2023 IEEE International Conference on Edge Computing and Communications (EDGE)

Reducing Inference Latency with Concurrent Architectures for Image Recognition at Edge

Ramyad Hadidi[§*]
Rain AI
ramyad@rain.ai

Jiashen Cao[§]
Georgia Tech
jiashenc@gatech.edu

Michael S. Ryoo
Stony Brook University and Google
mryoo@cs.stonybrook.edu

Hyesoon Kim
Georgia Tech
hyesoon.kim@gatech.edu

***Abstract*—Satisfying the high computation demand of modern deep learning architectures is challenging for achieving low inference latency. The current approaches in decreasing latency only increase parallelism within a layer. This is because architectures typically capture a single-chain dependency pattern that prevents efficient distribution with a higher concurrency (*i.e.*, simultaneous execution of one inference among devices). Such single-chain dependencies are so widespread that even implicitly biases recent neural architecture search (NAS) studies. In this visionary paper, we draw attention to an entirely new space of NAS that relaxes the single-chain dependency to provide higher concurrency and distribution opportunities. To quantitatively compare these architectures, we propose a score that encapsulates crucial metrics such as communication, concurrency, and load balancing. Additionally, we propose a new generator and transformation block that consistently deliver superior architectures compared to current state-of-the-art methods. Finally, our preliminary results show that these new architectures reduce the inference latency and deserve more attention.**

***Index Terms*—Edge AI, Neural Architecture Search, Distributed and Collaborative Edge Computing, IoT, Collaborative Edge & Robotics**

I. Introduction & Motivation

Increasingly deeper and wider convolution/deep neural networks (CNN/DNN) [1]–[3] with higher computation demands are continuously attaining higher accuracies. Nevertheless, the high computation and memory demands of these DNNs hinder achieving low inference latency [4]. Although current platforms exploit parallelism, we discover that, since most architectures capture a *single-chain dependency pattern* [5]–[7], shown in Figures 1a & b, we cannot efficiently extend concurrency and distribution beyond current explicit parallelism exposed within intra-layer computations (*i.e.*, matrix-matrix multiplications) to reduce the latency of an inference. In other words, distribution and concurrency, if any, are implemented at data level [8], which only increases the throughput.

The status quo approaches in reducing the inference latency are always applied *after* an architecture is defined (*e.g.*, reducing parameters with weight pruning [9], [10] or reducing computation with quantization or compression [11]–[13]). Additionally, for extremely large architectures, limited model parallelism is applied on final layers (*i.e.*, large fully-connected layers that do not fit in the memory of edge devices [14]–[16]). However, since model-parallelism methods do not change the architecture, distributing all layers with such methods adds several synchronization/merging points, incurring high communication overheads (Figure 1a & b). We discover that the single-chain inter-layer dependency pattern, common in all the well-known architectures and even in state-of-the-art neural architecture search (NAS) studies [17], prevents the efficient model distribution for reducing inference latency.

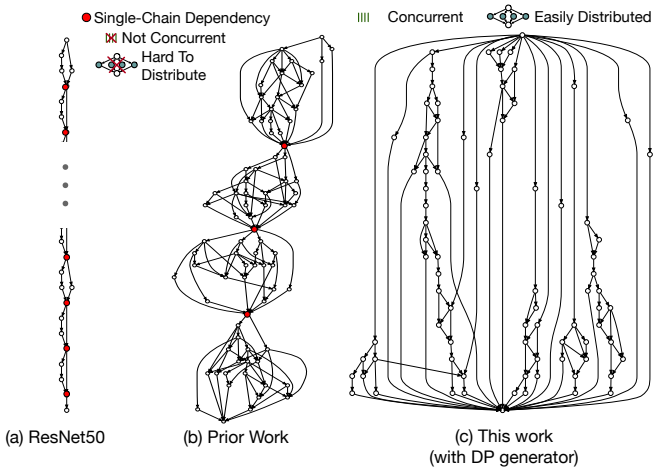


Fig. 1: **Sampled Architectures Overview** – (a) & (b) Limited concurrency and distribution due to single-chain dependency. (c) Improved concurrent architecture.

This visionary paper addresses the single-chain data dependency in current architecture designs and endeavors to inspire discussion for new concurrent architectures for at-edge distribution. To do so, first, we analyze architectures generated by recent unbiased NAS studies [17] and discover that *scaling/staging* blocks implicitly enforce dependencies. Then, we generate new architectures with prior and our new distance-based network generators using our new probabilistic scaling block. Then, for quantitatively comparing generated architectures, we propose a *concurrency score* that encapsulates important metrics such as communication, load balancing, and overlapped computations, by reformulating the problem as a hypergraph partitioning problem [18], [19]. Based on the scores and experiments, our generated architectures have higher concurrency and are more efficient for distribution

This work was partially supported by the NSF grant number 2103951 and Institute of Information and Communications Technology Planning and Evaluation grant funded by the Korea government (No. 2021-0-00766).

[§]Equal contribution

[*]This work was done when the author was affiliated with Georgia Tech.

2767-9918/23/$31.00 ©2023 IEEE
DOI 10.1109/EDGE60047.2023.00046

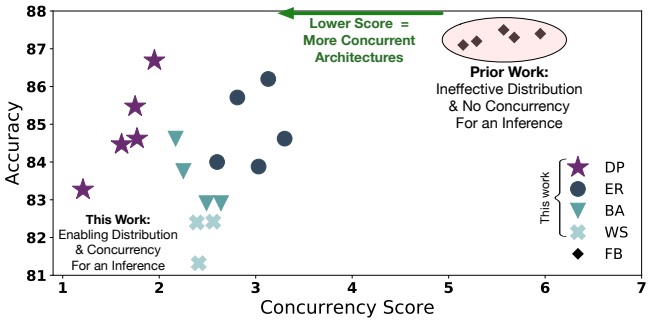


Fig. 2: **Accuracy vs. Concurrency Score** – Randomly sampled concurrent architectures generated with our NAS consistently achieve competitive accuracies with a higher concurrency and distribution opportunities during an inference (Flower-102, §III).

than current architectures, an example of which is shown in Figure 1c. Additionally, as shown in Figure 2, they provide competitive accuracy while delivering high concurrency, directly proportional to inference latency (Figure 8). Our experiment results (on over 1000 samples) show that our architectures achieve 6–7× faster inference time. As an added benefit, the current methods for reducing the inference latency can be applied on top of our generated architectures. The following is our contribution:

Addressing Single-Chain Data Dependencies: Our concurrent architectures created by network generators (especially the new distance-based generator) break current biased designs by delivering high concurrency.

Proposing Representative Concurrency Score: Our problem formulation based on hypergraph theory encapsulates critical metrics to quantitatively compare all architectures for efficient distribution and concurrency.

II. Related Work

Computation & Parameter Reduction: Reducing computation and parameters to reduce inference latency is an active research area. These techniques are applied after a model's architecture is fixed. One common approach is to remove the weak connections with weight pruning [9], [20]–[23], in which the close-to-zero weights are pruned away. It is also been shown that moderate pruning with iterative retraining enables superior accuracy [9]. Quantization and low-precision inference [11], [24]–[27] change the representation of numbers for faster calculations. Several methods also have been proposed for binarizing the weights [28]–[30]. The concurrent architectures can also benefit from these approaches, making them complementary to further reduce inference latency.

