A Computational-Graph Partitioning Method for Training Memory-Constrained DNNs

Fareed Qararyah[a], Mohamed Wahib[b], Doğa Dikbayır[c], Mehmet Esat Belviranli[d], Didem Unat[a]

[a]*Koç University, Turkey*
[b]*National Institute of Advanced Industrial Science and Technology, Japan*
[c]*Michigan State University, USA*
[d]*Colorado School of Mines, USA*

Abstract

Many state-of-the-art Deep Neural Networks (DNNs) have substantial memory requirements. Limited device memory becomes a bottleneck when training those models. We propose ParDNN, an automatic, generic, and non-intrusive partitioning strategy for DNNs that are represented as computational graphs. ParDNN decides a placement of DNN's underlying computational graph operations across multiple devices so that the devices' memory constraints are met and the training time is minimized. ParDNN is completely independent of the deep learning aspects of a DNN. It requires no modification neither at the model nor at the systems level implementation of its operation kernels. ParDNN partitions DNNs having billions of parameters and hundreds of thousands of operations in seconds to few minutes. Our experiments with TensorFlow on 16 GPUs demonstrate efficient training of 5 very large models while achieving superlinear scaling for both the batch size and training throughput. ParDNN either outperforms or qualitatively improves upon the related work.

Keywords: DNN, graph partitioning, model parallelism

[1]

1. Introduction

Deep Learning (DL) is being increasingly applied in a wide range of scientific and engineering domains. DNNs have doubled in size roughly every 2.4 years due to the ability of larger models, *deeper* or *wider* or both, to produce results with higher accuracy on more complex tasks [1]. This growth is expected to continue in the coming years [2, 3]. The deepening and/or widening of these models comes at a cost of larger memory required to store the parameters and the intermediate results[4]. An example from computer vision field is Wide Residual Network [5], a widened variant of the well-known Resnet [6], widening the model 8 times increases the number of its parameters ~ 60 times [5] leading to a substantial increase in the memory requirements. The same trend shows up in the NLP field where deep-stacked LSTMs [7] or attention layers [8] often give more accurate results compared to shallower models. Introducing residual connections among the layers in a stack enabled the training of very deep encoder and decoder networks, e.g. larger versions of the Transformer model [8], with newer models pushing the number of parameters up to $O(10B)$ [9, 2].

Different approaches have been proposed to tackle the issue of training very large models on multiple devices. One approach is to work on the model level, where the model is partitioned across multiple devices through model, pipeline, channel parallelism, or combinations of them [10, 11, 12, 13, 14, 9, 15]. Even though these methods are successful to some extent, they suffer from either: (a) being not generic as they target a specific class of DNNs, (b) introduce non-negligible memory overhead to maintain the statistical efficiency, or (c) can incur a high implementation cost and necessitate detailed understanding of the DNN model for an accurate cost model. Another approach works at the systems level by partitioning the computational graph that represents the operations in a neural network model and distributes it over multiple devices. The existing work in this direction has some limitations. The method proposed in [16] has a restricted applicability because it relies on a descriptive language to specify computations and cannot describe all the operations used in DL. Others propose a reinforcement learning-based approach, which is impractical in many cases due to substantial resource and time requirements [17, 18]. In [19] authors propose a set of practical, generic, and low overhead heuristics to partition the DNN graph. They concluded that critical path-based approaches yield the best performance. However, their evaluation is based on an event-based simulation rather than on an actual DL framework.

We adopt the system-level approach and propose a generic, efficient, and non-intrusive partitioning strategy (ParDNN) that avoids the drawbacks of the related work. ParDNN directly works on the computational graph representation of the neural network adopted by the most popular general-purpose DL frameworks such as TensorFlow [20] and MXNet [21]. Operating on the graph level has three main benefits. First, it provides a fine-grained view of the model, which gives more parallelization options and allows better load balancing and resource uti-

lization. Second, it isolates our strategy from the details of the learning process, what provides more generality and guarantees unaffected statistical efficiency [14] of the model. Third, working at the level of the graph enables us to leverage decades of work on graph partitioning and static scheduling (as will be discussed later).

ParDNN's strategy is composed of two main steps. First, we cluster the operation-nodes of the computational graph into K partitions, where K represents the number of the available devices. The objective of this step is to reduce the end-to-end runtime by assigning the operations on the partitions such that the computational loads are balanced and the communication is minimized. In the second step, we check whether the memory constraints are met in each partition. If they are not, we reassign some operations to different partitions such that the reassigned operations have the least possible perturbed effect on the placement generated by the first step but at the same time meet the memory constraints.

Most existing graph partitioning libraries are designed to handle undirected graphs. Extensive experimenting done in [18, 17] with state-of-the-art graph partitioning-based tools, such as Scotch static mapper [22, 23] and MinCut optimizer, shows that they result in 2 to 10 times slowdown when applied on directed graphs of DL models. Our algorithm outline is inspired by the principle of the multilevel approach used in graph partitioning [24] but the design and algorithmic details of ParDNN includes a mix of variants of static scheduling heuristics [25] that are mutated to reduce the time complexity, and novel techniques to address some shortcomings in the existing ones [26, 27]. Our contributions are:

- We propose a novel computational graph partitioning method that enables training models with large memory consumption on a set of devices with limited memory.
- We conduct extensive related work comparisons with large DNNs: (a) ParDNN outperforms other graph-based approaches such as linear clustering [25] and a critical path-based method [19], (b) ParDNN outperforms Mesh-TensorFlow, a state-of-the-art distributed training framework [28] as well as having qualitative advantages over it by automating the partitioning and not requiring model rewrite. (c) It generally outperforms redundant recomputation methods (Gradient Checkpointing [29]).
- For models that do not fit into a single GPU's memory, ParDNN enables training models having up to 5.1 billion parameters using only 4 GPUs. For models that barely fit into a single device memory, it allows more efficient training by superlinearly scaling the batch size, and in many cases, the training throughput.
- ParDNN's overhead is negligible. For a graph having hundreds of thousands of nodes representing DNNs with billions of parameters, it takes $\sim$ 2 minutes to find a partition for 16 GPUs, while training these models takes days or even weeks.
- To the best of our knowledge ParDNN is the first of its type that permits the training of models that do not fit into a single device memory while being generic due to (a) having zero dependency and requiring no knowledge about the DL aspects of the models, and (b) not requiring any modifications of the model or the operation kernels.

