l,d,s} \\ \min(p_{l,d,s}, p_{l+1,d,e}) - p_{l+1,d,s} & \text{o/w} \end{cases} \tag{6}$$

$$data_{l,d,e} = \begin{cases} 0 & \text{if} \quad p_{l+1,d,e} < p_{l,d,e} \\ p_{l+1,d,e} - \max(p_{l+1,d,s}, p_{l,d,e}) & \text{o/w} \end{cases} \tag{7}$$

Example 4.2. For this example, we consider two layers division partitions at $d = 1$: where $p_{l,1} = [3, 6)$ and $p_{l+1,1} = [2, 3)$, the application of Equations (6) and (7) yields $data_{l,1,s} = 1$ and $data_{l,1,e} = 0$.

The total data transfer time (t_{data}) is calculated by summing individual transfer requirements and dividing by the bandwidth capacity (BW_n) for each device:

$$t_{data} = \sum_{d=1}^{D} (data_{l,d,s} + data_{l,d,e}) / BW_d \tag{8}$$

4.4.3 Overlapped Computation Overhead. As explained in Section 4.2, certain operators require duplicate computation across multiple devices when partitioned. Equation (10) calculates this overlapped computation overhead (t_{ov}), accounting for processing speed differentials across devices.

$$r_{ov} = \begin{cases} p_{l,d,e} - p_{l,d+1,s} & d = 1 \\ p_{l,d,s} - p_{l,d-1,e} & d = D \\ p_{l,d,s} - p_{l,d-1,e} + p_{l,d,e} - p_{l,d+1,s} & 1 < d < D \end{cases} \tag{9}$$

$$t_{ov} = \sum_{d=1}^{D} \frac{r_{ov}}{speed_d} \tag{10}$$

Note in Equation (9), that for the first and the device ($d = 1$ and $d = D$, respectively), only one of the sides of the partitions are considered, as these devices are not positioned between two other devices.

4.4.4 Total Execution Time. The total execution time ($score$) for a candidate partitioning strategy is given by the sum of the components:

$$score = t_{comp} + t_{data} + t_{ov} \tag{11}$$

It is important to note that our model does not explicitly capture the overlap between computation and data transfer operations. Instead, these components are aggregated into a unified cost score by summing computation time, data transfer time, and overlapped computation time. This formulation is not intended to precisely model real execution time, but to provide a consistent and computationally efficient scoring function for comparing candidate partitioning strategies.

While more sophisticated pipelining and overlap modeling could further refine performance estimation, they would introduce significant complexity and hardware-specific dependencies. In contrast, the proposed score preserves relative differences between candidate configurations, enabling effective guidance during the search process. The resulting score is therefore used as the primary metric in Section 4.5 to rank and select partitioning strategies, with lower scores indicating more efficient candidates.

4.5 Search Space Exploration

This section describes our approach to efficiently exploring the vast partitioning search space while identifying near-optimal solutions. We present a tree-based search algorithm enhanced with pruning techniques and dynamic programming to significantly reduce computational complexity.

4.5.1 Tree-Based Representation. We employ a hierarchical tree structure to represent the partitioning search space, enabling the application of Alpha-Beta pruning techniques. Figure 6 illustrates this representation, where each layer node encodes a specific division of the layer's tensor along the selected dimension. Each layer node contains three essential components:

- **Layer:** Represents the CNN layer being partitioned, with boundaries from 0 to the layer's dimension size in the selected direction.
- **Device:** Indicates the specific device assignments for a partition segment. For example, in Figure 6, device 1 the left-most assigned partition [2, 5).
- **Partition Interval:** Defines the start and end indices of each partition segment within the tensor dimension.

The organization of tree nodes within the tree significantly impacts search efficiency. To leverage the convex nature of our optimization function, we order tree nodes to represent monotonically increasing partition sizes when traversing from left to right. For example, in Figure 6, partitions progress from [0, 3) to [0, 4), incrementing the partition size. We apply a similar ordering to starting points within each device. This structured ordering enables more effective pruning during the search.

4.5.2 Evaluation Methodology. Our algorithm employs a depth-first search to reach terminal nodes, followed by backpropagation to evaluate partitioning decisions. The evaluation of layer partitioning configurations is governed by Equation (12), which addresses both current layer optimization and interactions with adjacent layers.

$$r_l = \begin{cases} opt(div_{0,out}, div_{1,in}) & l = 0 \\ opt(div_{l,out}, \emptyset) + r_{l-1} & l = L-1 \\ opt(div_{l,out}, div_{l+1,in}) + r_{l-1} & 0 < l < L-1 \end{cases} \tag{12}$$

Equation (12) comprises two key components. The first component, opt, evaluates the relationship between the input division, $div_{l+1,in}$, (derived from the previous layer's output) and the current layer's output division, $div_{l,out}$. In Figure 6, this corresponds to analyzing partitions from Layers 1 and 2 using the optimization formulation from Section 4.4. If no previous layer exists ($l = L - 1$), the data transfer component from Equation (8) is omitted.