Concurrency & Distribution: With increasingly larger architectures and widespread usage of deep learning, distribution have gained attention [31]–[34]. Most of the techniques either exploit data or model parallelism [5], [31]. *Data parallelism only increases the throughput of the system and does not affect the latency.* Model parallelism divides the work of a single inference. However, model parallelism keeps the connections intact. Thus, applying model parallelism on intra-layer computations results in a huge communication overhead for sharing the partial results after each layer due to existing single-chain dependency. SplitNet [35] focuses on improving the concurrency opportunity within an architecture by explicitly enforcing dataset semantics in the distribution of *only* the final layers. Each task needs to be handcrafted individually for each dataset by examining the semantics in the dataset. In this paper, we propose concurrent architectures that is generated by NAS by considering all important factors for distribution, which has not been explored by prior work.

Neural Architecture Search: With the growing interests in automating the search space for architectures, several studies [2], [3], [17], [36]–[39] have proposed new optimization methods. Most of these studies [2], [36] utilize an LSTM controller for generating the architecture. However, as pointed out in [17], the search space in these studies is determined by the implicit assumption in network generators and sometimes explicit staging (*i.e.*, downsampling spatially while upsampling channels). Although Xie *et al.* [17] aimed to remove all the implicit wiring biases from the network generator by using classical random graph generator algorithms, they introduced a scaling/staging bias in the final architecture to deal with a large amount of computation. Such stagings create a merging point after a stage where all the features are collected and downsampled before the next stage. Hence, the generated architecture still carries the single-chain of dependency which limits the further concurrency. In contrast, our proposed architectures do not enforce such a dependency by removing this bias. Moreover, compared to prior work, our target is to reduce inference latency by increasing concurrency, which has not been explored before.

III. Concurrent Architectures

Here, we propose concurrent architectures that break the single-chain dependency pattern for enabling the concurrent execution of an inference. To improve distribution and concurrency, we aim to search for an architecture that has minimal communication overhead and is load balanced when it is distributed. To do so, the following provides the general problem formulation, while §III-A and §III-B describe our implementation details. In §III-C, we extend the representation to quantitatively study distribution and concurrency opportunities, derived by reformulating the problem as a hypergraph partitioning problem.

Overview: The current design of neural architectures is optimized for prediction accuracy and has an implicit bias towards the single-chain approach [17], [36], as we discussed in §I. This bias limits concurrency and distribution for reducing inference latency. In other words, only the computation within a layer is performed in parallel and not the computation within a model. To tackle this challenge, we aim to consider concurrency and distribution during the design stage and test if such architectures provide higher concurrency with good accuracy. To do so, first, we use network generators to create a

random graph structure, which represents a potential architecture. Among all generated architectures, we sample (without any optimized search) and evaluate generated architectures with our proposed concurrency score. Then, we transform the graph to a DNN and perform experiments. Our final results show a promising direction worth exploring.

DAG Representation: A neural architecture, $\mathcal{N}$, can be represented as a directed acyclic graph (DAG) because the computation flow always goes in one direction without looping. We define a DAG as $\mathcal{G} = (V, E)$ where V and E are sets of vertices and edges, respectively. We define a network generator, f, as a function that constructs random DAG. f creates the edge set, E, and defines the source and sink vertices for each edge, regardless of the type of the vertices. Although network generators could be deterministic (*e.g.*, a generator implemented with NAS approach), we are interested in stochastic network generators. The reasons are two-fold. First, the stochastic generator provides a larger search space than the deterministic generator, so it is more likely to remove any bias. Second, since, unlike prior work, we don not use scaling/staging to glue different parts of our NAS generated network [17] (shown in Figure 1b), stochastic generators provide more options for a potential solution. Note that the generated DAG only represents the dataflow and does not include the weights, which are learned in subsequent steps. §III-A provides more details about our network generators and how we utilize them to create a DAG.

DAG to DNN: Once we have found a promising DAG representation after the concurrency score study, we transform the DAG into an actual DNN. Vertices in DAG are components (*e.g.*, layers or sub-networks) and edges are connections. Within the process of transformation, we convert the nodes in DAG to a block of layers and connect blocks with its corresponding edge in DAG. Each vertex, V_i, has several properties such as the type of the layer and its properties (*e.g.*, depth, width, activation size, *etc*). In this paper, we use a uniform computation in vertices: ReLU, 3x3 separable convolution [40], and batch normalization [41].

A. Network Generators

We use three classical random graph generators as baselines. Additionally, after discovering that state-of-the-art generators do not generate a concurrent architecture, we propose a new graph generator with distance-based heuristics. Below, we describe the generators identified by how their stochastic nature influences the graph. Note that although the first three generators are based on [17], to generate concurrent architectures, we have removed the introduced staging blocks, which enforces the single-chain dependency in prior work. Thus, all the studied architectures in this work are novel and have never been studied before.

Once we obtain an undirected random graph from the generator, we convert the undirected graph to DAG by using the depth-first search algorithm. The vertices with smaller vertex ID is traversed earlier than vertices with larger ID. As the final step, we add an input vertex to all vertices without predecessors and an output vertex to all vertices without successors. This ensures that we obtain a DAG with a single source and sink.

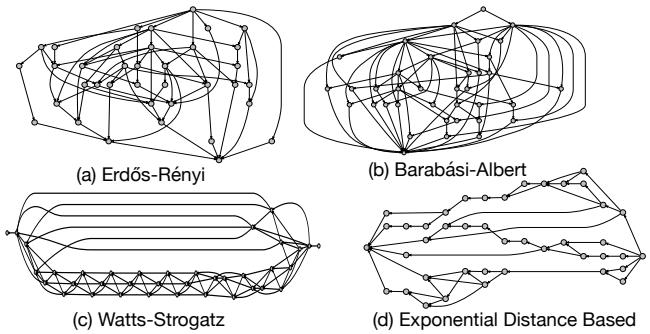


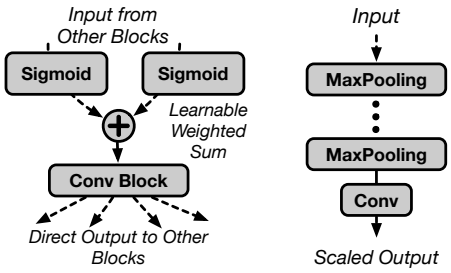
Fig. 3: **Network Generators** – Four examples of different random graph generators. Note that only (d) produces a good concurrent balanced graph.

(1) Independent Probability: In this group, the probability of adding an edge is independent of other properties. Similar to the Erdős and Rényi model (ER) [42], in which an edge exists with a probability of P. Generators with independent probability completely ignore the graph structure and create a connected graph (Figure 3a) that is hard to efficiently distribute.