2. Background

Modeling a computation as a directed graph has been adopted in scheduling theory [30], in parallel programming and run-time environments [31, 32, 33, 34], and recently in DL frameworks [20, 21, 35]. TensorFlow uses a stateful dataflow graph to represent a computation. It extends the classical dataflow graph model to allow maintaining and updating the persistent state of some special nodes, branching, and loop control. In a TensorFlow graph $G = (V, E)$, each node $n \in V$ represents the instantiation of an operation (e.g., matrix multiplication or convolution) and it has zero or more inputs and zero or more outputs. Each edge $e \in E$ represents a dependency between its incident nodes. Normal edges represent the data flowing between the nodes, while special edges, e.g. control dependencies, are used to enforce happens-before relationships with no data flows along them [20].

Graph partitioning is, in general, defined as splitting the graph $G(V, E)$ into K disjoint subsets [36]. The constrained version of the graph partitioning aims at partitioning in such a way that the sums of the vertices weights in each set are as equal as possible, and the sum of the weights of edges crossing between sets is minimized [24]. An extension of general graph partitioning which aims to assign a set of communicating tasks to processors is called *static mapping* [36]. Static mapping does not consider the logical and temporal dependencies of the tasks, it is assumed that all the tasks simultaneously coexist throughout the program execution.

Finding a spatial and temporal assignment of the set of nodes in a task graph $G = (V, E)$ onto a set of processors resulting in the fastest possible execution, while respecting the precedence constraints expressed by all $e \in E$ is referred to as task scheduling problem [30]. The schedule length, *makespan*, is the completion time (C_t) of the last node in G assuming that the graph execution starts at time 0. Where $C_t(n)$ is the time required to execute the operation represented by n added to the time at which this operation starts to execute. The goal is to minimize C_{tmax}, where $C_{tmax} = max_{n \in V} C_t(n)$. Finding an optimal schedule or static mapping is *NP-hard* [36, 30].

3. ParDNN: A Partitioning Strategy for DNNs

ParDNN works at the computational graph level and offers a practical, non-intrusive, and generic method to partition a neural network model on a set of processing elements(PE).

The main objective of ParDNN is to minimize C_{tmax}, the makespan of the graph, while satisfying the memory capacity constraints of the target processing elements. It is important to mention that ParDNN does not have a runtime component. All

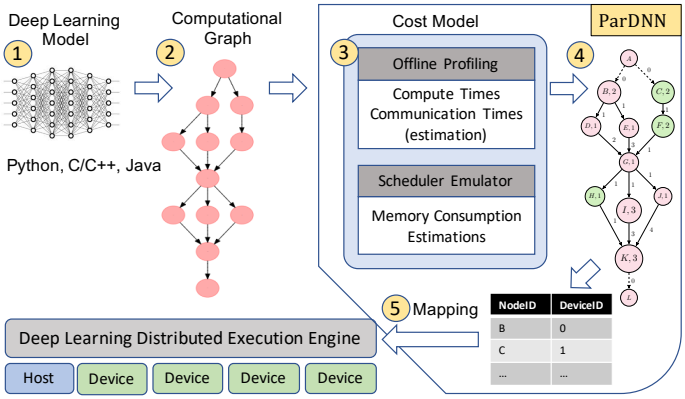


Figure 1: ParDNN Overview

Table 1: Complexity of Each Step of ParDNN

Step-1	Partition to Minimize Makespan
Graph Slicing (inc. sorting)	$O(K(\|V\|+\|E\|))$
Mapping	$O(\|V\|*log^2\|V\|)$
Step-2	Memory Heuristic - I
TensorFlow Scheduler Emulator	$O(\|V\|+\|E\|)$
Memory Consumption Tracker	$O(\|V\|)$
Overflow Handler	$O(\|V^2\|)$
Step-2	Memory Heuristic - II
Residual Nodes Movement and CP splitting	$O(\|V\|)$
Overall ParDNN Complexity (w. Heuristic-I)	$O(\|V\|^2)$
Overall ParDNN Complexity (w. Heuristic-II)	$O(\|V\|*log^2(\|V\|) + K\|E\|)$

the steps of ParDNN are done ahead of time. After running ParDNN once, the resulting partitioning can be used as long as the model parameters that affect the memory consumption do not change.

Figure 1 shows the overall process. ParDNN takes a computational directed acyclic graph as an input, it annotates this graph with computation, communication, and memory consumption information gathered using offline profiling. It adds an artificial source and sink node to the graph. Using the collected data, ParDNN splits the graph into parts to be mapped to processing elements. ParDNN outputs the mapping information to be used by the execution engine of the DL framework (e.g., TensorFlow).

Our strategy is divided into two major steps. Step-1 aims to obtain a partitioning that has a minimal makespan. Step-1 is further divided into two stages. Stage-I, *graph slicing* splits the graph into K disjoint primary and S disjoint secondary clusters. This splitting enables working at a coarser level in the upcoming stages. Stage-II, *mapping*, merges these S secondary clusters into the K primary clusters using a novel mapping algorithm. In Step-2, we propose two alternative heuristics to overcome the memory overflow; one is threshold-based and the other is balancing-based. In heuristic-I, the result from Step-1 is validated against the memory constraints of the given devices. If the memory constraints are satisfied, the partition will be the final output. Otherwise, nodes are moved between the partitions until the memory consumption by a processing element pe at any time $t \in [0, C_{tmax}]$ is less than or equal to pe's memory capacity. On the other hand, heuristic-II tries to balance the memory consumption across the available processing elements. Heuristic-I is fully applied in Step-2 to avoid higher time and code complexity; applying it in Step-1 requires continuous backtracking as the movement of nodes affects the temporal load balance and communication cost. Moreover, this gives a global view of the graph and wider range of moves to mitigate the memory overflow. Heuristic-II is partially done in Step-1 as explained in Section 3.2.2.