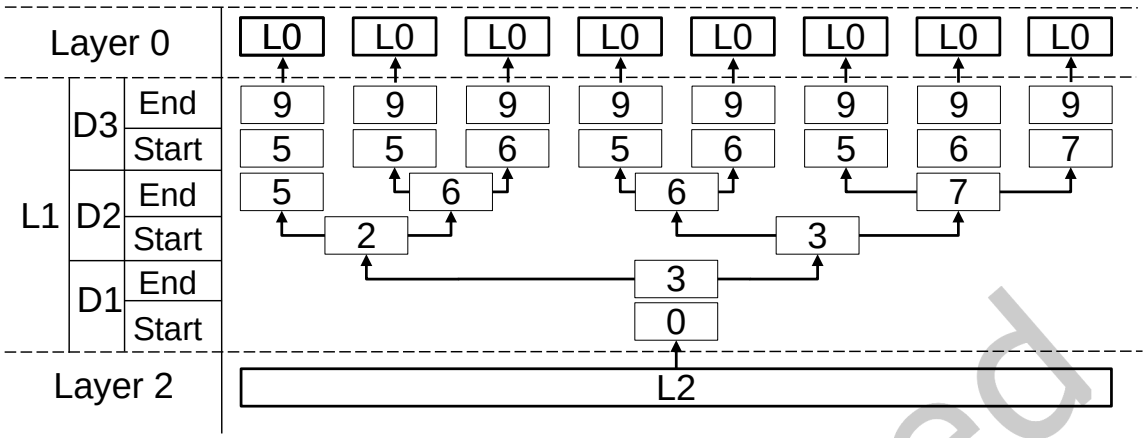


Fig. 6. Search space exploration example. Tree nodes prefixed with 'L' represent layer structures, 'D' indicates device index, and Start/End denote partition interval boundaries.

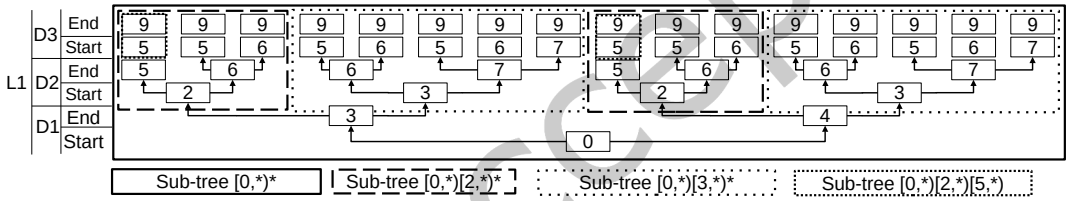


Fig. 7. Visual representation of sub-tree reuse in dynamic programming optimization. Identical sub-trees (highlighted) are computed once and reused across branches with matching preceding conditions.

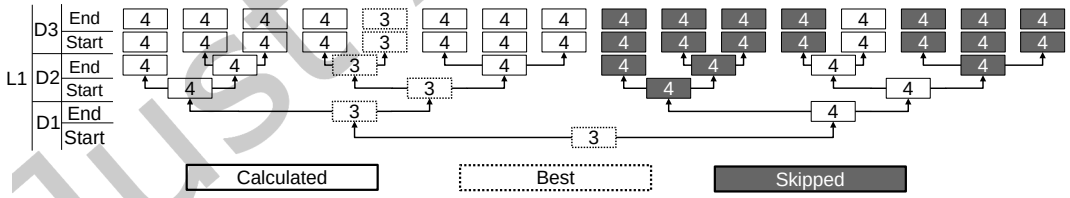


Fig. 8. Visual representation of Alpha-Beta branch pruning. Values inside nodes represent optimization function results, with shaded nodes indicating pruned search paths that cannot improve upon the current best solution.

The second component r considers the impact of the current partitioning decision on subsequent layers. For example, when evaluating Layer 1, we incorporate results from Layer 0. This recursive formulation enables the discovery of layer fusion opportunities across multiple layers, not just adjacent pairs.

The evaluation results for each unique partitioning configuration are stored to prevent redundant computation, significantly reducing the overall search time.

4.5.3 Dynamic Programming Optimizations. To accelerate search space exploration, we implement two dynamic programming techniques:

Memoization of Evaluation Results: As noted in Section 4.5.2, identical partitioning configurations may be encountered multiple times during the search. We cache evaluation results for each unique configuration, retrieving them when the same configuration is encountered again rather than recalculating.

Sub-Tree Reuse: This technique identifies and reuses pruned branches across different parts of the search tree when all preceding node values match. Figure 7 illustrates this concept, where branches sharing the same previous starting node values can reuse identical sub-trees. This approach preserves the convexity properties established by our node ordering, ensuring consistent evaluation patterns while eliminating redundant exploration.

4.5.4 Alpha-Beta Pruning Implementation. The efficiency of our search algorithm is substantially enhanced through a modified Alpha-Beta pruning approach. By leveraging the sorted nature of our search tree (Section 4.5.1), we can eliminate entire branches that cannot improve upon the best solution found so far.

As illustrated in Figure 8, after evaluating several partitioning possibilities, our algorithm identifies and prunes branches that cannot yield better results, significantly reducing search time. The pruning propagates up through previous branches to the layer's root node, maximizing the elimination of unnecessary computation.

It is important to note that when evaluating multiple layers with Equation (12), this pruning approach might occasionally exclude potentially superior solutions in specific edge cases. However, this represents a necessary trade-off to make the vast search space computationally feasible. In practice, our experimental results demonstrate that the algorithm consistently finds high-quality solutions despite this theoretical limitation.

4.6 Fully connected and Small-Output Division Strategy

Once all layers that can be partitioned along the height or width dimensions have been processed, ADaPS-FC performs a final evaluation step to determine whether channel-based partitioning is beneficial for the remaining layers. Consistent with the formulation in Section 4.4, the framework evaluates a cost score under both the current configuration and a configuration augmented with channel partitioning. Channel division is applied only if it improves the overall cost score; otherwise, the original height-based configuration is retained. This ensures that channel partitioning is introduced only when it provides a measurable improvement over the baseline ADaPS [9].

For operators that support both Channel Input and Channel Output partitioning, ADaPS-FC selects the partition type based on communication reuse potential and the cost score. In particular, a Channel Output partition can be paired with a subsequent Channel Input partitioned layer to reuse the same channel decomposition and avoid redundant inter-device communication.

This design is inspired by prior work such as DeeperThings [23], which demonstrated that combining Channel Input and Channel Output partitioning can reduce inter-layer communication. However, differs fundamentally from DeeperThings [23], which determines channel-based partitioning using a static heuristic comparing weight size and spatial feature map size. In contrast, ADaPS-FC does not rely on a fixed structural threshold. Instead, partitioning decisions are driven by a hardware-aware cost score that jointly models communication cost, computation cost, and synchronization overhead under the observed system configuration. This allows ADaPS-FC to adapt partitioning decisions to both model structure and runtime execution characteristics, rather than relying solely on tensor size comparisons.