(2) Degree Probability: In this group, the probability of adding an edge is defined by the degree of one of its connected vertices. A vertex with a higher degree has more probability of accepting a new edge. Figure 3b shows an example of such a generator. Barabási-Albert model (BA) [43], first adds M disconnected vertices, then for the total number of vertices until N, it adds a total of M edges with a linear probability proportional to the degree of each vertex (*i.e.*, a total of $M(N-M)$ edges). Generators with degree probability create a tree-structured graph, in which at least one vertex is strongly connected to other vertices. Such a graph structure is hard to distribute since all the vertices are dependent on at least one vertex, if not more.

(3) Enforced Grouping: In this group, initially, a pre-defined grouping is performed on disconnected vertices and then edges are added based on the groups. Small world graphs [44]–[46] are good examples. In one approach (WS) [45], vertices are placed in a ring and each one is connected to $K/2$ neighbors on both sides. Then, in a clockwise loop on vertices, an existing edge between its i_{th} neighbor is rewired with a uniform probability of P for $K/2$ times. As shown in Figure 3c, a graph with the WS algorithm tends to form a single-chain structure if P is small. With a larger P, the structure becomes similar to ER.

(4) Distance Probability: In distance probability (DP), initially, a pre-defined grouping is performed on disconnected vertices, then a distance probability function defines the existence of an edge. We first arrange the vertices in a ring. Then, the probability of adding an edge between two vertices is dependent on their distance. In other words, closer vertices have a higher probability of getting edges.



(a) Basic Building Block. (b) Scaling Building Block.

Fig. 4: **Building Blocks** – Building blocks used for conversion from DAG to DNN.

– *Distance Metrics:* We define distance d as the smallest number of nodes plus one between two nodes in a ring. The maximum distance can be half of the total number of nodes, which is $N/2$. We use the distance to re-scale the passed in probability P presented in WS. We use the exponential re-scaling function:

$$P_{\text{new}} = \alpha P^{\beta d}, \tag{1}$$

in which α and β are constants. The probability quickly decreases as the distance increases. This mechanism naturally creates multiple locally strongly connected graphs, Figure 3d, which can be distributed. However, we still need to examine the distribution and concurrency opportunities, which are presented in §III-C.

B. Transformations

Transformations are operations, the main objective of which is to create a reasonable architecture, that happens after the construction of the DAG. We first introduce the building blocks, which include a scaling building block that, contrary to previous work, does not enforce a single-chain dependency.

Building Block: During the process of transforming a DAG to DNN, vertices are interpreted as basic building blocks, as shown in Figure 4. Inside a basic building block, Sigmoid activations are applied on inputs, then, the activations are summed with a learnable weighted sum. The Sigmoid function is used to avoid weighted sum overflow. As described before, the `conv` block consists of a ReLU, 3×3 separable convolution, and batch normalization.

Redefining Staging: Staging is deemed to be necessary for all NAS generated architectures to reduce the computation and facilitate learning. For staging, after a few layers, usually, the common method is to gather and merge outputs from all transformation vertices, conduct downsampling, and channel upsampling. However, such staging points create a rigid architecture with single-chain dependencies that are hard to distribute and execute concurrently (*e.g.*, [17]). To address the single-chain bottleneck problem caused by staging, the first solution is implementing a uniform channel size for the entire architecture. In other words, all `conv` blocks share the same filter size. Thus, there would be no need to merge and synchronize at a point during an inference. However, as shown in Table I, the uniform channel size approach works well on a small image dataset (*e.g.*, Cifar-10), but it fails to achieve good accuracy on a dataset with larger image dimension (*e.g.*, Flower-102).

In this paper, we propose individual staging after any `conv` block. Because of that, inputs to a `conv` block could have different dimensions. To tackle this problem, we dynamically add a new scaling block in the process of construction. The scaling block consists of a number of maxpooling layers. Maxpooling layers downsamples the dimensions to match with the smallest dimension in the input. We also use 1×1 convolution layers to upsample the channel size to match the highest channel size in the inputs in these scaling blocks. Therefore, we avoid bottlenecks in generated architecture.

We adopted two design choices for the staging mechanism. In the first design, greedy-based staging, we start with greedy-based staging. Within the construction process, we set an upper limit for channel size. As long as channel sizes have not reached the upper bound, we conduct staging (*i.e.*, downsample the input & upsample the channel). However, this design raises an issue that intermediate outputs are quickly squeezed through the maxpooling layer, which discards important features. This approach hurts the accuracy to some extent. In the second design, probabilistic-based staging, we use a probabilistic method in staging. In this design, although the channel size may have not reached the limit, staging is done with a fixed probability of 0.5 to avoid discarding features too quickly. As shown in Tables II and III, the probabilistic approach achieves a better accuracy rate than the greedy-based approach. In addition, Table III shows that probabilistic staging supports higher accuracy with less parameter size because (i) probabilistic staging gracefully discards features, so the architecture learns better; and (ii) the aggressive greedy-based staging creates more size mismatch, so it requires more scaling blocks.

C. Concurrency & Distribution

Our goal in this paper is to inspire concurrent architecture designs to improve inference latency performance. As a result, besides common accuracy considerations, we need to study the concurrency and distribution opportunities of a candidate architecture. To help the community to extend our study, instead of focusing and showcasing on a single architecture, we are interested in finding a customized *concurrency score* (CS) for a given architecture, $\mathcal{N}$, that is easily calculated. In this way, we can study various architectures and future works that can further improve this work. CS shows how optimal the concurrent and distributed task assignment for an architecture

TABLE I: **Accuracy of Uniform Channels** – The mean accuracy comparison between sampled group architectures with uniform channel *vs.* handcrafted without any advanced optimizations. (baselines Cifar-10 and Flower-102 are vanilla CifarNet and ResNet-50, respectively).

Dataset	Baseline	DNNs with Uniform Channels
Cifar-10 32×32	80.70	81.13
Flower-102 224×224	87.80	74.73 **(Fails to Scale!)**

TABLE II: **Average Accuracy** – Comparison of randomly sampled group of generated architectures with different staging choices (trained on Flower-102).

Staging/Samples	A	B	C	Overall Mean
Greedy	82.30	81.32	82.42	82.01
Probabilistic	82.42	86.69	84.62	**84.58**

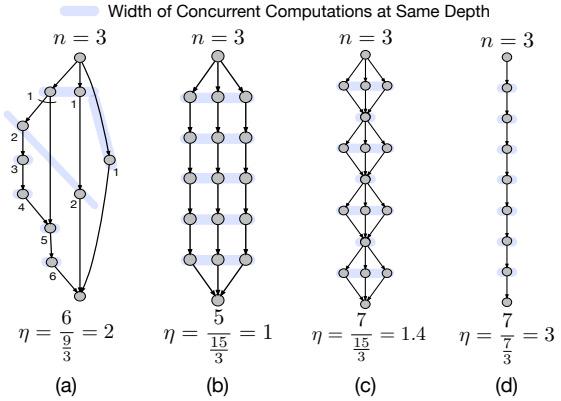


Fig. 5: **Overlapped of Computation Metric** – Illustration of η, lower η means higher overlap.

is. A lower PS score represents fewer communications, better load-balanced tasks, and more distribution opportunities with more overlapped computation, so the architecture is more efficient for concurrency.