Table 2: Terminology used in this work

Term	Description
$G = (V, E)$	Computational graph with vertex set V, edge set E
CP	Critical path of a graph, the longest path in the graph considering computation and communication costs.
C_{tmax}	Makespan of G, schedule length
PE, pe	Set of processing elements, a processing element
K	Number of processing elements (e.g., # of GPUs)
$comp(n)$	Weight of a node n ; time required to execute that node on a processing element.
$mem(n)$	Memory consumption of outputs of a node n
$comm(e)$	Cost of an edge e ; time required to communicate the data from the source node (operation) to the destination node (operation).
sc	Secondary cluster, which is a node or a path
$comm(sc)$	Total communication cost incurred by all edges that have one end in sc
$tl(n)$	Node top level: length of the costliest path between the the source node of the graph and the node n, excluding the node n. Where the length of a path, is the summation of the computation costs of the nodes on the path and the communication cost of its edges $\sum_{n \in p} comp(n) + \sum_{e \in p} comm(e)$
$bl(n)$	Node bottom level: length of the costliest path between n and the sink node including the node n
$w\_lvl(n)$	Node weighted level: $tl(n) + bl(n)$
$span(sc)$	Time between the expected finish time of the last parent of the first node in a sc, and the expected starting time of the first child of the last node in that path. Last and first here mean topologically.
$potential(sc)$	Summation of the weights of all nodes that can be executed within $span(sc)$
$st(n)$	Starting time of node n, the time when n is assigned to a pe to execute
$ft(n)$	Finish time of node n, the time when pe is done with executing n
$M_{cons}(pe, t)$	Memory consumed by the processing element at time t
$M_{pot}(n, t)$	Memory potential of a node n at time t. The summation of the memory occupied by the outputs of n's direct ancestors that are executed before t, and for which n is the last direct descendant in its pe. Plus n's memory consumption if $st(n) \leq t \leq ft(n)$

Table 1 summarizes the time complexity of each step of ParDNN. The reported complexities after each step are relaxed ones and for some stages a tighter bound maybe driven with amortized analysis. In practice, the average running time of ParDNN on the DNN models listed in Table 3 is roughly 2 minutes on a typical laptop processor, namely an Intel i7-7600u CPU @ 2.80GHz. Considering the training time of those models is in the orders of days or even weeks, ParDNN offers an extremely lightweight and practical approach to partition the DNN graphs.

Next, we explain the details of each step along with the time complexity. Table 2 summarizes the terms and notations for the explanations.

Algorithm 1 Graph Slicing

```
   In : K, Graph G
   Out: pri_clusters[ ], sec_clusters[ ]                ▷ initially empty
 1: j ← 1
 2: w_lvls ← compute_weighted_levels(G)
 3: while G ≠ φ do
 4:     heaviest_path ← find_heaviest_path(G, w_lvls)
 5:     if j ≤ K then
 6:         pri_clusters[j] ← heaviest_path
 7:         w_lvls ← compute_weighted_levels(G)
 8:     else
 9:         sec_clusters[j − K] ← heaviest_path
10:     end if
11:     G ← G − {heaviest_path}
12:     j ← j + 1
13: end while
```

3.1. Step-1: Partitioning To Minimize Makespan

This step aims at reducing the makespan of the graph. Before presenting the details of the step, it is important to point its distinction from both static task scheduling and static mapping. Unlike scheduling algorithms, we do not specify an order of task execution; we rather focus on spatially allocating the tasks on a set of processors while addressing the locality-parallelism trade-off. The order of execution decision is left to the runtime dynamic scheduler, e.g., TensorFlow scheduler. Unlike static mapping, ParDNN considers the logical and temporal dependencies between the tasks.

The size, ($|V|$), of a large DNNs' computational graph is usually in the order of hundreds of thousands and is projected to grow to millions of operation-nodes [2]. As a result, efficiency and scalability are essential features of any proposed solution. The multilevel method, the most widely used technique in graph partitioning, addresses the scalability issue by grouping vertices together and dealing with groups of vertices, rather than individual ones [36]. This grouping reduces the problem size and allows good quality heuristics to be applied within a reasonable time. Inspired by this method, we designed Step-1 of our partitioner in two main stages.

3.1.1. **Graph slicing**

This stage groups the nodes of the graph into disjoint clusters. It iteratively finds the critical path (CP) in the graph and removes CP's nodes and their incident edges from the graph by marking them as visited so that they are not explored in the following iterations. This is repeated K times and the resulting K many CPs are called *primary clusters*, which are the initial partitions assigned to different processing elements. Hence, the terms primary cluster and *pe* are going to be used interchangeably. After finding those primary clusters, the remaining nodes are grouped into *secondary clusters*. A secondary cluster, which is a linear cluster [30], is either a single node or a path. All the secondary clusters are identified and tagged until there is no node left on the graph that is not part of any cluster. Figure 2(b) shows an example.

Algorithm 1 shows the pseudo-code of the graph slicing, which takes device count K and graph G as inputs and outputs primary and secondary clusters. Line 2 computes the weighted level ($w\_lvl(n)$) for all the nodes in the graph. The heaviest path, (Line 4), is the CP when $w\_lvl(n)$ are recalculated. Finding the heaviest path is done by traversing the graph using the computed w_lvls as priorities until reaching a dead-end. After forming a CP, it is added to the primary clusters and its nodes and edges are removed from the graph (Line 11). Unlike linear clustering [25], we obtain only K many CPs, then we stop recalculating $w\_lvl(n)$ for the secondary clusters since computing weighted levels is expensive. In section 5.3.1 we demonstrate that avoiding this expensive computation does not harm the quality of the results. When weighted levels are not recalculated, $find\_heaviest\_path$ may not return a CP, rather returns a path of a heavy cost. Thus the term *heaviest path* refers to the heaviest path from the slicing algorithm perspective, which is not necessarily the actual critical path. If a path could not be obtained, it returns a single node.

Complexity: The most expensive part of Algorithm 1 is computing weighted levels for all the nodes. This operation performs a variant of topological sorting and has time complexity of $O(|V|+|E|)$ [30]. It is done K times, resulting in an overall complexity of $O(K(|V|+|E|))$. In linear clustering that would cost $O(|V|(|E|+|V|))$ [37]. Given that the priorities are already specified, finding the paths costs $O(|E|)$[2]. This is because each node is visited at most once, since the nodes are removed from the graph (marked as visited) once they join a path. From each node deciding the next to visit entails picking its highest priority neighbor, which requires checking its adjacent nodes' priorities. Since the edges are removed from the graph with their incident vertices as well, no edge will be visited twice. Hence, overall there are $O(|E|)$ steps. It is important to note that all the graphs we experimented are sparse, having $|E|< |V|log(|V|)$.