5 Evaluation

In this section, the framework is evaluated based on the following criteria: inference time, throughput, node utilization, and memory footprint.

Table 4. Hardware Configurations for Experimental Evaluation

Platform	#Devices	Processor	Memory	Eth. Interface
Raspberry Pi Zero 2 W	1	Cortex-A53 (1GHz)	512MB	Wi-Fi 4G
Raspberry Pi 4B	2	Cortex-A72 (1.5GHz)	8 GiB	LAN / Wi-Fi 5G
Raspberry Pi 5B	4	Cortex-A76 (2.4GHz)	8 GiB	LAN / Wi-Fi 5G

5.1 Experimental Setup

To rigorously evaluate ADaPS-FC, we conducted experiments on resource-constrained edge devices using the hardware configurations detailed in Table 4. Our implementation leverages PyTorch [13] and its Gloo backend for distributed communication primitives. For comparing inference latency across frameworks, we used PyTorch's built-in profiling tools to ensure accurate timing measurements.

We selected a diverse set of CNN architectures to thoroughly evaluate our framework: AlexNet [6], VGG16 and 19 [16], MobileNetV2 [15], InceptionV3 [18], HardcoreNAS [12], ResNet50 and 200 [3] and ResNeXT100 [20]. These networks were chosen to represent a spectrum of complexity, from relatively simple architectures to those with challenging multi-path structures. All network models were obtained from ONNX Zoo [1].

To explore different network bandwidths, we test each configuration at 10 Mbps and 100 Mbps Wi-Fi and LAN connections. This allowed us to evaluate the frameworks' sensitivity to communication constraints. We compared ADaPS-FC against two state-of-the-art decentralized frameworks: BBGraP [10], EdgeFlow [4], ADAPS [9] and ADAPS + DeeperThings [17]. All reported results are normalized to EdgeFlow's performance for consistent comparison. The partitioning strategies were computed once per network and hardware configuration, with computation times ranging from a few seconds to under five minutes on a single CPU core. All reported performance results represent averages across 100 independent runs on real hardware.

5.2 Homogeneous Environment

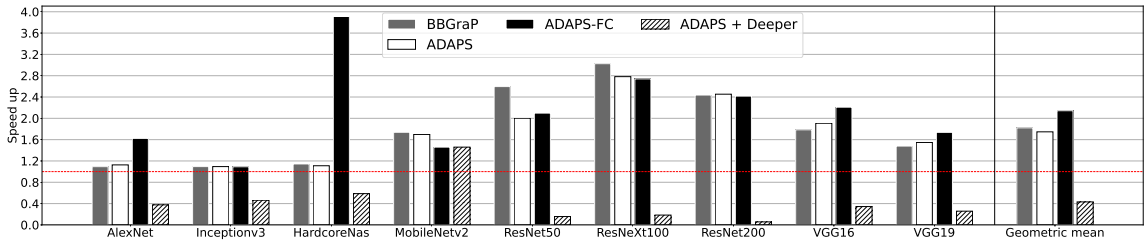
For the two-node homogeneous setup, we use two Raspberry Pi 5 devices in the first experiment (Fig. 9) and two Raspberry Pi 4 devices in the second experiment (Fig. 10). In both experiments, our method demonstrates consistent behavior and delivers strong performance. This is particularly evident on *HardcoreNAS*, which contains a sub-path in the middle of the network with spatial dimensions of size 1. Since this part of the model cannot be divided along height or width, ADaPS-FC is especially effective, yielding a substantial performance advantage in both setups.

However, as shown in Fig. 9 and Fig. 10, ADaPS-FC underperforms BBGraP in the 10 Mb/s bandwidth configuration for architectures with strong shortcut dependencies, such as *ResNet* and *MobileNet*. In low-bandwidth scenarios, the communication overhead introduced by maintaining consistency across shortcut paths can outweigh the benefits of partitioning and layer fusion. This result highlights that the effectiveness of ADaPS-FC depends on both the network structure and the available bandwidth, with greater benefits observed in architectures with fewer shortcut constraints and in moderate to high bandwidth regimes.

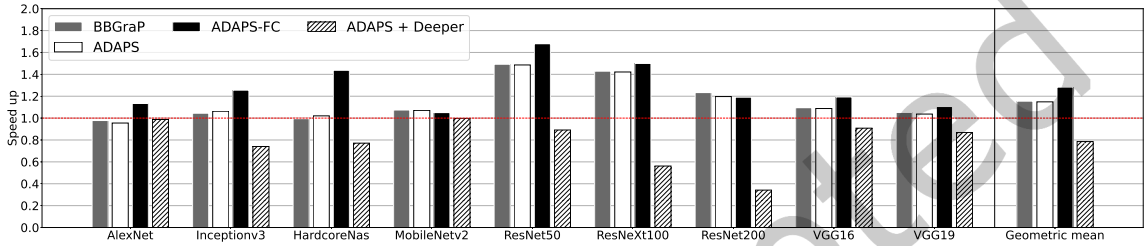
The *Deeper* method also shows improvements relative to the other baselines, though its performance remains significantly lower overall. Models that are more linear and contain heavy operations that cannot be partitioned, such as *AlexNet*, *VGG16*, and *VGG19*, also benefit from our method, as the splitting strategies of prior approaches are less suited to such architectures.

Overall, for the two-node homogeneous configuration, our method is 2× faster than the previous method EdgeFlow and approximately 1.3× faster than ADaPS and BBGraP.