Metrics in The Score: We can formulate our problem of allocating tasks on n units as a multi-constraint problem. The first constraint is that all units should perform the same amount of work, or be load balanced. Second, the communication amount, the main bottleneck in distribution, should be at a minimum. And third, we want to minimize runtime by increasing overlapped computations among the units. The first two constraints are addressable by finding a set of hyper-graph partitions, in which we divide the vertices into equally weighted sets so that few hyper-edges cross between partitions. The derivable metric is the amount of variability in loads (δ_W) and a total of communication (Λ). The third constraint is measurable by finding the longest path between the input and output vertices on the DAG and quantify concurrency (η). For instance in pipeline parallelism, the longest path is the entire architecture, as a result the latency is never reduced (and throughput is increased). Now, we provide the formal definition of these solutions by first studying the DAG.

Maximizing Overlapped Computations: We measure how overlapped is the inter-layer computations of an architecture from its DAG, or η, as a ratio. We measure this by observing the longest path in the distinct paths between input and output vertices in the DAG, $\mathcal{G}$, relative to the number of the computation cores, n. Assume $\{d_i\}$ is the set of distinct longest paths in $\mathcal{G}$. We define η as

$$\eta = \frac{\max\{d_i\}}{|\mathcal{V}|/n}, \tag{2}$$

in which $|\mathcal{V}|$ is the total number of vertices. Figure 5 depicts an examples of η. A higher η value shows a more limited opportunity to overlap the computation. Figure 5 also shows the width of the overlapped computation at the same depth (*i.e.*, DFS depth with the source of input), which is a good representation of why some architectures are more efficient for concurrency.

Hypergraph Representation: Using graph representations in task assignment for distributed computing is a well-known problem [47]. Basically, in the generated DAG, vertices of the graph represent the units of computations, and edges encode data dependencies. We can indicate the amount of work and/or data, by associating weights (w) and costs (λ) to vertices and edges, respectively. However, a DAG representation does not sufficiently capture the communication overhead, load balancing factor, and the fact that some edges are basically sending the same data/features. Therefore, for task assignment, we use an alternative graph representation, derivable from the DAG, hypergraph. A hypergraph [18] is a generalization of a graph, in which an edge can join any number of vertices [48]. The hypergraph representation, common in optimization for integrated circuits [19], enables us to consider the mentioned factors.

Formal Definition of Hypergraph: A hypergraph $\mathcal{H} = (\mathcal{V}, \mathcal{E})$ is defined as a set of vertices $\mathcal{V}$ and a set of hyper-edges $\mathcal{E}$ selected among those vertices. Every hyper-edge $e_j \in \mathcal{E}$ is a subset of vertices, or $e_j \subseteq \mathcal{V}$. The size of a hyper-edge is equal to the number of vertices.

Hypergraph Partitioning: We assign weights (w_i) and costs (λ_j) to the vertices ($v_i \in \mathcal{V}$) and edges ($e_j \in \mathcal{E}$) of the hypergraph, respectively. $\mathcal{P} = \{V_1, V_2, V_3, ..., V_P\}$ is a P-way partition of $\mathcal{H}$ if (i) $\forall V_i, \emptyset \neq V_i \subset \mathcal{V}$, (ii) parts are pairwise disjoint, and (iii) $\bigcup \mathcal{P} = \mathcal{V}$. A partition is balanced if $W_p \leq \varepsilon W_{\text{avg}}$ for $1 \leq p \leq P$, where $W_{\text{avg}} = \Sigma_{v_i \in \mathcal{V}} {}^{w_{v_i}}/_P$ denotes the weight of each part, and ε represents the imbalance ratio, or δ_W.

In a partition $\mathcal{P}$ of $\mathcal{H}$, a hyper-edge that has at least one vertex in a part is said to connect that part. The number of connections γ_j of a hyper-edge e_j denotes the number of parts connected by e_j. A hyper-edge is a cut if $\gamma_j > 1$. We define such hyper-edges as an external hyper-edges $\mathcal{E}_E$. The total communication for $\mathcal{P}$ is

$$\Lambda = \sum_{e_j \in \mathcal{E}_E} \lambda_j (\gamma_j - 1). \tag{3}$$

Therefore, our two constraints can be defined as a hypergraph partitioning problem, in which we divide a hypergrpah into two or more parts such that the total communication is minimized, while a given balance criterion among the part weights is maintained. We can solve this NP-hard [19] problem with multi-paradigm algorithms, such as hMETIS [49] relatively

TABLE III: **Average Accuracy/Parameters Ratio** – Comparison of randomly sampled generated architectures with different staging choices (trained Flower-102).

Staging/Samples	A	B	C	Overall Mean
Greedy	2.31	2.27	2.63	2.40
Probabilistic	3.00	3.28	3.58	**3.29**

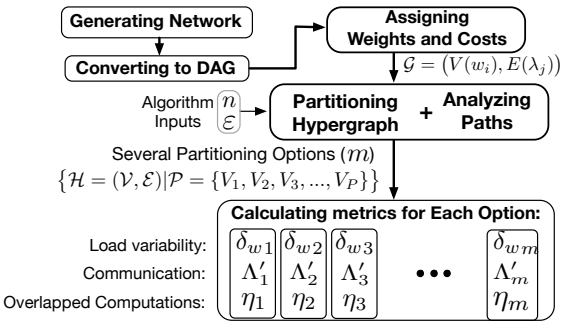


Fig. 6: **Calculating Concurrency Score** – Summarizing steps for deriving the score.

fast. Note that solving this problem is a pre-processing step, which does not affect runtime.

Concurrency Score: Now, we have the tools to calculate the concurrency score, CS. Figure 6 summarizes all the steps to derive our metrics: Load variability, δ_w; total amount of communication, Λ; and overlapped computations, η. Hypergraph algorithm accepts the number of units and a higher bound of ε. By changing the ε, we create a set of partitioning options, for each of which we compute all the metrics. Note that the DAG input requires a weight and cost value for every vertex and edge, respectively. Both of these values are easily derivable. The weight of a vertex is directly proportional to its floating operations (FLOPs), reported by most frameworks. The cost of an edge is directly proportional to the transferred data size. To get CS, first, we need to normalize the communication metric. We write Λ as $\Lambda' = {}^{\Lambda}/_{(U_c \times n)}$, in which U_c is a unit of data and n is the number of units. We define

$$\text{CS} = \sqrt[1/3]{\delta_w^a \Lambda'^b \eta^c}, \tag{4}$$

as a custom concurrency score, in which a, b and c are constant that show the relative importance of each metric for a user. In this paper, we assume $a = c = 1$ and $b = 1.5$, for a higher priority for communication. We chose U_c as the smallest amount of communication for an edge in a generator. Hence, a higher CS value shows poor distribution and concurrency opportunities.

IV. Experimental Analysis

In this section, we evaluate our generated architectures by comparing our customized generator and transformation process with prior work. The results demonstrate that our generated architectures preserve accuracy while achieving better concurrency scores by removing the implicit bias of single-chain dependency. Besides, by running the final architecture on actual devices, we show that the concurrency score provides a reasonable heuristic about the real performance.

