Partitioning and Placement of Deep Neural Networks on Distributed Edge Devices to Maximize Inference Throughput

Arjun Parthasarathy
Crystal Springs Uplands School
Email: aparthasarathy23@csus.org

Bhaskar Krishnamachari
University of Southern California
Email: bkrishna@usc.edu

Abstract—Edge inference has become more widespread, as its diverse applications range from retail to wearable technology. Clusters of networked resource-constrained edge devices are becoming common, yet no system exists to split a DNN across these clusters while maximizing the inference throughput of the system. We present an algorithm which partitions DNNs and distributes them across a set of edge devices with the goal of minimizing the bottleneck latency and therefore maximizing inference throughput. The system scales well to systems of different node memory capacities and numbers of nodes. We find that we can reduce the bottleneck latency by 10x over a random algorithm and 35% over a greedy joint partitioning-placement algorithm. Furthermore we find empirically that for the set of representative models we tested, the algorithm produces results within 9.2% of the optimal bottleneck latency.

I. INTRODUCTION

Deep Neural Networks (DNNs) have greatly accelerated machine learning across different disciplines, such as Computer Vision [5] and Natural Language Processing [21]. Edge Inference is becoming an increasingly popular field with multiple facets [33], as sensor-driven computation in IoT systems necessitates DNN inference in the field. IoT applications for edge inference range from retail to wearable technology [4], [6].

The edge can come in multiple configurations [19], [28], and there are multiple approaches to facilitate edge inference. For cloud-edge hybrid inference, one such approach is model compression [11], which deals exclusively with DNN optimization but does not address the system's runtime configuration. In this paper, we focus on clusters of resource-constrained edge devices. These *edge clusters* are becoming increasingly common due to their low-cost and scalability at the edge [24]. Unlike a cloud data center, the edge brings system resource limitations and communication bottlenecks between devices.

With this in mind, we address the following problem: **How can we take advantage of multi-device edge clusters to enable high-performance DNN inference while respecting computational resource constraints and taking into account the heterogeneity of communication links?**

To partition a deep learning model, we first split the model into components that are executed sequentially. Each partition is assigned to a different edge device, and once each
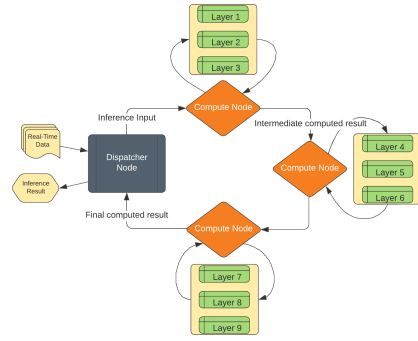

Fig. 1: Partitioning and Distributing a Model Across Edge Devices to Create an Inference Pipeline

node performs inference with its piece of the model, that intermediate inference result is sent to the next node with the corresponding partition in the sequence. This inference pipeline is shown in Figure 1.

In an edge cluster, although we have a lower computational power in each node, we can take advantage of this inference pipelining to increase system throughput. Since each node can perform inference with its partition individually, prior nodes in the pipeline can send their finished inference results to the subsequent nodes in the pipeline and accept new batches.

We define the *throughput* metric of a system as the number of inference cycles it can perform per unit time. As we showed in our previous work DEFER [23], we can achieve higher throughput with distributed edge inference as opposed to inference on a single device because of pipelining. The throughput is defined as the reciprocal of the *bottleneck latency*. For nodes $[k] = \{1, 2, \ldots, k\}$, the bottleneck latency β is defined as

$$S = \{k \in [k] \mid c_k, \gamma_k\}$$
$$\beta = \max_{s \in S} s \tag{1}$$

where c_k is the compute time of the operations on node k, and γ_k is the communication time between node $k-1$ and k. We use ResNet50 [12], which is a representative model for our use case. On a Raspberry Pi 4, the inference speed was found to be 225 ms [25]. Next, we found the amount

of data transferred between each layer of the model. On average, 10.2 Mbits of data was transferred between layers. Given an average WiFi bandwidth of 6 Mbps for a low-end edge network, this gives us a communication time of $1.7s$. This is 7.5x slower than the compute time. In reality, many models are larger than ResNet50 and will therefore be split across devices, so each device will have less operations to execute. This means that communication time will outweigh compute time as the bottleneck. Therefore, we can simplify the expression for bottleneck latency to the following:

$$\beta = \max_{k \in [k]} \gamma_k \qquad (2)$$

Since throughput is defined as $\frac{1}{\beta}$, by minimizing the bottleneck latency we maximize inference throughput. Additionally, we assume that all nodes are homogeneous in RAM. If the devices are not the same capacity, then the algorithm will take the smallest memory capacity across all nodes in the cluster, and take that as the capacity of each node. In this paper, we primarily analyze image and text models due to their prevalence on the edge for visual analytics applications [22], [34].