For the four-node homogeneous configuration, we use four Raspberry Pi 5 devices to evaluate performance at scale (Figure 11). As shown in the results, the performance of ADaPS-FC and ADaPS becomes much closer in this setting. The main reason for this reduced gap is the increased amount of data transfer required when performing

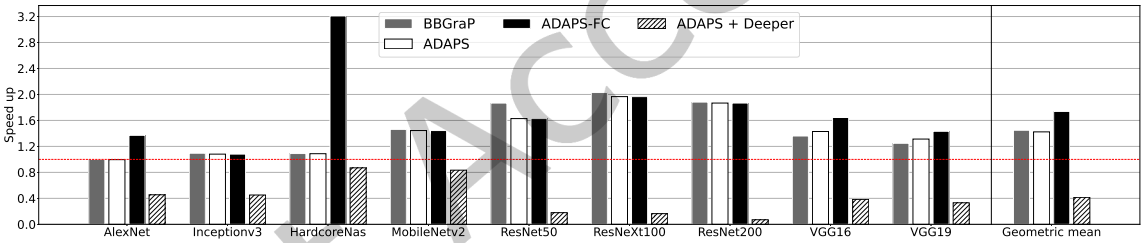


(a) 10Mb/s bandwidth

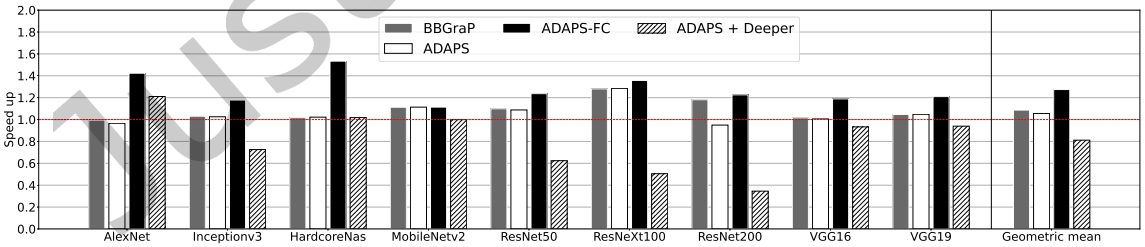


(b) 100Mb/s bandwidth

Fig. 9. Homogeneous two devices Rbp5 speed up normalized to EdgeFlow

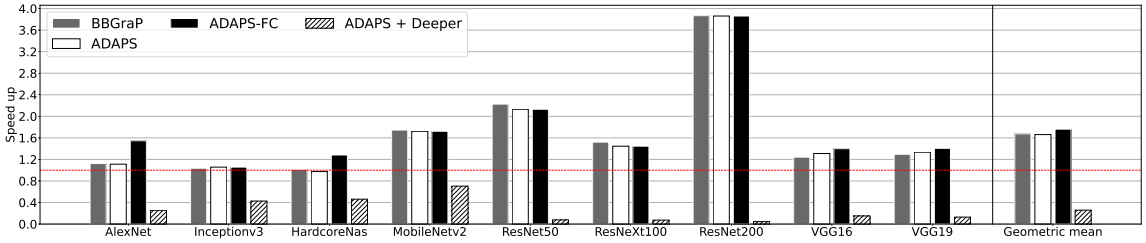


(a) 10Mb/s bandwidth

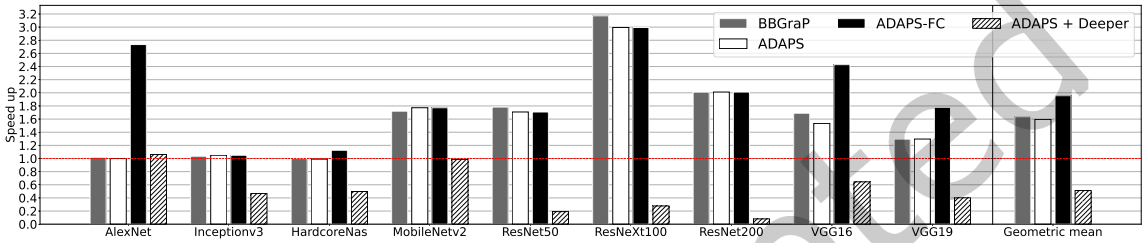


(b) 100Mb/s bandwidth

Fig. 10. Homogeneous two devices Rbp4 speed up normalized to EdgeFlow



(a) 10Mb/s bandwidth



(b) 100Mb/s bandwidth

Fig. 11. Homogeneous 4 devices Rbp5 speed up normalized to EdgeFlow

channel partitioning. This observation also helps explain why the *Deeper* method performs particularly poorly here: since channel partitioning demands significant data movement across all nodes, its overhead becomes prohibitive.

In this setup, model weights are generally not the primary bottleneck. However, for networks such as *AlexNet*, *VGG16*, and *VGG19*, the later layers cannot be divided along the height dimension. When channel-wise division is applied instead, it still leads to a measurable improvement in inference time.

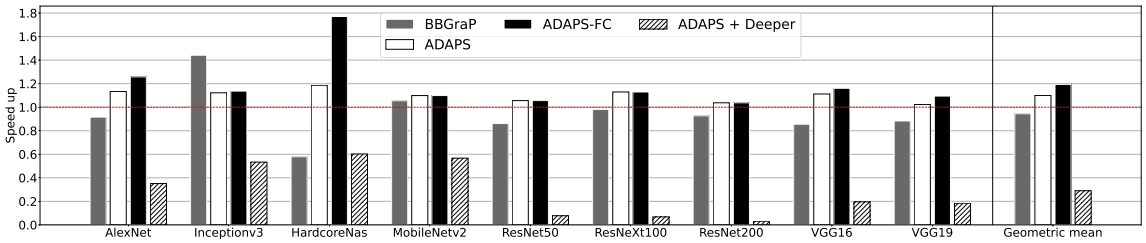
As explained previously, *HardcoreNAS* benefits substantially due to the presence of a sub-path in the middle of the network whose output has height and width equal to 1. Because this part of the architecture cannot be partitioned spatially, channel-wise partitioning becomes the only viable strategy, and our method handles this case particularly effectively, resulting in the observed performance gains.