A. Experimental Setup

Generators: All generators use probabilistic scaling blocks. FB represents prior work in unbiased NAS with staging blocks [17]. As mentioned in §III-A, although ER, BA, and WS generators are based on [17], we remove the staging block that causes the limited concurrency. As a result, all the studied

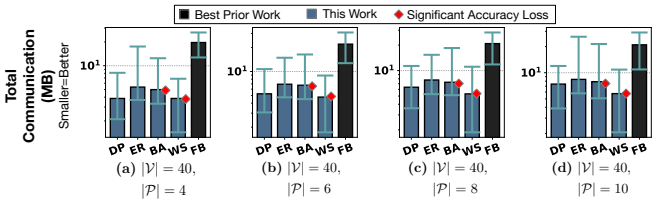


Fig. 7: **Total Communication with Distribution** – Measured communication in MB for 1000 sampled architectures in each category for 40 vertices on {4,6,8,10} units.

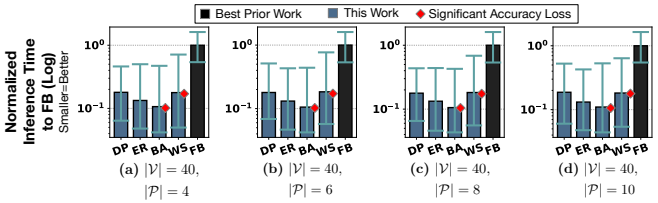


Fig. 8: **Inference Time** – Normalized inference time normalized to FB (§IV-A) for 1000 sampled architectures in each category for 40 vertices on {4,6,8,10} units.

network generators and resulting architectures are novel and have never been studied before.

Randomization: To evaluate the accuracy of randomly generated architecture, we collect representative samples with *no optimized search*. we followed the same training procedure for architectures and reported the average accuracy. For CS, total communication, and computation time evaluations, we collect 1,000 samples with no optimized search and compare across different generators.

Datasets: We conducted experiments on multiple datasets to ensure the extensibility of concurrent architectures. We use two image classification datasets; (i) Cifar-10 [50], which contains 60K 32×32 images in 10 classes; and (ii) Flower-102 [51], which contains 16K 224×224 images in 102 classes. We strongly encourage future extensive studies on larger datasets, but given the heavy-compute bound of NAS-based experiments, we chose to use representative datasets studied in most of the prior works [52].

Training Procedure: We use a uniform training pipeline with a stochastic gradient descent optimizer for all architectures. We train on Cifar-10 with 100 epochs and on Flower-102 with 300 epochs. We report the top-1 classification accuracy on the test sets. For the first 100 epochs, we set the learning rate to be 1e-3 and momentum to be 0.9. We changed the learning rate to 5e-4 and momentum to 0.95 for the remaining 200 epochs on Flower-102.

Implementation: We implemented all graph representations in Python NetworkX [53] library. Then, we convert a graph to a PyTorch [54] compatible model. We constructed a graph-based forwarding path in PyTorch module class to directly reproduce the graph structure.

B. Experiments

We analyze the results from three perspectives, communication, latency, and concurrency score. Because we are interested

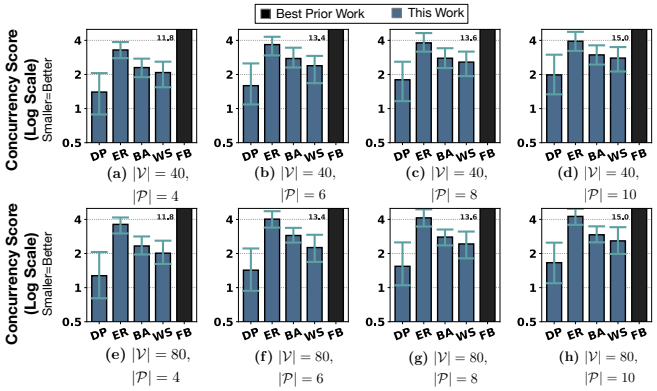


Fig. 9: **Concurrency Scores** – Measured CS for 1000 sampled architectures in each category with {40,80} vertices on {4,6,8,10} units (§IV-A).

in finding a general solution, we start with the architecture stability evaluation that particularly focuses on the architecture parameter size. Then, we show the generated architectures achieve competitive accuracies, while, in the last part, we illustrate the high concurrency and distribution opportunities of these architectures.

Architecture Stability: For the architecture stability experiment, we used a fixed number of 40 building blocks. We created 1,000 samples from each network generator. We recorded mean and standard deviation regarding the parameter sizes. We also evaluate the architecture stability under different staging design choices (greedy vs probabilistic). From Table IV, we see that proposed generators with greedy scaling blocks creates larger but more stable architectures than with probabilistic scaling blocks. Additionally, we see that our proposed DP generator creates the most efficient architecture. We will see that architectures that use DP generators are generally the most optimized.

Accuracy Study: Here, we demonstrate that the concurrent architectures achieve competitive accuracy on both Cifar-10 and Flower-102 datasets. Given the heavy-compute bound of NAS-based experiments, we encourage further studies on larger datasets. We used the same architecture samples as before without any optimized search and reported both mean and best results. As shown in Table V and VI, our concurrent architectures achieve comparable accuracy on both datasets. Generated DNNs achieve better or similar accuracy on Cifar-10. For Flower-102, because both network generation and transformation processes have more randomness, the mean accuracy has a small gap compared to the baseline. However,

TABLE IV: **Parameter Size Stability** – The mean and standard deviation of parameter size in sampled generated architectures with different staging.

		ER	AB	WS	DP
Greedy Staging	**Mean**	48.63	48.33	42.03	35.03
	Std	1.11	0.91	1.28	2.25
Probabilistic Staging	**Mean**	46.03	45.63	36.44	26.69
	Std	2.70	4.41	3.52	3.05

TABLE V: **Concurrent Architectures on Cifar-10** – Overall sampled metrics.

	Mean Acc.	Best Acc.	Mean Acc./Param.	Best Acc./Param.
CifarNet	80.70	80.70	5.38	5.38
ER	81.33	81.81	4.94	5.03
BA	80.29	81.66	4.81	4.92
WS	79.89	81.45	4.75	4.84
DP	80.87	82.47	4.81	4.90

TABLE VI: **Concurrent Architects on Flower-102** – Overall sampled metrics.

	Mean Acc.	Best Acc.	Mean Acc./Param.	Best Acc./Param.
ResNet-50	87.80	87.80	3.43	3.43
ER	84.88	86.20	2.11	2.43
BA	82.91	84.62	2.41	2.91
WS	81.46	86.57	3.17	3.10
DP	84.66	86.69	3.19	3.28

the best accuracy is close to the baseline, so we believe the accuracy gap can be leveraged by conducting an optimized search in terms of accuracy.