3.1.2. **Mapping**

This stage attaches the secondary clusters to the primaries with the goal of obtaining a partition with minimal makespan by addressing the locality parallelism trade-off. This process is referred to as *cluster mapping* in the scheduling literature. There are mapping heuristics such as wrap cluster merging [38], list scheduling based cluster assignment [39], and Guided Load Balancing (GLB) [27]. In a comprehensive evaluation of scheduling and cluster merging algorithms in [37], GLB is shown to produce the best result. However, GLB assumes that its preceding clustering step has eliminated the largest communication delays. As a result, the communication delays are not considered for cluster mapping [27]. Ignoring communication cost results in a low-quality mapping when the graph becomes very large. Even if each inter-cluster communication is small, the cumulative effect becomes considerable. In addition, the load balancing is global rather than time dependent (temporal). This balancing is not suitable especially for graphs with frequent forks and joins (e.g., DNN graphs), where the local and the global loads become more uncorrelated. We propose a novel time-efficient heuristic called *Locality-First*

[2] The task graph is assumed to be connected graph, which means it has at least $n - 1$ edges

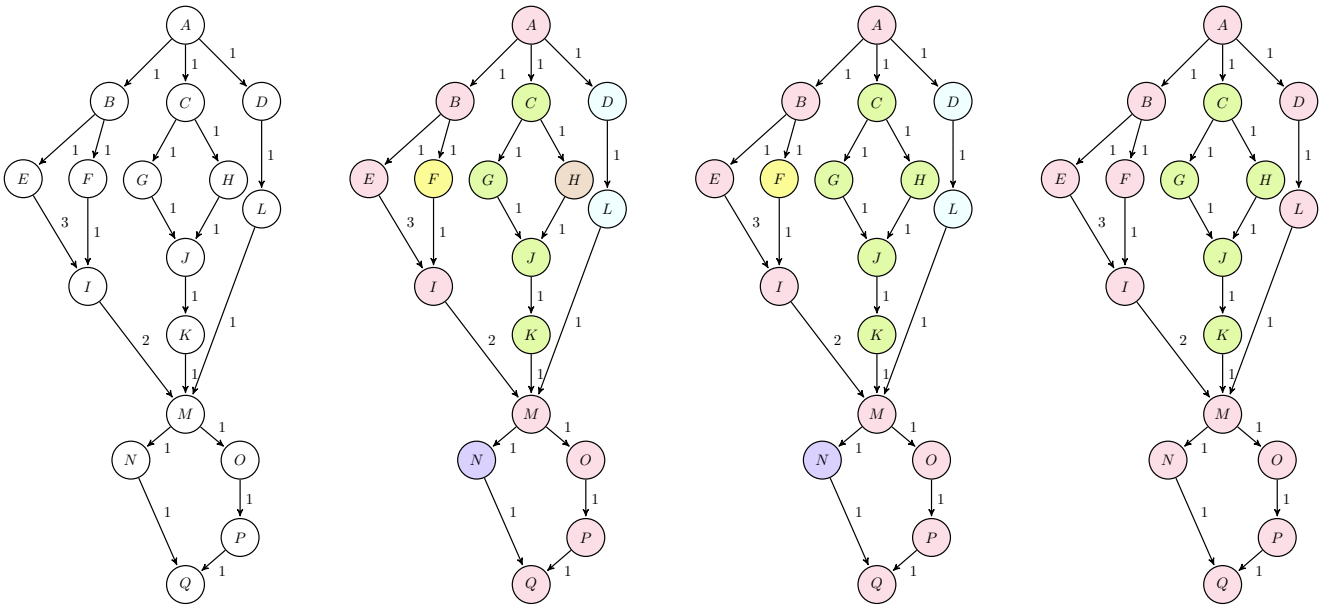


Figure 2: In the computational graph, edge weights indicate communication costs. All nodes are assumed to have a weight of 1 for simplicity. **(a)** Original computational graph. **(b)** Shows the slicing stage when there are two *pe*(s). The obtained clusters are: {*A*, *B*, *E*, *I*, *M*, *O*, *P*, *Q*}, {*C*, *G*, *J*, *K*}, {*H*}, {*N*}, {*F*}, {*D*, *L*}. First two are primary clusters, the other four are secondaries. **(c)** Locality First Lookahead Mapping: Secondary clusters sorted by their criticality, in decreasing order, are {*H*}, {*N*}, {*F*}, {*D*, *L*} (note that communications inside a cluster are considered to be zero). {*H*} is totally-communicating with a primary and there is enough unmapped work within its span (*L* and *F*) to cover the temporal/local imbalance. {*F*} is totally-communicating with a primary and its mapping does not cause an imbalance (keeping in mind that H is already mapped to the green primary). Both {*D*, *L*} and {*N*} are totally-communicating as well, but their mapping causes an imbalance and there is no unmapped work left within their spans to cover. **(d)** Level Aware Load Balancing: {*D*, *L*} and {*N*} are mapped according to the criteria in equation 1. In th case of {*N*} there is a tie (equation 1 result is 2 for both of the primaries), so it is assigned to the one with which it communicates the most. The makespan is 13.

Lookahead and Level-Aware Mapping (LFLAM). LFLAM considers both critical-communication minimization and the temporal load balancing.

Locality-First Lookahead Mapping. First, we perform locality-first lookahead mapping, which assigns a secondary cluster *sc* to the primary with which it communicates the most as long as there is unmapped work (other tasks ahead) within *span*(*sc*) that can cover the temporal load imbalance caused by that mapping step. The intuition is that if both the balance and the locality are considered in the cost function at the beginning of the mapping, the locality might be sacrificed in some cases to account for balance. In such a case, an additional refinement step would be needed to improve locality. Such a refinement step would swap some of the already mapped clusters so that the locality is regained without hurting the load balance. However, optimizing for the locality first, knowing that there is a sufficient amount of tasks ahead to balance again, diminishes the need for such a refinement step. On the other hand, overprioritizing locality may harm parallelism. Hence, it is crucial to decide which paths should be targeted. Moreover, some communications are less important to eliminate than others. For example, a communication edge between two nodes that have enough amount of work to hide (e.g., (*A*, *D*) *and* (*L*, *M*) in Figure 2 b) is not as important as communication that can be a pure delay (e.g., (*H*, *J*)).