Our main contribution in this paper is a novel partitioning and placement algorithm for DNNs across a cluster of edge devices distributed spatially within the same WiFi network. The algorithm finds the candidate partition points, finds the optimal partition sizes to transfer the least amount of data, and finds the arrangement of nodes with the highest bandwidth. Together, these aim to minimize the resulting bottleneck latency according to the throughput metric. We found that our algorithm results in a 10x improvement over a random partitioning/placement algorithm, and a 35% reduction in bottleneck latency for systems with 50 compute nodes. We empirically observe an average approximation ratio of 1.092 for the bottleneck latency (i.e. it is 9.2% more than the optimal bottleneck latency, on average).

II. Related Work

Early works on the topic of partitioning DNN models divided them into head and tail models with the former distilled to enable running on a resource-constrained device and reduce data transfer [20]. Some prior works on DNN edge inference mathematically perform DNN model slicing by layer [35], [36], after calculating layer impact during the training stage; these do not account for communication demands on the edge. Others abstract model layers into certain "execution units," [7], [17] which they then choose to slice based on certain resource requirements. Li *et al.* [16] regressively predict a layer's latency demand and optimize communication bandwidth accordingly. DeeperThings [29] performs layer fusion on CNNs to optimize data transfer. These works are optimized for a hybrid edge-cloud pipeline and do not address the demands of a cluster of edge devices. Couper [13] uses a similar partitioning scheme to minimize inter-partition data transfer, but does not address the communication bottleneck

associated with an edge cluster. Hu *et. al* [14] optimize the partitioning of a CNN onto a set of devices by taking compute time as a bottleneck, while employing compression to deal with communication constraints, and do not consider placement. Our paper builds on and differentiates itself from these works by addressing the bandwidth limitation of an edge cluster, and aims to maximize inter-node bandwidth during the placement stage to minimize bottleneck latency.

III. Partitioning and Placement Algorithm

We are given two graphs:
1) An unweighted DAG G_m representing the computation graph of a DNN, where each vertex represents a layer in the model. This DAG can be found using common ML libraries such as Tensorflow [1] and Keras [8].
2) A weighted complete graph G_c representing the communication graph of a cluster of homogeneous physical compute nodes, where each vertex represents a physical compute node and each edge represents the bandwidth between those nodes. The graph is complete because we assume that these edge devices will communicate over the same WiFi network.

Our goal is to optimally partition the model and place these partitions on a set of edge devices. We do so as follows.

A. Converting a Complex DAG to a Linear DAG

First, we need to distill G_m into a linear DAG. The vertices where it is possible to partition the model are called "candidate partition points." We illustrate this in Figure 2.

For $v \in V$, edges $e \in E$ and source vertex s of G_m, find the longest path from s to v. This can be done by topologically sorting the DAG and for each vertex in the resulting list, relaxing each neighbor of that vertex. We call the length of this longest path the *topological depth* of that vertex in the graph. Let $LP(v)$ denote the length of longest path from s to v.

To verify that all paths from vertex v_{prev} go through vertex v, use a modified DFS by recursing on the incident edges of each vertex. If we encounter a vertex with a greater topological depth than v, return false. If we reach vertex v, return true. Let $AP(v_{prev}, v)$ denote the result of this algorithm.

Given the previously found candidate partition point p_{k-1} and the current vertex u, the next candidate partition point $p_k = u$ iff:
1) $LP(u) \neq LP(v) \forall v \in \{V - u\}$
2) $AP(p_{k-1}, u) = $ true

with $p_0 = s$.

The time complexity of LP is $O(V + E)$. AP runs in polynomial time by returning upon reaching a vertex with a greater topological depth. Therefore, this algorithm runs in polynomial time.

Figure 2 shows the candidate partition points at certain sections of the DAG of ResNet50 [12] and InceptionResNetV2 [30]. Each rectangle represents a model layer in the DAG.

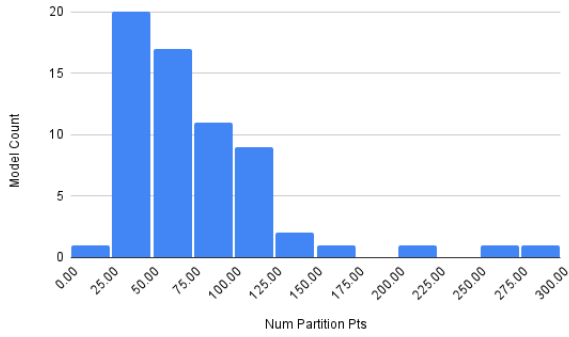Fig. 2: Partition points for ResNet50 and InceptionResNetV2 models

Fig. 3: Histogram of Number of Candidate Partition Points

We then calculated the number of partition points for the set of Keras pretrained models. As shown in Figure 3, almost all the models have at least 25 candidate partition points. There are some model architectures, like NASNet [37], which do not allow partitioning under our scheme.

As shown in Figure 4, NASNet cannot be partitioned because there is no single point that splits the model into a distinct execution unit that does not have any dependencies to a previous or subsequent layer. If we run our LP algorithm, we

Fig. 4: Portion of NASNet's layer DAG

find that there is no single layer that has distinct topological depth from other layers. We found that 64 of the 66 (97 %) pretrained Keras models [15] could be partitioned under our scheme, and only the NASNet variants could not.