Concurrency Study: Finally, to show improved distribution and concurrency opportunities, we examined the concurrency score of our architectures to ResNet-50 and FB (§IV-A) by sketching width/depth histograms in Figure 10. As shown, we achieve higher width/depth, which enables more concurrency, while provides lower maximum depth, which enables shorter execution time. To quantitatively compare the generators and FB, Figure 9 depicts concurrency scores, summarized on over 1000 architectures in each category per set. As seen, our generators (and specifically DP) consistently gain the best score. Moreover, to gain more insights, Figure 7 and 8 illustrate total communication with distribution and inference (*i.e.* computation) time, when each architecture is deployed on $|\mathcal{P}|$ units. We see that though ER and BA methods deliver better computation speedup, they suffer performance slow down more from data communication. For our new generator, DP, we see an 6–7× speedup in inference time. We observe a close relationship between the reported score and actual latency and communication. In fact, latency and communication measure performance in an orthogonal way, but CS score captures the overall efficiency of the generated architecture pretty well and could be used in future studies.

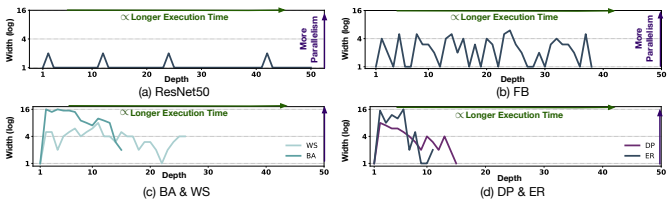


Fig. 10: **Width/Depth Histograms** – Illustration of ResNet50, FB, and concurrent architectures, which enable more concurrency and shorter inference latency.

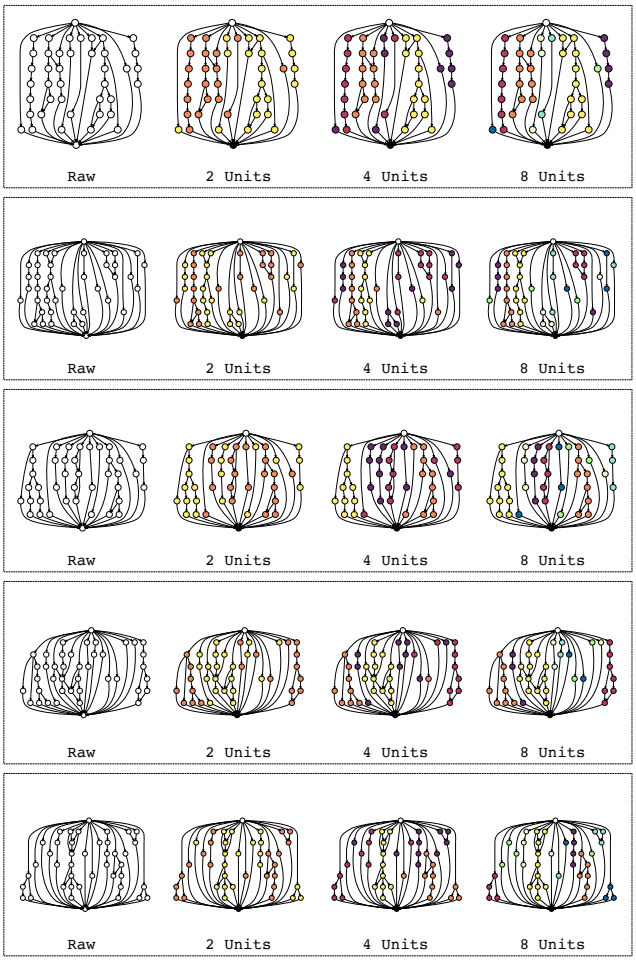


Fig. 11: **Random Neural Network Distribution** – This gives 5 examples of raw random generated neural networks, their distributions on two, four and eight units.

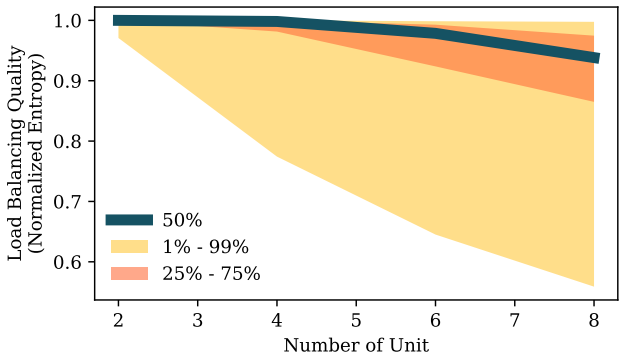


Fig. 12: **Load Balance Quality** – The load balance quality analysis on two, four, six and eight units compared to the normalized Shannon entropy value.

V. Example Deep Dive

A. Distribution

To distribute the generated networks according to the number of units, we first group node in the same sequential path together to minimize the communication overhead. The detailed algorithm of grouping can be found in §V-A5. After the nodes in the graph are grouped together, we use heuristic-based greedy algorithm §V-A6 to distribute all nodes to units. The objective of the algorithm is to balance the workload. To make the load balancing simple, we assume the final goal is that each unit performs a similar amount of computations. Ultimately, this process can be improved using various other techniques that currently is out of the scope of this paper. Here, we provide an example of our process, which starts from network generation to workload distribution.

1) Network Generation: Figure 11 demonstrates a example of raw random neural network generated. This network is later fed into a grouping and distribution algorithm to decide which unit runs which nodes.

2) Distribution to 2,4 and 8 Units: Figure 11 shows network distribution on 2,4 and 8 units. The coloring marks the node is distributed on which unit. Because all units need to run the computations of the first node, we leave it as a common node (this could be just a scatter operation). In addition, for the last node, an extra unit is needed to merge all results together, so we mark that unit as black (this could be just a gather operation).

3) Load Balancing: From the graphs, we observe that the current grouping and distribution algorithm does well load balancing under the scenario with a small number of units. The quality of load balancing affects the final inference latency, because the final results may slow down due to a bottleneck node, which happens when unbalanced loads exist. We conduct a load balance quality study as well as shown in Figure 12. We use normalized Shannon entropy value to indicate the load balancing quality (the higher the number represents the load is more balanced, and 1 means the load is perfectly balanced across distribution units). In the Figure 12, we showcase the median, $25\% - 75\%$ percentile, and $1\% - 99\%$ percentile load balancing qualities. We observe that as the number of distribution units increases, the overall load balancing quality downgrades, and the variation of quality increases. We aim to develop distribution algorithms with higher quality; however, currently, our aim in this paper is to show that parallel inference computations of a single request is a viable option and should be studied more.

4) Performance Scaling: As the final step, we also conduct a study on performance scaling. We use a total of 10 AWS t2.micro EC2 instances for performance evaluation. Each instance is equipped with only 1 vCPU and 1 GB memory. The specification are chosen to emulate edge units with limited compute and memory that have a higher computational cost (remember that constants in the Equation 4 give higher priority to communication). As shown in Figure 13, the inference latency improves when the system has more distribution units. However, The latency stops to decrease as the number of distribution units becomes 8, because the workload is not well balanced on each unit, as shown in our load balancing study. In this example, the bottleneck unit in the system causes longer latency for the entire system.