We decide which paths to be targeted based on *CCR* (total communication cost to total computation cost ratio) of the graph [40, 30]. Higher *CCR* means more communication compared to computation, hence locality is more important. In the literature [30], graphs with $CCR \geq 10$ are considered as highly communicating. Hence we use the *CCR* value as a threshold to decide to what extent to prioritize the locality. We divide the secondary clusters into two groups: totally-communicating and maximally-communicating clusters. Totally-communicating clusters are the ones whose all the incoming and outgoing communications are with only one of the primaries. A maximally-communicating cluster is the one whose communication with one of the primaries is greater than its all communication/*K*. Note that this does not always hold since an *sc* can be communicating with other secondaries not only with primaries. For graphs which have equal or higher *CCR* value than the threshold, clusters from both totally-communicating and maximally-communicating are considered because we need to save as much communication as possible. Otherwise only totally-communicating clusters are considered. These clusters should be added to their primaries even when *CCR* is low as long as there is unmapped work within *span*(*sc*) that can cover the possible temporal load imbalance because such mapping is a pure gain as it would cancel all their communications.

We decide the most important communication edges among clusters based on their *criticality*. The criticality of a linear

cluster *lc* is the length of the longest path going from the graph source to its sink and completely overlapping with *lc*. This is equivalent to the $w_{lvl}(n)\ \forall n \in lc$, where $w_{lvl}(n)$ is recalculated by setting communications within *lc*s to zeros after the slicing stage of ParDNN. The clusters are sorted by their criticality in a non-increasing order. Then the desired clusters, depending on the *CCR* threshold, are traversed and any secondary cluster is mapped to its most communicating primary if any of three conditions holds: (a) *lc* has enough unmapped work within its span to cover the local load imbalance of its mapping if any is caused by the mapping, (b) if no local load imbalance occurs or the local balance is improved, and (c) *lc* communication with the target primary is larger than its weight, the work of the primary within its span, and the unmapped work in its span. In the case of condition (c) if mapping were not done, the communication would dominate potentially creating a new longer critical path.

Level-aware Load Balancing. Next, level-aware load balancing is applied to map the clusters that were not mapped in the previous step. Temporal balance of the loads is achieved by considering the workload of every *pe* within *span*(*sc*), where *sc* is the secondary cluster that is going to be mapped to one of the primary clusters. *sc* is mapped to a *pe* that has the minimal computational load within the *span*(*sc*), and minimizes the incurred communication with the other processing elements. Equation (1) shows the selection criteria. In case of ties, we assign *sc* to the *pe* which has the highest communication value with it.

$$\min_{pe \in PE} \Big(\sum_{\substack{n \in pe, \\ tl(n) \in span(sc)}} comp(n) + \sum_{\substack{(n,u) \in E, \\ n \in \{PE\}-pe, \\ u \in sc}} comm(n,u) + \sum_{\substack{(u,n) \in E, \\ n \in \{PE\}-pe, \\ u \in sc}} comm(u,n) \Big) \quad (1)$$

Algorithm 2 shows a high level description of mapping. Lines 1-7 perform the Locality-First Lookahead Mapping. *locality_first_lookahead_mapping* takes the secondary clusters as a parameter and decides whether to map the maximally-communicating clusters or not depending on the *CCR* value. It then maps the clusters based on the three conditions we explained in the previous paragraph and returns the number of the mapped clusters. Line 6 could be repeated as long as at least one path has been mapped since the mapping changes the communication. This possibly leads to the formation of new totally-communicating or maximally-communicating paths. We choose to repeat this step at most $log(|V|)$ times to maintain low complexity. The while loop (in Line 10) applies the level-aware load balancing. The most time consuming part in this algorithm is to calculate the work within the span of the target secondary cluster *sc* both as a part of *locality_first_lookahead_mapping* and on (Line 12) in each of the primary clusters. We model this part as a problem of frequent range queries with updates. More specifically, we need to find the sum of the weights of the nodes whose levels fall in the span, and upon merging, the weights of those levels are updated. We use binary-indexed-trees [41] as a data structure, where the tree nodes store the weights per level. This data structure allows logarithmic range summation and value updates.

Algorithm 2 LFLAM Mapping

```
   In: pri_clusters[ ]
   In: sec_clusters[ ]
 1: w_lvls ← compute_weighted_levels(G)
 2: sort(sec_clusters, criticality_sorting_criteria)
 3: num_of_mapped_lcs ← 0
 4: iteration ← 0
 5: do
 6:     num_of_mapped_lcs ← locality_first_lookahead_mapping(sec_clusters)
 7: while num_of_mapped_lcs > 0 and iteration ≥ log(|V|)
 8:
 9: comps[ ] ← φ, comms[ ] ← φ
10: while sec_clusters ≠ φ do
11:     sc ← remove_next_secondary(sec_clusters)
12:     comps ← calc_work_at_span(span(sc), pri_clusters)
13:     comms ← calc_comms_with(sc, pri_clusters)
14:     target_pri ← find_minimal(comps, comms)
15:     target_pri ← target_pri + {sc}
16: end while
```

Line 13 calculates the cost of communication between the secondary cluster *sc* in each of the primary clusters. Line 14 performs the selection criteria defined in Equation (1) to select the best primary cluster to merge the *sc* with.

Complexity: Before starting LFLAM, we sort the clusters by their weights, this has an upper bound of $O(|V|*log|V|)$ since the number of clusters is upper-bounded by the number of nodes. Since the clusters are disjoint paths or singular nodes, and have no common nodes, the total number of the update operations is bounded by $|V|$. The number of range summation queries is bounded by the number of the paths which is again bounded by $|V|$. The cost of either of the operations is logarithmic in the number of the levels. The number of the levels is $\leq |V|$, so we end up with $O(|V|*log|V|)$. The Locality-First Lookahead Mapping is repeated $log(|V|)$ times. Hence, the overall complexity of the mapping stage is $O(|V|*log^2|V|)$.

3.2. Step 2: Handling Memory Overflow

Similar to Step-1, we handle the memory constraints statically ahead of time for two main reasons: (a) to avoid any runtime overhead, and (b) to reduce the chance of conflicting with other runtime optimizations. We propose two memory heuristics both of which could be seamlessly used with any optimization policies provided by the DL frameworks. First heuristic is emulation-based and guarantees to meet the memory capacity constraint. The second is a practical method that aims to balance memory consumption on devices.

3.2.1. Memory Heuristic I

This heuristic has three components; scheduler emulator, memory consumption tracker, and overflow handler.