B. Optimal model partitioning and placement

Our goal is to maximize throughput of the system. As previously discussed, this means we need to minimize the bottleneck latency. Latency is defined as $\frac{\text{data}}{\text{bandwidth}}$. Given a tuple of partition points P_{opt}, their transfer sizes T, and a set of bandwidths B between compute nodes, the latency between each set of compute node is defined as

$$\gamma_k = \frac{T_{opt,k}}{B_k} \forall 0 \le k < |P_{opt}| \tag{3}$$

The bottleneck latency for the system is then given by Equation 2. For the purposes of explanation, we separate the problems of optimizing the partitions (thereby optimizing transfer size) and optimizing placement (thereby optimizing bandwidth between nodes). We show empirically that this results in the the smallest bottleneck latency. In Section V, we compare this formulation to an algorithm that tries to jointly optimize transfer size and bandwidth.

1) Finding optimal partitions: Our heuristic for finding optimal partitions is the "transfer size" of the partition; i.e how much data will be transferred from that partition to the next. Given the tuple of candidate partition points $P = (p_0, p_1, \ldots, p_k)$, we now need to find a set of model partitions which minimizes the sum of transfer sizes. Assuming a batch size of 1, the transfer size t_k of candidate partition point p_k is defined as

$$t_k = \frac{\eta}{\lambda} \tag{4}$$

Given a floating point array representing the output of the model layer l_k (which is the same layer given by the candidate partition point p_k), η represents the size of that array.

$\lambda \approx 1.44 * 2.1$ represents the total compression ratio given by multiplying the average ZFP compression ratio [18] by the average LZ4 compression ratio [9].

To better illustrate our algorithm, we classify the transfer size t_k into 3 transfer size classes ("low", "medium", or "high") based on the distribution of the transfer sizes.

$$C = \{L, M, H\} \quad t_k \subseteq C \tag{5}$$

The optimal set of partitions is the scheme which minimizes the sum of the transfer sizes of said partitions.

Let G_p represent a DAG, where each vertex is represented by a possible partition. The vertices are defined as follows:

$$p_i, p_{i+1}, \ldots, p_j\}) < \kappa \mid \{p_i, p_{i+1}, \ldots, p_j\}\} \\ \forall 0 \le i < |P_{opt}|, 0 \le j < |P_{opt}| - i \tag{6}$$

The set of vertices represents every possible contiguous subarray of candidate partition points, where $\omega(P)$ finds whether the memory use of partition P is within the memory
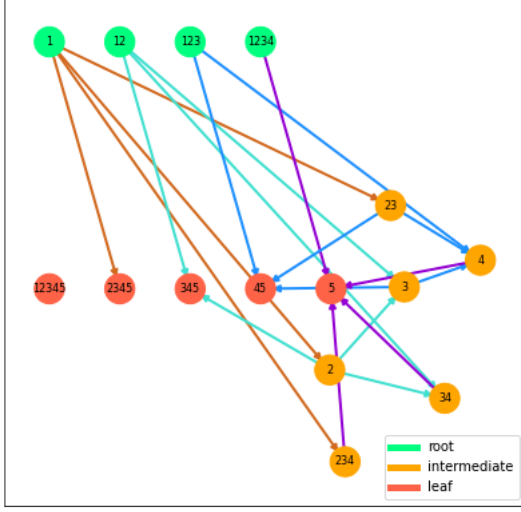
241

Fig. 5: Example partition graph, where the partition points are $P = \{1, 2, 3, 4, 5\}$

capacity κ of the compute node. We quantize the models using TFLite [32] quantization to reduce their memory footprint. However, when calculating the memory footprint of a partition, we do not consider this quantization. This means that we are conservative on partition size and in turn provide extra space on each device for the memory overhead from containerization. Each partition is a set of layers that fall between the partition points p_i and p_j.

The set of edges is defined as follows:

$$E = \{(u, v) \in V, \rho(u_{|u|-1}) = \rho(v_0 - 1) \mid (u, v)\} \quad (7)$$

The function $\rho(v)$ finds the index of element v in P_{opt}. There is an edge between vertices if the last partition point of u's partition is adjacent in P to the first partition point of v's partition. For example, if $u = [1, 2]$, $v = [3, 4]$, and $P = (1, 2, 3, 4)$, then (u, v) is an edge. Each edge has a weight $w(u, v)$ which corresponds to its transfer size class.

Figure 5 shows an example partition graph, where edges that are the same color will have the same weight. In the figure, "root" vertices have in-degree 0, "leaf" vertices have out-degree 0, and "intermediate" vertices have neither.