5.3 Heterogeneous Environment

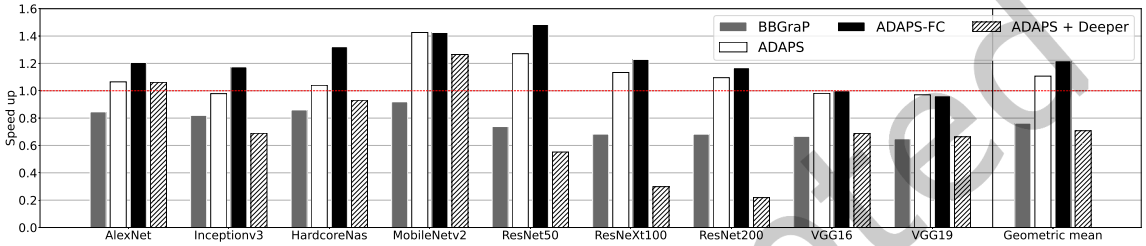
For heterogenenous devices starting with two-node configuration (Figure 12), we use two Raspberry Pi 5 devices and two Raspberry Pi 4 devices to examine how parameter asymmetry affects performance in a two-node setting. In this scenario, our method achieves overall improvements because the layers that cannot be divided still benefit significantly from being executed across multiple nodes rather than on a single device. Additional gains come from the layers that are divided, where ADaPS provides a much better partitioning strategy than the alternative methods.

The *Deeper* method shows some improvement when the bandwidth is increased to 100 Mb/s compared to 10 Mb/s, as its high data-transfer requirements demand a faster network. However, even with improved bandwidth, this increase is insufficient for Deeper to outperform the other approaches.

BBGraP also falls behind in this heterogeneous setup due to its poor ability to generate effective partitions when the participating devices are not homogeneous.



(a) 10Mb/s bandwidth



(b) 100Mb/s bandwidth

Fig. 12. Heterogeneous two devices Rbp5-4 speed up normalized to EdgeFlow

For the four-node heterogeneous setup, we use two Raspberry Pi 5 devices and two Raspberry Pi 4 devices to examine how parameter asymmetry affects performance in a 4-node configuration (Figure 13). As shown in the results, the performance of ADaPS-FC and ADaPS becomes much closer in this scenario. The more modest improvements observed in this configuration are largely attributable to the increased number of nodes. Although adding more devices substantially increases total computational capability, it also shifts the performance balance toward communication. In this regime, data transfer dominates the execution time, naturally constraining the achievable speedups.

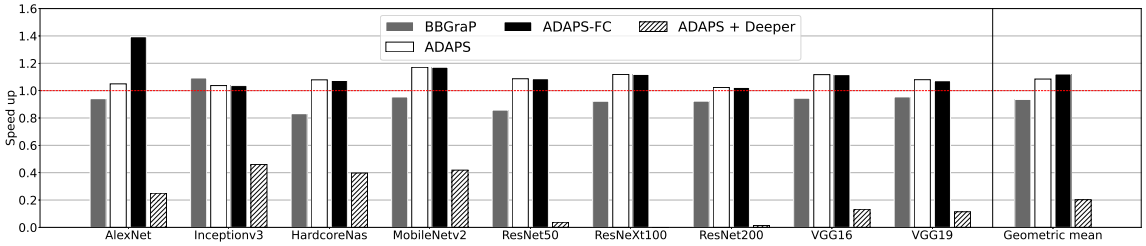
While ADaPS and our method exhibit similar performance under these conditions, the other baselines fall behind because they cannot adequately adapt to the data-reduction requirements. Like in the homogeneous experiment this limitation is most evident in the *Deeper* method, which involves the highest amount of data exchange: it must retrieve data from every node and redistribute it, resulting in substantial performance penalties.

This lack of adaptability becomes even more apparent when varying the available bandwidth. As the bandwidth decreases, the performance of the competing methods degrades significantly, whereas our method continues to maintain a clear advantage.

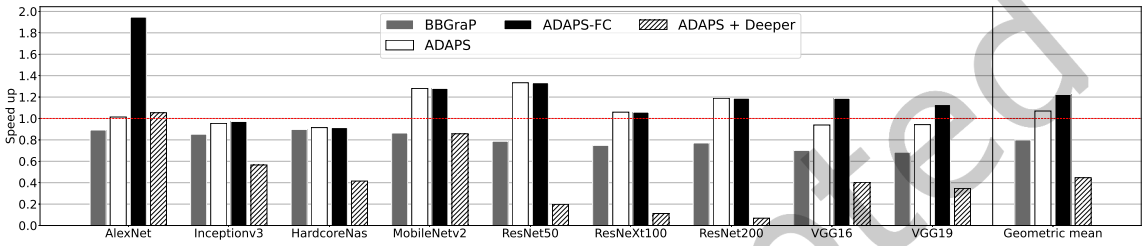
Finally, in terms of overall average performance, ADaPS-FC achieves a speedup of approximately 1.2× over EdgeFlow and 1.1× over the previous ADaPS method.

5.4 Raspberry Pi Zero 2W Experiments

In this section, we evaluate how operating under more constrained hardware conditions affects the performance of our method. All experiments were conducted using a Raspberry Pi Zero 2 W, which supports only Wi-Fi connectivity; therefore, all reported results assume full capacity bandwidth. The main limiting factor in this setup is the reduced RAM available on the Zero 2 W.