Scheduler Emulator. To address the memory consumption statically, temporal modeling of the allocation and deallocation patterns is required. Such modeling necessitates knowledge about scheduling in the DL framework to estimate when an operation is going to start and finish execution. Consequently, when the memory allocated for the operation inputs is released and when a new memory is allocated to hold the operation outputs. To estimate those values, we emulate TensorFlow scheduler. TensorFlow scheduler maintains a ready queue that is initially filled with nodes with no ancestors. Each node in the

graph has an in-degree representing the number of nodes it depends on. The nodes are executed in FIFO order. Once a node is executed, the in-degrees of its children are decremented by one. Any node having an in-degree of zero will be pushed to the queue. Using the per-node running times and communication sizes collected from profiling, we emulate this behaviour to get the expected start- and end-times of the operations under a certain partitioning.

Complexity: The scheduler emulator estimates the starting time $st(n)$ and finishing time $ft(n)$ of the nodes in the graph. The emulation has a time complexity of $O(|V|+|E|)$ as the nodes are visited and on each visited-nodes, the in-degrees of its direct descendants are decreased.

Memory Consumption Tracker. In TensorFlow, from the memory consumption perspective, operation-nodes broadly fall into three main categories. First, operations of which the data survives across the iterations [20] and we refer to them as *residual nodes* (res_ns). Second, special operations that mutate the referenced tensor, of the first type, we refer to them as *reference nodes* (ref_ns). Those operations do not reserve any additional memory. However, they are co-located with the variables that they are mutating and must be moved together with their referred to variable nodes. Third, operations that require additional memory proportional to their output size and we call them *normal nodes* (nor_ns). Memory for the output of these nodes is allocated upon scheduling and released once all their direct descendants are executed.

To create a functional cost model, our memory consumption estimation takes into account the scheduler, node types, and profiled per-node memory consumption. One might think that profiling solely peak memory footprints would be sufficient to predict the overflows. However, to handle an overflow, nodes have to be moved between partitions and that in turn changes the schedule and the memory consumption as a function of time. Our cost model takes this dynamic behavior into account and models the interplay between the scheduler and memory usage.

Equation(2) defines the memory consumption of a *pe* at time t, $M_{cons}(pe,t)$. The first term is the memory consumption of the res_ns assigned to that *pe*. The second term indicates the memory consumption of the normal nodes that have started on that *pe* at $\leq t$ and are being executed at t. The third indicates the nodes that have descendants assigned to that *pe* and the descendants' expected starting time is $\geq t$, and those nodes have finished at $\leq t$ at any processing element except *pe*, or are non-residual that have finished at $\leq t$ on that *pe*.

$$M_{cons}(pe,t) = \sum_{\substack{n\in pe,\\ n\in res\_ns}} mem(n) + \sum_{\substack{n\in pe,\\ n\in nor\_ns,\\ st(n)\leq t\leq ft(n)}} mem(n) + \sum_{\substack{n\notin(pe\cap res\_ns),\\ ft(n)\leq t,\\ \exists(n,u)\in E: st(u)\geq t, u\in pe}} mem(n) \quad (2)$$

The overall memory consumption needs to be estimated for each node ($|V|$ time points) because the change in memory consumption is triggered by node executions. Once a node starts executing, new memory space needs to be allocated and that may cause an overflow. Estimating memory consumption is done by visiting all the nodes in the graph in the order of their estimated starting times, which is obtained from the scheduler emulator, and keeping track of the accumulated memory consumption. In the same pass, the memory potential values of the nodes (M_{pot} in Table 2) are obtained. A node's memory consumption is added to the cumulative value once it is visited, and subtracted after its last descendent in a certain *pe* is visited unless it is a res_ns.

Complexity: Tracking the memory consumption requires $O(|V|)$ time since it is done in one pass over the graph nodes while keeping the cumulative values and calculating the potentials. This is given that the node last descendant assigned to each processing element is knows; which is collected while emulating the scheduler and hence its complexity is implicitly included in the scheduler emulator part.

Overflow Handler. After estimating the memory consumption, we traverse the graph starting from the sink and keep the nodes in a heap data structure, namely $nodes\_heap$. When the memory consumed exceeds the limit, we deal with the overflow as a 0-1 min-knapsack problem [42]. The min-knapsack problem is formulated as follows; given N pairs of positive integers (c_j, a_j) and a positive integer O, find x_1, x_2, ..., x_N so as to:

$$minimize \sum_{j=1}^{N} c_j x_j \qquad s.t. \sum_{j=1}^{N} a_j x_j \geq O \text{ and } x_j \in \{0,1\} \quad (3)$$

In our case, O represents the amount of memory *overflow*, and a_j represents $M_{pot}(n,t)$. For the cost cj, we use the summation of the node computation cost and the communication with its direct ancestors and descendants located on the same *pe*, as shown in Equation(4), which defines $move\_cost$.

$$comp(n) + \sum_{\forall u,v\in pe(n)} \Big(\sum_{u:(u,n)\in G} comm(u,n) + \sum_{v:(n,v)\in G} comm(n,v) \Big) \quad (4)$$

The idea behind $move\_cost$ is that when a node is moved from a *pe* to another, it incurs a computational load imbalance proportional to its weight and extra communication proportional to its communication with the nodes assigned to the same *pe*. Our goal is to find a set of operation-nodes that the summation of their memory consumption potentials at the overflow time is $\geq$ *overflow* when their total movement cost is minimized. The *movement criteria* is to pick the node that has the lowest $move\_cost/M_{pot}(n,t)$. In other words we choose the node that alleviate the overflow while incurring the least amount of communication and computation imbalance.

The $nodes\_heap$ is a min heap in which the *movement criteria* is the ordering key. To avoid choosing a node that has a low *movement criteria* but high $move\_cost$, each node for which the $M_{pot}(n,t) >$ *overflow* is inserted in another heap at which the sorting key is $move\_cost$. When selecting, the top node is removed from both heaps and the one with the least $move\_cost$ is chosen, and the other is returned to its heap. The selected node is moved to another *pe* if the target *pe* has sufficient memory

to accommodate that node memory potential. Otherwise, the node is not considered again and another node is picked from the heap. The algorithm terminates when either the overflow is eliminated or we run out of nodes without addressing it.