Algorithm 1 finds the shortest path in the graph from a root to a leaf. Since edges which bridge the same candidate partition points (and have the same color as shown in Figure 5) will have the same subsequent paths, we can memoize the shortest path. On line 2, we store a map on which tells us for each candidate partition point what the shortest path is from that point. Using memoization, Algorithm 1 takes $O(N)$ to find the shortest path, but $O(N^2)$ to construct the partition

---

Algorithm 1 Optimal Partitioning

// Map to store memoized paths
$pathFrom \leftarrow$ NEW-MAP()
procedure MIN-COST-PATH(G, v)
    if $v.children = \emptyset$ **then**
        return $v, 0$
    end if
    $partitionLastLayer \leftarrow v[v.length - 1]$
    if $partitionLastLayer \notin pathFrom$ **then**
        $paths \leftarrow []$
        for $c \in v.children$ **do**
            $path, cost \leftarrow$ MIN-COST-PATH(G, c)
            $paths \leftarrow$ APPEND($paths, (path, cost)$)
        end for
        $pathFrom[partitionLastLayer] =$ MIN($paths$)
    end if

    $best \leftarrow pathFrom[partitionLastLayer]$
    $minPath, minCost \leftarrow best$
    $chosenNode \leftarrow minPath[0]$
    // Path starting at v and going to a leaf
    $newPath \leftarrow$ APPEND($[v], ...minPath$)
    $newCost \leftarrow minCost + w(v, chosenNode)$
    return $newPath, newCost$
end procedure

procedure PARTITION(G)
    $roots \leftarrow$ GET-ROOT-VERTICES(G)
    for $r \in roots$ **do**
        $path, cost \leftarrow$ MIN-COST-PATH(G, r)
        $paths \leftarrow$ APPEND($paths, (path, cost)$)
    end for
    $minPath, minCost \leftarrow$ MIN($paths$)
    return $minPath$
end procedure

$\Theta \leftarrow$ PARTITION(G_p)

---

graph. Therefore the runtime of Algorithm 1 is $O(N^2)$, where N is the number of nodes.

Let Θ represent the set of chosen partitions. For each subarray in Θ, we take the last element of the subarray, add that to the list of partition points Q, and add its corresponding transfer size to the list S. The resulting list is then sorted based on the topological depth of each partition point, so that the partitions are executed in the order they appear in the model.

2) Finding optimal model placement: With the set of optimal partitions Q and their corresponding transfer sizes S, we now need to "match" them to the vertices of G_c. We know from Equation 5 that $S \subseteq C$. Let $c(e)$ return the bandwidth class of a given edge of G_c. We use the following threshold function to classify each edge:

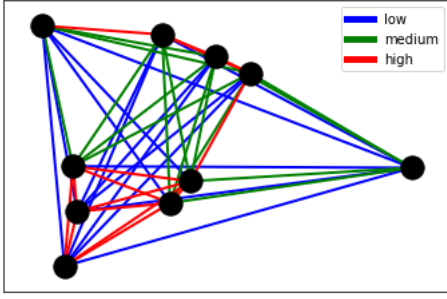Fig. 6: Example communication graph with different bandwidth classes

$$\tau(X,t) = \left\{ \begin{array}{ll} c(e) = C_{\arg X - 1}, & \text{if } e < t \\ c(e) = X, & \text{if } e \geq t \end{array} \right\} \forall e \in E_c \quad (8)$$

If the edge is greater than or equal to the threshold, it will be classified as class X, otherwise it will be classified as the class in C right below X. In order for our algorithm to work, we set the number of transfer size classes equal to the number of bandwidth classes.

Figure 6 shows an example communication graph, where the nodes are colored black and different bandwidth classes of edges are shown.

Given the array of transfer sizes S and array of communication graph edges E_c, the lower bound on bottleneck latency we can achieve is given by Theorem 1.

Theorem 1: The lowest bottleneck latency we can achieve is:

$$\min(\beta) = \frac{\max S}{\max E_c} \quad (9)$$

Therefore, if we achieve $\min(\beta)$, then we have found the optimal minimum bottleneck latency.

We prove Theorem 1 as follows:

Given the highest transfer size $(\max S)$, then it must be matched with the highest bandwidth $(\max E_c)$ to have the lowest bottleneck latency. There are two cases in which the system would have another bottleneck latency:

1)
$$\beta = \frac{\max S}{e} \forall e \in E_c - max(E_c) \quad (10)$$

2)
$$\beta = \frac{s}{e}$$
$$\forall \{s \in S - max(S),$$
$$e \in E_c - max(E_c) \mid \beta \geq \frac{\max S}{\max E_c}\} \quad (11)$$

In Equation 10, the latency of the system would be higher than 9, since the transfer size is being matched with a lower bandwidth edge. In Equation 11, some other transfer size s and bandwidth e may result in a higher bottleneck latency, in which case Equation 9 still holds. Therefore, Theorem 1 holds.

We run tests in Section V to see how often we get this optimal solution. Algorithm 2 performs the matching between

S and G_c to try to reach the optimal latency as outlined above. Let N represent the array of nodes that we choose from G_c, with length $|S|$.