(a) 10Mb/s bandwidth



(b) 100Mb/s bandwidth

Fig. 13. Heterogeneous four devices two Rbp5 and two Rbp4 speed up normalized to EdgeFlow

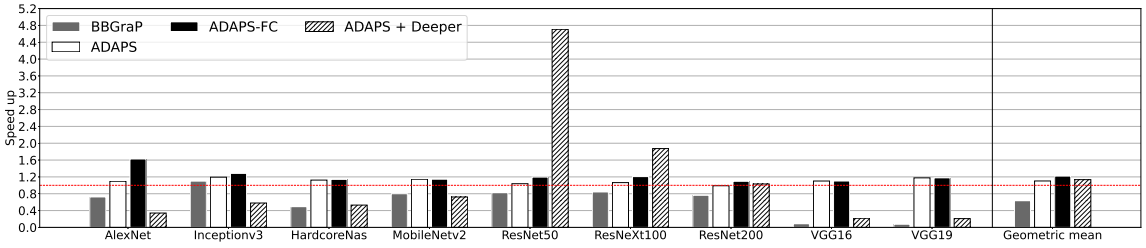
As shown in the first experiment (Fig. 14(a)), comparing the Raspberry Pi 5 and the Zero 2 W, our algorithm outperforms the alternatives in all models except *ResNet50*, *ResNeXt100* and *ResNet200*. In these cases the weight size becomes the primary bottleneck much earlier than in other architectures. Under these conditions, the *Deeper* method performs better because it alleviates the impact of weight transmission. As observed previously, however, when weight transmission is not the dominant factor, *Deeper* yields significantly poorer performance.

For *ADaPS-FC*, the balanced use of height division and weight division in the later layers enables consistent improvements over other baselines. Its worst results remain comparable to ADaPS. The cases in which ADaPS-FC performs similarly to ADaPS occur when the data transmission required is larger than the communication savings provided by splitting, diminishing the performance advantage. Furthermore, due to the more limited resources in this setup, *BBGraP* exhibits considerably lower performance compared to the earlier results.

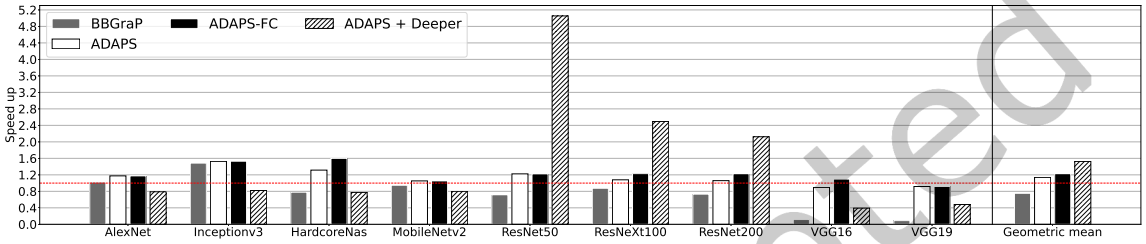
The second experiment (Fig. 14(b)), performed using a Raspberry Pi 4 combined with a Raspberry Pi Zero 2 W, although *Deeper* shows an advantage only in a small set of corner-case architectures (ResNet50, ResNeXt100, and ResNet200), our method consistently surpasses all alternatives in this cases and on the rest of the models. A clear example appears in *InceptionV3*, *HardcoreNAS*, where our method achieves up to a 1.6× speedup. This improvement arises because the previous methods fail to generate effective partitions when hardware resources become scarce. This effect is especially evident in both experiments for *BBGraP* on *VGG16* and *VGG19*, where the large memory footprint of these models leads to significantly degraded performance relative to other architectures.

5.5 Model Evaluation

In this section we evaluate how reliable is our model score compared to the real runtime. To evaluate the accuracy of the proposed performance model, we analyze the relationship between the predicted score and the measured execution time. Figure 15 show this comparison for VGG16 distributed across two and four devices.

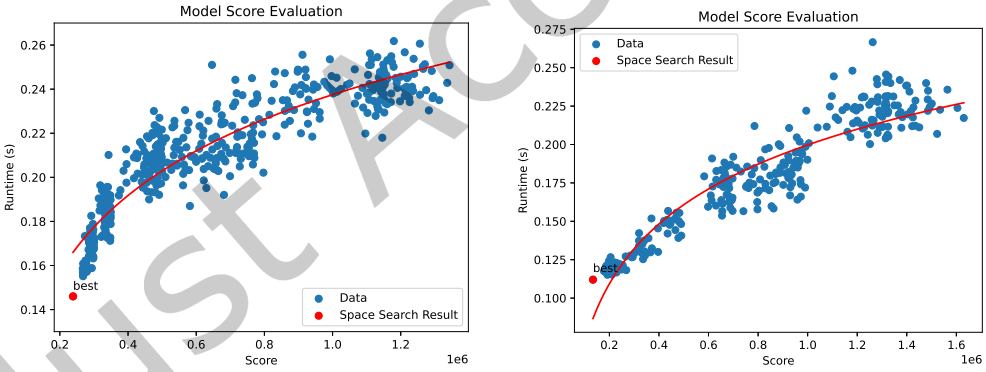


(a) Heterogeneous two devices RbpZero2W and Rbp5 using Wi-Fi speed up normalized to EdgeFlow



(b) Heterogeneous two devices RbpZero2W and Rbp4 using Wi-Fi speed up normalized to EdgeFlow

Fig. 14. Heterogeneous two device experiment with a big difference on capabilities



(a) two devices

(b) four devices

Fig. 15. Model score compared to runtime on VGG16

As observed, there is a clear correlation between the predicted score and the actual runtime: configurations with lower scores consistently result in lower execution times. This trend holds across both device configurations, indicating that the model effectively captures the relative cost of different partitioning strategies. In particular, we observe a strong Pearson correlation coefficient of **0.88** for two devices and **0.95** for four devices, confirming that the model provides a reliable ordering of candidate configurations.

Although the model does not aim to predict the exact execution time, it provides a consistent ranking of candidate configurations. This property is sufficient for guiding the search space exploration, enabling ADaPS-FC to identify high-performance partitioning strategies without an exhaustive search.