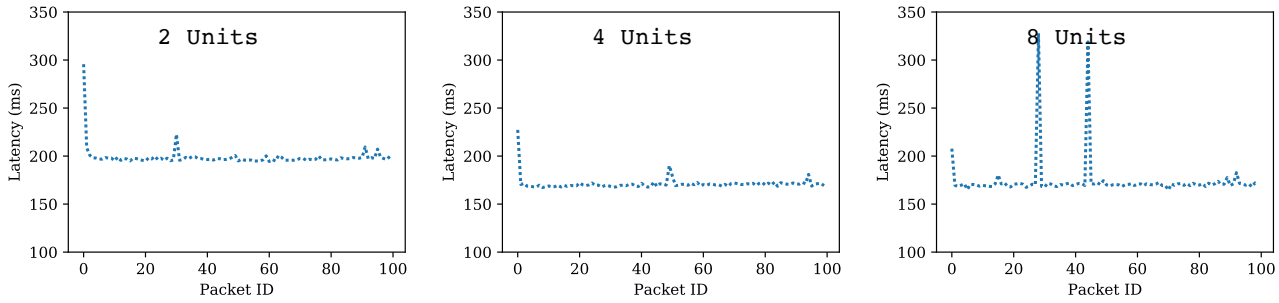


Fig. 13: **Performance Scaling** – the random neural network latency on two, four, and eight distribution units.

5) Node Grouping: The below code snippet is used to group node in a graph. The computation graph will be later used tp distribute the computations across the units in a granularity of a group.

```
def grouping(dag):
    group = dict()

    # first assign the node group to its self
    for node in dag.nodes:
        group[node] = node

    # create a undirected graph for grouping
    ug = nx.Graph()
    for edge in dag.edges:
        src, dst = edge
        ug.add_edge(src, dst)

    # use dfs to grouping
    def dfs(group_id, group, node, ug, visited):
        if node in visited or node == -1 or node ==
        ↪  0:
            return
        visited.add(node)
        group[node] = group_id
        for n in ug.neighbors(node):
            dfs(group_id, group, n, ug, visited)

    visited = set([])
    for n in ug.nodes:
        dfs(n, group, n, ug, visited)

    # collect grouping
    groups = dict()
    for n, group_id in group.items():
        if group_id in groups:
            groups[group_id].append(n)
        else:
            groups[group_id] = [n]

    return groups
```

6) Load Balancing Work Distribution: The below code snippet is used to distribute the work on each unit in the granularity of the nodes group. To make the algorithm simple, we assume each node performs a similar amount of computation, so the workload of a group is equal to the cardinality of the group (*i.e.*, number of nodes in a group).

```
def assign_workload(groups, n):
    # push jobs and units into queue
    job, unit = [], []
    for group_id, group in groups.items():
        hq.heappush(job, (-len(group), group_id))
    for i in range(n - 1):
        hq.heappush(unit, (0, i))

    # assign workload
    assignment = [[] for _ in range(n - 1)]
    while job:
        job_load, job_id = hq.heappop(job)
        job_load = -job_load
        if job_id == 0 or job_id == -1:
            continue
        unit_load, unit_id = hq.heappop(unit)
        hq.heappush(unit, (unit_load + job_load,
        ↪  unit_id))
        assignment[unit_id] += groups[job_id]
    assignment += [[-1]]

    return assignment
```

VI. Conclusion

In this work, we proposed concurrent architectures that break the single-chain dependency, a common bias in modern architecture designs. We showed that these architectures are concurrent and have more distribution opportunities for reducing the inference time while achieving competitive accuracy. Since we discover that previous NAS studies were implicitly biased in creating a sequential model, we introduced a new generator that naturally creates concurrent architectures. To quantitatively compare concurrent architectures, we proposed the concurrency score that encapsulates critical metrics in distribution.

References

[1] C. Szegedy, S. Ioffe, V. Vanhoucke, and A. A. Alemi, "Inception-v4, inception-resnet and the impact of residual connections on learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.

[2] B. Zoph, V. Vasudevan, J. Shlens, and Q. V. Le, "Learning transferable architectures for scalable image recognition," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2018, pp. 8697–8710.

[3] E. Real, A. Aggarwal, Y. Huang, and Q. V. Le, "Regularized evolution for image classifier architecture search," in *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 33, 2019, pp. 4780–4789.

[4] R. Hadidi, J. Cao, Y. Xie, B. Asgari, T. Krishna, and H. Kim, "Characterizing the deployment of deep neural networks on commercial edge devices," in *2019 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2019, pp. 35–48.

[5] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet classification with deep convolutional neural networks," in *26th Annual Conference on Neural Information Processing Systems (NIPS)*. ACM, 2012, pp. 1097–1105.

[6] K. Simonyan and A. Zisserman, "Very deep convolutional networks for large-scale image recognition," in *3rd International Conference on Learning Representations*. ACM, 2015.

[7] J. Redmon, S. Divvala, R. Girshick, and A. Farhadi, "You only look once: Unified, real-time object detection," in *Proceedings of the IEEE conference on computer vision and pattern recognition*, 2016, pp. 779–788.

[8] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, *et al.*, "Applied machine learning at facebook: A datacenter infrastructure perspective," in *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. IEEE, 2018, pp. 620–629.

[9] S. Han, H. Mao, and W. J. Dally, "Deep compression: Compressing deep neural network with pruning, trained quantization and huffman coding," in *4th International Conference on Learning Representations*. ACM, 2016.

[10] B. Asgari, R. Hadidi, H. Kim, and S. Yalamanchili, "Lodestar: Creating locally-dense cnns for efficient inference on systolic arrays," in *Proceedings of the 56th Annual Design Automation Conference 2019*. ACM, 2019, p. 233.

[11] V. Vanhoucke, A. Senior, and M. Z. Mao, "Improving the speed of neural networks on cpus," in *Proceeding Deep Learning and Unsupervised Feature Learning NIPS Workshop*, vol. 1. ACM, 2011, p. 4.

[12] B. Asgari, R. Hadidi, J. Dierberger, C. Steinichen, A. Marfatia, and H. Kim, "Copernicus: Characterizing the performance implications of compression formats used in sparse workloads," in *2021 IEEE International Symposium on Workload Characterization (IISWC)*. IEEE, 2021, pp. 1–12.

[13] B. Asgari, R. Hadidi, and H. Kim, "Ascella: Accelerating sparse computation by enabling stream accesses to memory," in *Proceedings of Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2020, pp. 318–321.

[14] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Towards collaborative inferencing of deep neural networks on internet of things devices," *IEEE Internet of Things Journal*, 2020.

[15] R. Hadidi, J. Cao, M. Woodward, M. S. Ryoo, and H. Kim, "Musical chair: Efficient real-time recognition using collaborative iot devices," *arXiv preprint arXiv:1802.02138*, 2018.

[16] R. Hadidi, J. Cao, M. S. Ryoo, and H. Kim, "Distributed perception by collaborative robots," *IEEE Robotics and Automation Letters (RA-L), Invited to IEEE/RSJ International Conference on Intelligent Robots and Systems 2018 (IROS)*, vol. 3, no. 4, pp. 3709–3716, Oct 2018.