Algorithm 2 Finding K-Paths

procedure SUBGRAPH-K-PATH(X, k, s, u)
    // Sort by weight in descending order
    $edgeList \leftarrow$ SORT($G_c, \{e \in E_c \mid w(e)\}, reverse$)
    $low \leftarrow 0$
    $high \leftarrow edgeList.length$
    $bestPath \leftarrow []$
    while $low < high$ **do**
        $median \leftarrow \frac{(low+high)}{2}$
        $\tau(X, edgeList[median])$
        $\tau(X, threshold)$
        // Induced subgraph of G_c w/ class X edges
        $G_c^X \leftarrow \{E_c^X = \{e \in E_c \wedge c(e) = X \mid e\} \mid V_c^X, E_c^X\}$
        $result \leftarrow$ K-PATH(G_c^X, k, s, u)
        if $result =$ FALSE **then**
            $low \leftarrow median + 1$
        else
            $high \leftarrow median$
            $bestPath \leftarrow result$
        end if
    end while
    for $N \in bestPath$ **do**
        DEL(G_c, N)
    end for
end procedure

In algorithm 2, we use the color-coding k-path algorithm [2], which finds a path of length k (where k is the number of vertices) in G_c^X if a k-path exists and does so in polynomial time if $k < \log(|V^X|)$. As we show in Section V, this is possible using node memory capacities which reflect real-world devices. We use a binary search to find the maximum threshold for which a k-path exists. On line 3, we sort in descending order so that we can find the maximum viable edge-weight threshold with a binary search. As N starts to be filled in, the k-paths have to be found between certain nodes in order for N to be a contiguous path of nodes. We modify the k-path algorithm to start at s and stop once it reaches u. We make the algorithm more efficient by stopping a particular iteration if we reach u before we have a path of length k. If s is $null$, find any find any k-path that ends at u. Similarly, if u is $null$, find any k-path that starts at s.

Algorithm 3 performs the k-path matching of partitions onto vertices of G_c.

FIND-SUBARRAYS() on line3 returns a list of subarrays of a certain class, by iterating over the list of transfer sizes S.

Lines 2-13 match paths for all bandwidth classes, starting with class H. On line 8, $startV$ represents the vertex before

---

Algorithm 3 K-Path Matching

1: **procedure** K-PATH-MATCHING(S, C)
2:     **for** $X \in C$ **do**
3:         $x\_paths \leftarrow$ FIND-SUBARRAYS(S, X)
4:         $x\_paths \leftarrow$ SORT$(x\_paths, \{p \in x\_paths \mid p.length\})$
5:         **for** $i \leftarrow 0$ **to** $x\_paths.length$ **do**
6:             $startIdx \leftarrow$ INDEX-OF$(x\_paths[i][0], S)$
7:             $x\_len \leftarrow x\_paths.length$
8:             $startV \leftarrow N[startIdx]$
9:             $endV \leftarrow N[startIdx + x\_paths[i].length + 1]$
10:            $path \leftarrow$ SUBGRAPH-K-PATH$(X, x\_len + 1, startV, endV)$
11:            $N[startIdx : startIdx + path.length] \leftarrow path$
12:         **end for**
13:     **end for**
14: **end procedure**

---

the current k path. If this is equal to $null$, meaning that the algorithm hasn't reached the iteration of finding the previous k-path, then SUBGRAPH-K-PATH will find a path that starts at any vertex. Similarly, on line 9 $endV$ represents the vertex of the current k-path. If this is equal to $null$, meaning that the algorithm hasn't reached the iteration of finding that subsequent k-path, then SUBGRAPH-K-PATH will find a path that ends at any vertex.

By starting with the longest H-subarrays and working to the shortest L subarrays, we are greedily finding the best bandwidth paths to match with the highest transfer size terms of S. We continue this process until we have found k-path matchings for all subarrays of S. In some cases, a high number of bandwidth classes will prevent the algorithm from returning a result, because it has very few edges to choose from during each iteration of the matching. In this case, we can re-run the algorithm with fewer bandwidth classes.

IV. EVALUATION METHODOLOGY

We simulated a set of randomly placed edge devices using a random complete graph. For each evaluation, we created a random complete graph by drawing the positions of the nodes from a uniform distribution with the range $x, y \in (-150, -1) \cup (1, 150)$ to simulate a WiFi router with a range of 150m. Between each set of nodes, we calculated the edge weight using Shannon's capacity equation, assuming that the SNR decreases proportionally to the inverse square of the device's distance from the router.

$$r(x,y) = \log_2\left(1 + \frac{a}{(\sqrt{x^2+y^2})^2}\right) = \log_2\left(1 + \frac{a}{x^2+y^2}\right)$$
$$x, y \in (-B, -1) \cup (1, B)$$
(12)

In Equation 12, we found $a = 283230$ by assuming that the bandwidth at 80 m from the router was 5.5 Mbps, which matches the characteristics of a low-power edge network. We exclude $x, y \in (-1, 1)$ to satisfy the domain of Equation 12, and to simplify the creation of our geometric graphs for our simulations. From a practical standpoint, this means that we assume that no devices will be within 1m of the router.

We used the following configuration to test the algorithm:

1) **Model**: MobileNetV2 [27], EfficientNetB1 [31], ResNet50, or InceptionResNetV2, chosen for their diversity in model topologies and memory sizes. Due to space constraints, we only show ResNet50 and InceptionResNetV2 in some results figures.
2) **Number of Nodes**: 5, 10, 15, 20, or 50 randomly placed edge devices.
3) **Number of Bandwidth Classes**: 2, 5, 8, 11, 14, 17, or 20 bandwidth classes, which provide granularity in how to classify the transfer sizes and edge bandwidths.
4) **Node Memory Capacity**: 64, 128, 256, or 512 MB of RAM for a compute node.