5.6 Pruning results

The search space for assigning partitions of a neural network layer to D heterogeneous devices is combinatorially large and cannot be exhaustively explored; heuristics are therefore essential. The presented ADaPS-FC technique first estimates a partitioning based on device performance, followed by aggressive pruning. The initial partitioning provides a strong initial guess. A threshold parameter thr defines the search range around this estimate: given an initial guess p, the search spans $[p - thr \cdot p, p + thr \cdot p]$. While the threshold parameter allows for tuning the trade-off between search quality and runtime, the resulting space remains exponentially large.

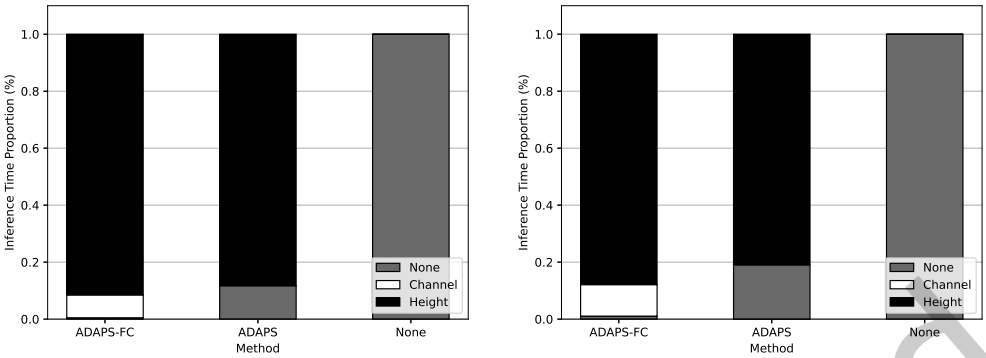
Pruning further reduces this space to a manageable size. Table 5 and Table 6 quantifies the impact of pruning on the search space for two and four devices respectively, across different networks and thresholds. Pruning becomes increasingly effective with a larger threshold and more devices, as it eliminates infeasible candidates early in the search tree. As shown in Section 4.5.4, ADaPS-FC halts exploration of subtrees that cannot yield better solutions, exploiting the tendency of high-quality solutions to occur early. The pruning benefit multiplies across layers, scaling exponentially with the network depth.

Table 5. Space search reduction evaluation for two devices (All results are in log base 10)

Model	Threshold (%)	Ours	Dynamic Programming	Original
VGG16	5	2.57	2.97	16.77
	10	3.07	3.55	29.47
	15	3.35	3.87	35.85
ResNet50	5	2.69	2.91	14.45
	10	3.07	3.4	27.72
	15	3.31	3.73	38.37
InceptionV3	5	2.54	2.54	7.68
	10	2.78	3.07	13.76
	15	2.94	3.34	17.87

Table 6. Space search reduction evaluation for four devices (All results are in log base 10)

Model	Threshold (%)	Ours	Dynamic Programming	Original
VGG16	5	4.89	8.29	90.6
	10	5.44	9.701	137.83
	15	5.70	10.42	174.63
ResNet50	5	4.54	7.61	79.02
	10	5.17	9.02	146.07
	15	5.59	9.74	175.48
InceptionV3	5	4.25	7.6	42.49
	10	4.84	9.01	68.13
	15	5.3	9.73	79.51



(a) two devices (b) four devices

Fig. 16. Division proportion comparison between methods on VGG16

5.7 Division Impact

To further understand the impact of the partitioning strategies, we analyze the proportion of operations assigned to each division type. Figure 16 show the normalized distribution of layers executed with no partitioning, channel partitioning, and height partitioning for different methods.

As observed, ADaPS-FC makes greater use of channel and height partitioning compared to the baseline approaches, which rely more heavily on height division only. In particular, channel partitioning accounts for a significant portion of the workload in ADaPS-FC, highlighting its importance in enabling efficient parallel execution in layers that height division is not possible.

We also observe differences between the two-device and four-device configurations. In the four-device case, there is a higher proportion of non-partitioned layers, as some layers are too small to be efficiently distributed across more devices. As a result, the remaining partitionable workload is more frequently assigned to channel partitioning, increasing its relative proportion in ADaPS-FC.

These results help explain the performance improvements observed in the speedup analysis. Increasing the proportion of operations that can be effectively partitioned, ADaPS-FC achieves better resource utilization and reduced execution time.

This behavior is consistent with the speedup results shown in Section 5.2, where methods with higher utilization of channel and height partitioning achieve greater performance gains.

6 Conclusions and Future work

In this work, we presented ADaPS-FC, a novel framework for distributing CNN inference workloads across heterogeneous edge devices. By extending previous approaches, our framework supports flexible partitioning along height, width, and channel dimensions, enabling efficient execution even for fully connected and small-output layers that traditional methods often leave to only one node to perform.

ADaPS-FC combines a search space exploration with layer fusion and adaptive channel partitioning, taking into account device heterogeneity and communication overhead.

Experimental evaluations across multiple CNN architectures and heterogeneous hardware configurations demonstrate that ADaPS-FC consistently improves inference performance compared to prior approaches. By carefully balancing computation and communication, our framework achieves notable reductions in latency, particularly for workloads where conventional partitioning strategies fail to scale effectively.

Overall, ADaPS-FC provides a practical, scalable solution for low-latency inference on resource-constrained devices, making it well-suited for end-users and small-to-medium enterprises that require efficient deployment of convolutional neural networks on commodity hardware.

Acknowledgments

The authors thank the anonymous reviewers and the associate editor for their thorough feedback and constructive suggestions, which helped improve the quality of this paper. This work was funded, in part, by the Korean National Research Foundation (grant no. 21A20151113068 – BK21 Plus for Pioneers in Innovative Computing - Dept. of Computer Science & Engineering, SNU), by the Ministry of Trade, Industry and Energy (MOTIE) and the Korea Evaluation Institute of Industrial Technology (KEIT) (grant no. 10077609), by the Ministry of Science and ICT (MSIT) (grant no. RS-2023-00302083), and by the EU Horizon program within the f-inference project (grant no. 101298393). ICT at Seoul National University provided research facilities for this study.