Complexity: We solve the knapsack greedily as the dynamic programming based solution complexity is impractical. When an overflow is detected, we pick the nodes from the heaps in a logarithmic time. Any node that is moved to another partition is guaranteed not to be moved again since it is moved only if the destination *pe* can accommodate it, meaning that it can neither cause nor solve an overflow on that *pe*. As a result, there is no repetition and a node can enter or exit the heap once, resulting in $O(|V|*log|V|)$. When a node is moved, the new potentials and memory consumption need to be recalculated ($O(|V|)$). It may happen at most $|V|$ times. Overall the complexity is $O(|V|^2)$. However, this upper bound is much larger than the practical one as the number of nodes to be moved is usually much less than $|V|$.

3.2.2. Memory Heuristic-II

Heuristic-II balances the memory consumption among devices but theoretically does not guarantee meeting the memory constraint. Its advantages are (a) it does not require the emulation of the DL scheduler, and (b) it is practically efficient as it permits training all the models trainable using Heuristic-I in our experiment set.

This heuristic is based on the observation that most of the long-living memory reserved throughout an iteration (one execution of the graph), is either for a resident node holding parameters, or a normal node holding output that will be fed to a distant node in the critical path. We apply two strategies to alleviate any possible high imbalance in memory consumption between the partitions. The first balances the residual and non-residual net memory consumption ratios across the partitions. Since the memory for residual nodes, which is allocated for the model parameters, is reserved and not freed until the training ends, it forms a permanent memory pressure. Moreover, most of these nodes are not critical nodes. Only the ones used in the first few layers should be provided quickly, meaning that their communications are critical. The rest, which are the vast majority, can be communicated by the time they are needed, which makes them good candidates to move. We move these nodes from the most loaded partitions to the least loaded ones until the summation of ratio of the residual node memories in a partition to the overall memory for residual nodes and the ratio of the normal node memories in a partition to the overall memory for normal nodes are as equivalent as possible. For example, if there are two partitions, where partition 1 holds 70% of the normal nodes' memory and 60% of the residual nodes' memory (partition 2 holds 30% and 40%, respectively), then the result of applying the strategy would be partition 1 having 70% of the normal and 30% of the residual nodes' memory, and partition 2 having 30% and 70%, respectively. In other words, the residual nodes are moved until Equation 5 holds or there are no more residual nodes to move.

$$\forall pe \in PE,\ K/2\ (\sum_{\substack{n\in pe,\\ n\in res\_ns}} mem(n)\ /\sum_{n\in res\_ns} mem(n)\ + \sum_{\substack{n\in pe,\\ n\in nor\_ns}} mem(n)\ /\sum_{n\in nor\_ns} mem(n)) \geq 1 \tag{5}$$

In some cases moving the residual nodes is not enough. This happens when the graph is very thin, hence most of the nodes are located on the critical path (the first path of the K paths discovered by the slicing algorithm). This case results in a highly unbalanced partitioning because a balanced partitioning, in this case, would require splitting the critical path which is not desired as it means longer execution time. However, if the memory of a single device is not sufficient to hold the peak memory that is required to execute the critical path then the path has to be divided to be able to run the graph. So, our second strategy detects if there is a high imbalance between the primaries and gives contiguous chunks of the longest to the shortest -one chunk for each of the shortest- to balance the memory requirements. This is performed right after slicing but before mapping stages so that the Locality-First Lookahead mapping guarantees the highest level of locality around the reassigned chunks and the level-aware load balancing balances the secondaries accordingly. Moving contiguous chunks of the critical path does not cause a considerable performance degradation in the case of our large and thin graphs since the critical path of a thin DNN graph contains thousands to tens of thousands of nodes and introducing few communications has a negligible effect. Without emulation or knowing the schedule, it is not possible to know the memory consumption pattern and the bottlenecks. As a result, heuristic-II should be enabled or disabled by the user, as the user may want to train with a certain batch size permitted by the partitioning without applying a memory heuristic. For example, for the models that fit into a single device memory using a small batch size, only partitioning the model is sufficient for a linear scaling of the batch size.

Complexity: This heuristic requires simply moving nodes among partitions. Since the movement is unidirectional, from the most loaded to the least, each node is moved at most once. Hence the total number of moves is bounded by the number of nodes resulting in a time complexity of $O(|V|+|E|)$.

4. Implementation

Our algorithm takes as an input the device count, their memory capacities, the interconnection bandwidth and latency between them, the model computational graph, profiling data, operations metadata. The profiling data contain execution time measurements and the size of the output of each operation-node. The operation metadata contain the operation types (section 3.2.1). TensorFlow standard APIs provide the profiling information including per-node time, memory consumption, and communication sizes at the granularity of graph nodes for regular as well as user-defined operators.

To estimate the memory consumption, we implemented an emulator of TensorFlow's scheduler described in [20]. It is important to note that if ParDNN with memory heuristic-I is intended to be used with another DL framework, another emulator can be written to emulate its scheduler, if needed, without modifying our partitioning algorithm. When handling memory constraints there is a trade-off between the overhead and the accuracy; static handling prioritizes overhead reduction over accuracy while dynamic handling targets the opposite. Due to the efficiency and maintainability reasons discussed in section 3.2, we adopt the static approach. To accommodate sacrificing the exact details of the memory management optimizations and allocation details, such as fragmentation and temporary memory for local variables, we spare 10% of the device memory and constrain ourselves to the remaining 90%. This threshold was sufficient to successfully run all our experiments without going out of memory (OOM). Nevertheless, this ratio might need to be tuned and it is the only parameter of ParDNN that needs tuning.

As shown in Figure 1, the output of our algorithm is a single file containing model operations placement as key-value pairs. Each key is an operation-node name and the value is the device on which the operation should be allocated. To control the placement at operation-node granularity, the TensorFlow backend reads the node-to-device assignment from the placement file generated by our algorithm.

ParDNN on multiple nodes: Despite the capability of designing ParDNN to partition a DNN on multiple nodes, in this work we assume a single node where the processing elements are identical GPUs connected to a common host. This is because the number of GPUs per node has been steadily increasing over time. For instance, systems with 16 or more GPUs per node are in production (e.g. NVIDIA DGX SuperPOD). As suggested by many state-of-the-art works [9, 2, 13], we argue that a hybrid approach of data parallelism across compute nodes and using ParDNN inside the compute node is a practical choice. This approach benefits from the efficiency and non-invasiveness of our method in tackling the memory capacity issue at the node-level, while also harnessing the weak scaling properties of data parallelism across the nodes.