For each test, we used a different random communication graph generated using the procedure above. With each algorithm result, we then calculated the bottleneck latency according to Equation 3. The resulting bottleneck latency from each configuration of model, node capacity, number of nodes, and number of bandwidth classes was run 50 times and averaged.

We compare the resulting bottleneck latency of our algorithm to that of the following two algorithms:

1) **Random Algorithm**: Select a random node and a random partition that can be accommodated on that node.
2) **Joint-Optimization Algorithm**: Let Q and N represent the optimal set of partitions and optimal arrangement of nodes, respectively, chosen under this algorithm For each node n, do the following:
   a) At each step choose the partition with the smallest transfer size that will fit within the node. Add this partition to the set of chosen partitions p.
   b) Starting at n, find the neighbor in the communication graph whose edge e has the highest bandwidth, and add that to the path of chosen nodes c. Then, find the highest bandwidth edge from e, and so on.
   c) Compare the bottleneck latency found with p and c to the smallest bottleneck found with all nodes n thus far, and update Q and N with p and c if the current bottleneck is smaller.

For each of these algorithms, we used the same configuration and methodology as above to find the bottleneck latency. These algorithms don't use bandwidth classes, so we didn't need to include that as part of the configuration.

V. RESULTS

Figure 7 shows a color map of the resulting bottleneck latency based on the factors described in Section IV. In Figure 7, the color map was only generated for the node capacities which were too small for the models to fit on a single device of that capacity. All models were able to fit on a single
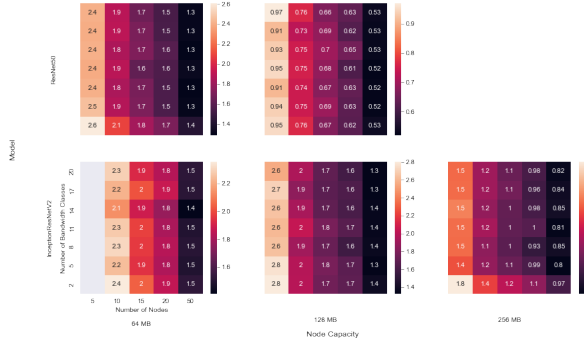
Fig. 7: Color Map of Bottleneck Latency (s) based on Model, Node Capacity, Number of Nodes, and Number of Bandwidth Classes - Optimal Partitioning/Placement
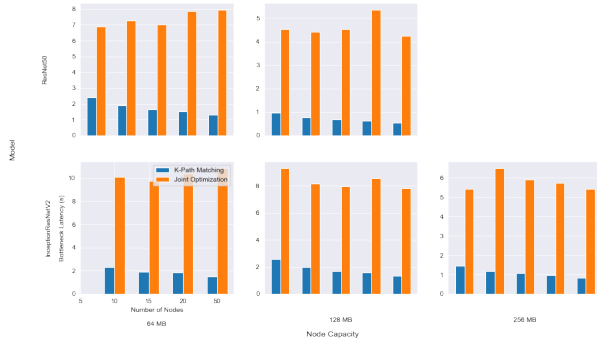

Fig. 9: Comparison of Algorithm 3 with Joint Optimization - based on Model, Node Capacity, Number of Nodes

Fig. 8: Comparison of Algorithm 3 with Random Algorithm - based on Model, Node Capacity, Number of Nodes

Fig. 10: Histogram of Average Approximation Ratio for Keras Pretrained Models

512 MB device. The lack of bottleneck latency values for InceptionResNetV2 with 5 nodes and 64MB node capacity indicates that the model could not be partitioned with these physical constraints. For each model, the lowest bottleneck latency for a given node capacity comes from the combination of the most number of bandwidth classes and number of nodes. The lowest bottleneck latency comes with the highest node capacity. These results follow from the fact that a larger node and number of nodes allows the partitioning algorithm to have greater choice in selecting the smallest transfer sizes. Similarly, a high number of bandwidth classes allows the placement algorithm to better perform the k-path matching.