[17] S. Xie, A. Kirillov, R. Girshick, and K. He, "Exploring randomly wired neural networks for image recognition," in *Proceedings of the IEEE International Conference on Computer Vision*, 2019, pp. 1284–1293.

[18] U. V. Catalyurek and C. Aykanat, "Hypergraph-partitioning-based decomposition for parallel sparse-matrix vector multiplication," *IEEE Transactions on parallel and distributed systems*, vol. 10, no. 7, pp. 673–693, 1999.

[19] T. Lengauer, *Combinatorial algorithms for integrated circuit layout*. Springer Science & Business Media, 2012.

[20] J. Yu, A. Lukefahr, D. Palframan, G. Dasika, R. Das, and S. Mahlke, "Scalpel: Customizing dnn pruning to the underlying hardware parallelism," in *44th International Symposium on Computer Architecture (ISCA)*. IEEE, 2017, pp. 548–560.

[21] J. Lin, Y. Rao, J. Lu, and J. Zhou, "Runtime neural pruning," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 2181–2191.

[22] W. Wen, C. Wu, Y. Wang, Y. Chen, and H. Li, "Learning structured sparsity in deep neural networks," in *Advances in neural information processing systems*, 2016, pp. 2074–2082.

[23] S. Anwar, K. Hwang, and W. Sung, "Structured pruning of deep convolutional neural networks," *ACM Journal on Emerging Technologies in Computing Systems (JETC)*, vol. 13, no. 3, p. 32, 2017.

[24] M. Courbariaux, Y. Bengio, and J.-P. David, "Training deep neural networks with low precision multiplication," *arXiv preprint arXiv:1412.7024*, 2014.

[25] Y. Gong, L. Liu, M. Yang, and L. Bourdev, "Compressing deep convolutional networks using vector quantization," *arXiv preprint arXiv:1412.6115*, 2014.

[26] U. Köster, T. Webb, X. Wang, M. Nassar, A. K. Bansal, W. Constable, O. Elibol, S. Gray, S. Hall, L. Hornof, *et al.*, "Flexpoint: An adaptive numerical format for efficient training of deep neural networks," in *Advances in Neural Information Processing Systems (NIPS)*, 2017, pp. 1742–1752.

[27] D. Lin, S. Talathi, and S. Annapureddy, "Fixed point quantization of deep convolutional networks," in *International Conference on Machine Learning*, 2016, pp. 2849–2858.

[28] F. Li, B. Zhang, and B. Liu, "Ternary weight networks," *arXiv preprint arXiv:1605.04711*, 2016.

[29] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to +1 or- 1," *arXiv preprint arXiv:1602.02830*, 2016.

[30] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *ECCV'16*. Springer, 2016, pp. 525–542.

[31] J. Dean, G. Corrado, R. Monga, K. Chen, M. Devin, M. Mao, A. Senior, P. Tucker, K. Yang, Q. V. Le, *et al.*, "Large scale distributed deep networks," in *NIPS'12*. ACM, 2012, pp. 1223–1231.

[32] J. Mao, X. Chen, K. W. Nixon, C. Krieger, and Y. Chen, "Modnn: Local distributed mobile computing system for deep neural network," in *2017 Design, automation and Test in eurpe (Date)*. IEEE, 2017, pp. 1396–1401.

[33] S. Teerapittayanon, B. McDanel, and H. Kung, "Distributed deep neural networks over the cloud, the edge and end devices," in *37th IEEE International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2017, pp. 328–339.

[34] Y. Kang, J. Hauswald, C. Gao, A. Rovinski, T. Mudge, J. Mars, and L. Tang, "Neurosurgeon: Collaborative intelligence between the cloud and mobile edge," in *22nd ACM International Conference on Architectural Support for Programming Languages and Operating Systems*. ACM, 2017, pp. 615–629.

[35] J. Kim, Y. Park, G. Kim, and S. J. Hwang, "Splitnet: Learning to semantically split deep networks for parameter reduction and model parallelization," in *Proceedings of the 34th International Conference on Machine Learning-Volume 70*. JMLR. org, 2017, pp. 1866–1874.

[36] B. Zoph and Q. V. Le, "Neural architecture search with reinforcement learning," 2016.

[37] B. Baker, O. Gupta, N. Naik, and R. Raskar, "Designing neural network architectures using reinforcement learning," *arXiv preprint arXiv:1611.02167*, 2016.

[38] C. Liu, B. Zoph, M. Neumann, J. Shlens, W. Hua, L.-J. Li, L. Fei-Fei, A. Yuille, J. Huang, and K. Murphy, "Progressive neural architecture search," in *Proceedings of the European Conference on Computer Vision (ECCV)*, 2018, pp. 19–34.

[39] M. Tan, B. Chen, R. Pang, V. Vasudevan, and Q. V. Le, "Mnasnet: Platform-Aware Neural Architecture Search for Mobile," *arXiv preprint arXiv:1807.11626*, 2018.

[40] F. Chollet, "Xception: Deep learning with depthwise separable convolutions," *arXiv preprint*, 2016.

[41] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," in *ICML'17*. ACM, 2015, pp. 448–456.

[42] P. Erdős and A. Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hung. Acad. Sci*, vol. 5, no. 1, pp. 17–60, 1960.

[43] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[44] J. Kleinberg, "The small-world phenomenon: An algorithmic perspective," Cornell University, Tech. Rep., 1999.

[45] D. J. Watts, "Networks, dynamics, and the small-world phenomenon," *American Journal of sociology*, vol. 105, no. 2, pp. 493–527, 1999.

[46] M. E. Newman and D. J. Watts, "Renormalization group analysis of the small-world network model," *Physics Letters A*, vol. 263, no. 4-6, pp. 341–346, 1999.

[47] B. Hendrickson and T. G. Kolda, "Graph partitioning models for parallel computing," *Parallel computing*, vol. 26, no. 12, pp. 1519–1534, 2000.

[48] Wikipedia, "Hypergraph," https://en.wikipedia.org/wiki/Hypergraph, 2019, [Online; accessed 12/11/19].

[49] G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, "Multilevel hypergraph partitioning: applications in vlsi domain," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 7, no. 1, pp. 69–79, 1999.

[50] A. Krizhevsky, V. Nair, and G. Hinton, "Cifar-10 (canadian institute for advanced research)."

[51] M.-E. Nilsback and A. Zisserman, "Automated flower classification over a large number of classes," in *Proc. of ICVGIP*, 2008.

[52] M. Wistuba, A. Rawat, and T. Pedapati, "A survey on neural architecture search," *arXiv preprint arXiv:1905.01392*, 2019.

[53] A. Hagberg, P. Swart, and D. S Chult, "Exploring network structure, dynamics, and function using networkx," Los Alamos National Lab.(LANL), Los Alamos, NM (United States), Tech. Rep., 2008.

[54] A. Paszke, S. Gross, S. Chintala, G. Chanan, E. Yang, Z. DeVito, Z. Lin, A. Desmaison, L. Antiga, and A. Lerer, "Automatic differentiation in pytorch," 2017, https://pytorch.org.