References

[1] Junjie Bai, Fang Lu, Ke Zhang, et al. 2019. ONNX: Open Neural Network Exchange. https://github.com/onnx/onnx.

[2] Myeonggyun Han, Jihoon Hyun, Seongbeom Park, Jinsu Park, and Woongki Baek. 2019. MOSAIC: Heterogeneity-, Communication-, and Constraint-Aware Model Slicing and Execution for Accurate and Efficient Inference. In *2019 28th International Conference on Parallel Architectures and Compilation Techniques (PACT)*. 165–177. doi:10.1109/PACT.2019.00021

[3] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2015. Deep Residual Learning for Image Recognition. *CoRR* abs/1512.03385 (2015). arXiv:1512.03385 http://arxiv.org/abs/1512.03385

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

[5] Donald E. Knuth and Ronald W. Moore. 1975. An analysis of alpha-beta pruning. *Artificial Intelligence* 6, 4 (1975), 293–326. doi:10.1016/0004-3702(75)90019-3

[6] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems*, F. Pereira, C.J. Burges, L. Bottou, and K.Q. Weinberger (Eds.), Vol. 25. Curran Associates, Inc.

[7] Nicholas D Lane, Sourav Bhattacharya, Petko Georgiev, Claudio Forlivesi, Lei Jiao, Lorena Qendro, and Fahim Kawsar. 2016. Deepx: A software accelerator for low-power deep learning inference on mobile devices. In *2016 15th ACM/IEEE International Conference on Information Processing in Sensor Networks (IPSN)*. IEEE, 1–12.

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

[9] Jaume Mateu Cuadrat and Bernhard Egger. 2025. ADaPS: Adaptive Data Partitioning to Parallelize CNN Inference on Resource-Constrained Hardware. In *Proceedings of the 26th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools for Embedded Systems* (Seoul, Republic of Korea) *(LCTES '25)*. Association for Computing Machinery, New York, NY, USA, 16–27. doi:10.1145/3735452.3735532

[10] Jaume Mateu Cuadrat, Daon Park, and Bernhard Egger. 2022. A Black-Box Graph Partitioner for Generalized Deep Neural Network Parallelization. In *International Conference on the Economics of Grids, Clouds, Systems, and Services*. Springer, 132–140.

[11] Svetlana Minakova, Erqian Tang, and Todor Stefanov. 2020. Combining task-and data-level parallelism for high-throughput CNN inference on embedded CPUs-GPUs MPSoCs. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 20th International Conference, SAMOS 2020, Samos, Greece, July 5–9, 2020, Proceedings 20*. Springer, 18–35.

[12] Niv Nayman, Yonathan Aflalo, Asaf Noy, and Lihi Zelnik-Manor. 2021. HardCoRe-NAS: Hard Constrained diffeRentiable Neural Architecture Search. arXiv:2102.11646 [cs.LG]

[13] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, Alban Desmaison, Andreas Kopf, Edward Yang, Zachary DeVito, Martin Raison, Alykhan Tejani, Sasank Chilamkurthy, Benoit Steiner, Lu Fang, Junjie Bai, and Soumith Chintala. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. In *Advances in Neural Information Processing Systems 32*. Curran Associates, Inc., 8024–8035. http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

[14] Ronald L Rivest. 1987. Game tree searching by min/max approximation. *Artificial Intelligence* 34, 1 (1987), 77–96.

[15] Mark Sandler, Andrew Howard, Menglong Zhu, Andrey Zhmoginov, and Liang-Chieh Chen. 2019. MobileNetV2: Inverted Residuals and Linear Bottlenecks. arXiv:1801.04381 [cs.CV]

[16] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).
[17] Rafael Stahl, Alexander Hoffman, Daniel Mueller-Gritschneder, Andreas Gerstlauer, and Ulf Schlichtmann. 2021. DeeperThings: Fully distributed CNN inference on resource-constrained edge devices. *International Journal of Parallel Programming* 49 (2021), 600–624.
[18] Christian Szegedy, Sergey Ioffe, Vincent Vanhoucke, and Alexander Alemi. 2017. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 31.
[19] K Vanishree, Anu George, Srivatsav Gunisetty, Srinivasan Subramanian, Shravan Kashyap, and Madhura Purnaprajna. 2020. CoIn: Accelerated CNN Co-Inference through data partitioning on heterogeneous devices. In *2020 6th International Conference on Advanced Computing and Communication Systems (ICACCS)*. IEEE, 90–95.
[20] Saining Xie, Ross B. Girshick, Piotr Dollár, Zhuowen Tu, and Kaiming He. 2016. Aggregated Residual Transformations for Deep Neural Networks. *CoRR* abs/1611.05431 (2016). arXiv:1611.05431 http://arxiv.org/abs/1611.05431
[21] Liekang Zeng, Xu Chen, Zhi Zhou, Lei Yang, and Junshan Zhang. 2020. Coedge: Cooperative dnn inference with adaptive workload partitioning over heterogeneous edge devices. *IEEE/ACM Transactions on Networking* 29, 2 (2020), 595–608.
[22] Shuai Zhang, Sheng Zhang, Zhuzhong Qian, Jie Wu, Yibo Jin, and Sanglu Lu. 2021. Deepslicing: collaborative and adaptive cnn inference with low latency. *IEEE Transactions on Parallel and Distributed Systems* 32, 9 (2021), 2175–2187.
[23] Zhuoran Zhao, Kamyar Mirzazad Barijough, and Andreas Gerstlauer. 2018. Deepthings: Distributed adaptive deep learning inference on resource-constrained iot edge clusters. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 37, 11 (2018), 2348–2359.

Received 1 December 2025; revised 3 April 2026; accepted 19 May 2026