In Figure 8, the optimal algorithm reduces bottleneck latency by $\approx 10x$ on average for this selection of models. The difference is the smallest for ResNet50, with the optimal algorithm producing a $\approx 2x$ lower bottleneck latency. The models with the greatest variance in transfer size will result in the largest difference in bottleneck latency between the optimal random algorithms. Overall, we see that the optimal algorithm produces a significant reduction in bottleneck latency compared to the random algorithm.
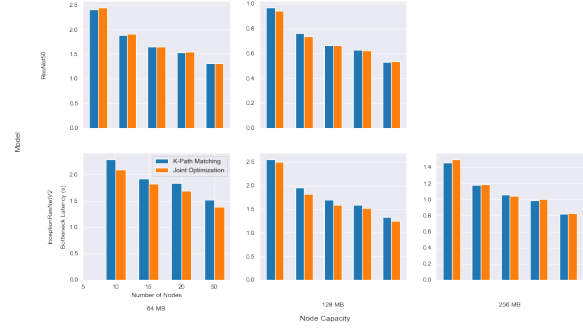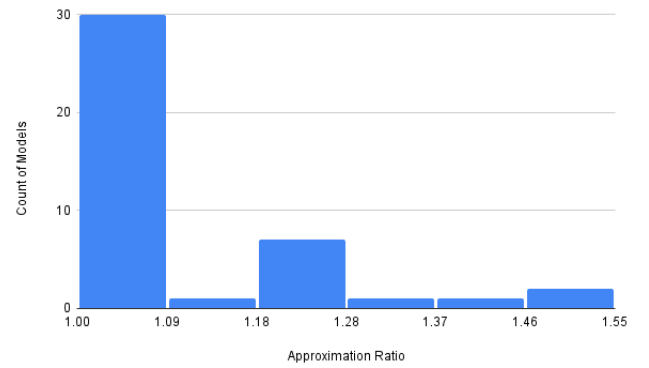
In Figure 9, the joint optimization algorithm tends to

perform better for a smaller number of nodes. Since each of these algorithms use the same optimal partitioning logic, we can only compare the models based on their differing placement logic. As the number of nodes increases, our k-path algorithm performs better. This makes sense, because the difference in the greedy strategy of the joint optimization algorithm and the matching strategy of our algorithm only becomes more apparent as the communication graph grows bigger and there are more options for node paths. In particular, for 50 nodes, our algorithm outperforms the joint optimization algorithm by 35%. We hypothesize that this trend would continue for more complex models which have a greater number of candidate partition points and a greater variance in transfer size, necessitating the k-path matching strategy to minimize bottleneck latency.

We then ran our algorithm 1000 times for the set of Keras pretrained models. We used a configuration of 50 nodes and 64 MB node memory capacity. For each trial, we then divided the resulting bottleneck latency by the optimal bottleneck latency as given by Theorem 1. We took the average of this ratio across all trials for each model. The results are presented in Figure 10. We see that the bottleneck latencies for 75% of these models are within 9% of the optimal

solution. The average approximation ratio across all these models is ≈ 1.092 or within 9.2% of the optimal bottleneck latency.

VI. Conclusion

We have presented a framework to partition and place a model across a set of resource-constrained edge devices, with the goal of maximizing inference throughput. Additionally, we show that for different models and node configurations, we can outperform a greedy joint-optimization algorithm. We further show empirically that our algorithm is within 9.2% of the optimal bottleneck latency for the models we tested.

A. Future Work

With minor edits, we could extend our framework to work with geographically-distributed edge devices for a truly scalable edge inference solution.

With software changes, we could potentially run the average image model on a cluster of micro-controllers. We could use RiotOS [26] without any containerization and perform optimizations to run with limited device memory. Some devices we could potentially take advantage of are the Raspberry Pi Pico [10] and Arduino Uno [3].

VII. Acknowledgements

We would like to acknowledge the helpful input and pointers provided by Prof. Anil Vullikanti from the University of Virginia, particularly in directing us to the color-coding k-path algorithm.

References

[1] ABADI, M., ET AL. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. Software available from tensorflow.org.

[2] ALON, N., YUSTER, R., AND ZWICK, U. Color-coding. *Journal of the ACM (JACM) 42*, 4 (1995), 844–856.

[3] ARDUINO. Arduino uno. https://store-usa.arduino.cc/products/arduino-uno-rev3, 2003.

[4] BISWAS, A., JAIN, A., ET AL. Survey on edge computing–key technology in retail industry. In *Computer Networks and Inventive Communication Technologies*. Springer, 2021, pp. 97–106.

[5] CHAI, J., ZENG, H., LI, A., AND NGAI, E. W. Deep learning in computer vision: A critical review of emerging techniques and application scenarios. *Machine Learning with Applications 6* (2021), 100134.

[6] CHEN, J., AND RAN, X. Deep learning with edge computing: A review. *Proceedings of the IEEE 107*, 8 (2019), 1655–1674.

[7] CHO, E., YOON, J., BAEK, D., LEE, D., AND BAE, D.-H. Dnn model deployment on distributed edges. In *International Conference on Web Engineering* (2021), Springer, pp. 15–26.

[8] CHOLLET, F., ET AL. Keras. https://keras.io, 2015.

[9] COLLET, Y. Lz4. https://github.com/lz4/lz4, 2011.

[10] FOUNDATION, R. P. Raspberry pi pico. https://www.raspberrypi.com/products/raspberry-pi-pico/, 2021.

[11] GHOLAMI, A., KIM, S., DONG, Z., YAO, Z., MAHONEY, M. W., AND KEUTZER, K. A survey of quantization methods for efficient neural network inference. *arXiv preprint arXiv:2103.13630* (2021).

[12] HE, K., ET AL. Deep residual learning for image recognition.

[13] HSU, K.-J., BHARDWAJ, K., AND GAVRILOVSKA, A. Couper: Dnn model slicing for visual analytics containers at the edge. In *Proceedings of the 4th ACM/IEEE Symposium on Edge Computing* (2019), pp. 179–194.

[14] HU, D., AND KRISHNAMACHARI, B. Fast and accurate streaming cnn inference via communication compression on the edge. In *2020 IEEE/ACM Fifth International Conference on Internet-of-Things Design and Implementation (IoTDI)* (2020), IEEE, pp. 157–163.

[15] KERAS. Keras applications. https://keras.io/api/applications/, 2021.

[16] LI, E., ZHOU, Z., AND CHEN, X. Edge intelligence: On-demand deep learning model co-inference with device-edge synergy. In *Proceedings of the 2018 Workshop on Mobile Edge Communications* (2018), pp. 31–36.

[17] LI, M., GAO, J., ZHOU, C., ZHUANG, W., ET AL. Slicing-based ai service provisioning on network edge. *arXiv preprint arXiv:2105.07052* (2021).

[18] LINDSTROM, P. Fixed-rate compressed floating-point arrays. *IEEE Transactions on Visualization and Computer Graphics 20* (08 2014).

[19] LUO, Q., HU, S., LI, C., LI, G., AND SHI, W. Resource scheduling in edge computing: A survey. *IEEE Communications Surveys & Tutorials 23*, 4 (2021), 2131–2165.

[20] MATSUBARA, Y., BAIDYA, S., CALLEGARO, D., LEVORATO, M., AND SINGH, S. Distilled split deep neural networks for edge-assisted real-time systems. In *Proceedings of the 2019 Workshop on Hot Topics in Video Analytics and Intelligent Edges* (2019), pp. 21–26.

[21] MIN, B., ROSS, H., SULEM, E., VEYSEH, A. P. B., NGUYEN, T. H., SAINZ, O., AGIRRE, E., HEINZ, I., AND ROTH, D. Recent advances in natural language processing via large pre-trained language models: A survey. *arXiv preprint arXiv:2111.01243* (2021).

[22] NAYAK, S., PATGIRI, R., WAIKHOM, L., AND AHMED, A. A review on edge analytics: Issues, challenges, opportunities, promises, future directions, and applications. *arXiv preprint arXiv:2107.06835* (2021).

[23] PARTHASARATHY, A., AND KRISHNAMACHARI, B. Defer: Distributed edge inference for deep neural networks. In *2022 14th International Conference on COMmunication Systems & NETworkS (COMSNETS)* (2022), IEEE, pp. 749–753.

[24] PREMKUMAR, S., AND SIGAPPI, A. A survey of architecture, framework and algorithms for resource management in edge computing. *EAI Endorsed Transactions on Energy Web 8*, 33 (2021), e15–e15.

[25] PYTORCH. Real time inference on raspberry pi 4. https://pytorch.org/tutorials/intermediate/realtime_rpi.html, 2021.

[26] RIOT. Riotos. https://www.riot-os.org/, 2013.

[27] SANDLER, M., HOWARD, A., ZHU, M., ZHMOGINOV, A., AND CHEN, L.-C. Mobilenetv2: Inverted residuals and linear bottlenecks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), pp. 4510–4520.

[28] SONKOLY, B., CZENTYE, J., SZALAY, M., NÉMETH, B., AND TOKA, L. Survey on placement methods in the edge and beyond. *IEEE Communications Surveys & Tutorials 23*, 4 (2021), 2590–2629.

[29] STAHL, R., HOFFMAN, A., MUELLER-GRITSCHNEDER, D., GERSTLAUER, A., AND SCHLICHTMANN, U. Deeperthings: Fully distributed cnn inference on resource-constrained edge devices. *International Journal of Parallel Programming 49*, 4 (2021), 600–624.

[30] SZEGEDY, C., IOFFE, S., VANHOUCKE, V., AND ALEMI, A. A. Inception-v4, inception-resnet and the impact of residual connections on learning. In *Thirty-first AAAI conference on artificial intelligence* (2017).

[31] TAN, M., AND LE, Q. Efficientnet: Rethinking model scaling for convolutional neural networks. In *International conference on machine learning* (2019), PMLR, pp. 6105–6114.

[32] TENSORFLOW. Tensorflow lite. https://www.tensorflow.org/lite, 2019.

[33] WU, R., GUO, X., DU, J., AND LI, J. Accelerating neural network inference on fpga-based platforms—a survey. *Electronics 10*, 9 (2021), 1025.

[34] ZHANG, Q., SUN, H., WU, X., AND ZHONG, H. Edge video analytics for public safety: A review. *Proceedings of the IEEE 107*, 8 (2019), 1675–1696.

[35] ZHANG, Z., LI, Y., GUO, Y., CHEN, X., AND LIU, Y. Dynamic slicing for deep neural networks. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (2020), pp. 838–850.

[36] ZHANG, Z., NG, L. K., LIU, B., CAI, Y., LI, D., GUO, Y., AND CHEN, X. Teeslice: slicing dnn models for secure and efficient deployment. In *Proceedings of the 2nd ACM International Workshop on AI and Software Testing/Analysis* (2022), pp. 1–8.

[37] ZOPH, B., VASUDEVAN, V., SHLENS, J., AND LE, Q. V. Learning transferable architectures for scalable image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), pp. 8697–8710.