5. Results

This section is organized into two parts. First part compares the performance of ParDNN against related work: critical path based heuristics, explicit model parallelism, redundant recompute, and an out-of-core method. The second part evaluates the scaling of ParDNN. Key findings of each part are as follows:

Comparison with Related Work. (i) ParDNN outperforms other critical path-based heuristics for computational graph placement [25, 19]. Moreover, its empirical overhead is much lower than alternative well-performing heuristics. Replacing Step-1 of ParDNN with other heuristics results in a significant performance drop or huge increase in the overhead. (ii) ParDNN outperforms the distributed tensor computation framework, Mesh TensorFlow [28] and provides much higher user productivity. (iii) ParDNN outperforms Gradient Checkpointing [43] combined with data parallelism in many cases, yielding up to 2.7x speedup. More importantly, ParDNN enables training models where applying Gradient Checkpointing result in out of memory (OOM) even with a batch size of 1. (iv) ParDNN outperforms CUDA Unified Memory for all configurations and GPU counts.

Table 3: Specifications of Models Datasets. (C)HSD: (Character) Hidden State Dimension, SL: Sequence Length, ED: Embedding Dimensions, RU: Residual Units, WF: Widening Factor, MD: Model Dimension, FS:Filter Size, P_SZ: patch size.

Model / Dataset	Acronym	#Layers	HSD	SL	#Para. (10^9)	#Graph Nodes
RNN for Word-Level Language [44] / Tiny Shakespeare [45]	Word-RNN	8	2048	28	0.34	10578
	Word-RNN-2	32	2048	25	1.18	39074
			CHSD	ED		
Character-Aware Neural Language Models [46] / Penn Treebank (PTB) [47]	Char-CRN	8	2048	15	0.23	22748
	Char-CRN-2	32	2048	15	1.09	86663
			#RU	WF		
Wide Residual Net. [5] / CIFAR100 [48]	WRN	610	101	14	1.91	187742
	WRN-2	304	50	28	3.77	79742
			HSD	MD		
Transformer [49] / IWSLT'16 German–English corpus [50]	TRN	52	4098	2048	1.99	204792
	TRN-2	48	8192	2048	5.1	160518
		HSD	FS	P˙SZ		
Eidetic 3D LSTM[51] / Moving MNIST digits [52]	E3D	320	5	4	0.95	55756
	E3D-2	512	5	8	2.4	55756

Scaling. (i) For the same number of GPUs, ParDNN enables the use of more than 9x batch size on average over the maximum possible with data parallelism. (ii) Superlinear speedup in most models and configurations is observed going from one GPU to 16 GPUs.

Using either of the memory heuristics gives similar performance (running time). Hence, we only report the results with the memory Heuristic-II for the sake of brevity.

5.1. Environment, Models, and Datasets

We conducted all our experiments on a NVIDIA DGX-2 with 16 Tesla V100 SXM3 32GB GPUs connected via NVSwitch. The throughput measurements are conducted over the interval between the 100^{th} and the 150^{th} training iterations to get stable results. We use TensorFlow 1.15 and CUDA 10.0.

5.2. Models and Datasets

To demonstrate our results we experimented with five large models representing four main tracks of DL applications: image classification, translation, video prediction, and language modeling. All models and datasets used in the experiments are listed in Table 3. We focus our analysis on the performance of ParDNN, rather than pursuing the accuracy as ParDNN has no effect on the learning aspect of the model. More specifically, the convergence and accuracy are mainly affected by changing the batch size and other hyper-parameters, and our algorithm does not alter the model and its hyper-parameters in any fashion. ParDNN changes the placements of the operations on devices after the computational graph has already been generated by the framework.

We use **Word-RNN** a multi-layer Recurrent Neural Network for word-level language inspired by the character-level

modeling [53]. Character-Aware Neural Language Models (**Char-CRN**) [46]. Both models can be enlarged by increasing the number of layers or the hidden state size. **WRN** [5] is a widened version of the residual network model. In WRN the number of residual units and the width of the convolutional layers can be configured. The model size grows quadratically when widened. WRN has been achieving better accuracy when the model is widened [5]. **TRN** (Transformer) [49] can be enlarged by increasing the number of layers, which deepens the model, and by widening the inner-layer dimensionality. Deeper [54] and wider [49] configurations of Transformer are shown to give higher accuracy. **E3D** is Eidetic 3D LSTM [51] for video prediction. *E3D* can be enlarged by increasing the number of the hidden state channels on the memory dimensions.

We experimented with models under two main use-cases of ParDNN. First, model instances that *fit into a single device memory only when very small batch size is used.* Small here is relative to the numbers used by the DL community and reported in the literature. In such a case, ParDNN provides a qualitative advantage over data parallelism (DP), which splits the input over different GPUs that hold the replicas of the model. The second use-case is *model instances that do not fit into a single GPU memory* even with small batch sizes. These are the larger variants of each model, as shown in Table 3 (e.g., Word-RNN-2). It is important to emphasize that the motivation of ParDNN is on cases when the model can not fit in memory or fits with very small batch size, which has become a major challenge for large models. Hence the experiments focus on weak scaling, not on strong scaling. If the model fits in memory with large enough batch size, then we would not suggest using graph-based methods.

5.3. Comparison with Related Work

We compare ParDNN with two graph (critical path)-based methods and three different state-of-the-art approaches used to circumvent the memory limitation when training DNNs. We compare with (a) Linear Clustering [25] and Critical Path [19] methods, (b) Mesh-TensorFlow [28] for explicit model parallelism, (c) Gradient Checkpointing [43] in combination with data parallelism for redundant recompute and (d) CUDA Unified Memory for out-of-core computing. Although there exists other graph-based solutions, we cannot directly compare with them either because we are not aware of any open source implementation [17] or the implementation is available for MXNet only [16] and cannot support all the operations used in Tensorflow. It is worth mentioning, however, that ParDNN takes no more than 2 minutes for the largest configuration we tested, in comparison to 10s of hours reported by the other graph-based methods [17]. In addition, ParDNN working on models 2.3x as large as what the these methods experimented with [17, 18].

5.3.1. Graph-based methods: Linear Clustering and CP

In [19], the authors proposed a set of heuristics to partition Tensorflow graphs among multiple devices. Among the proposed heuristics, a critical path-based heuristic, referred as CP,

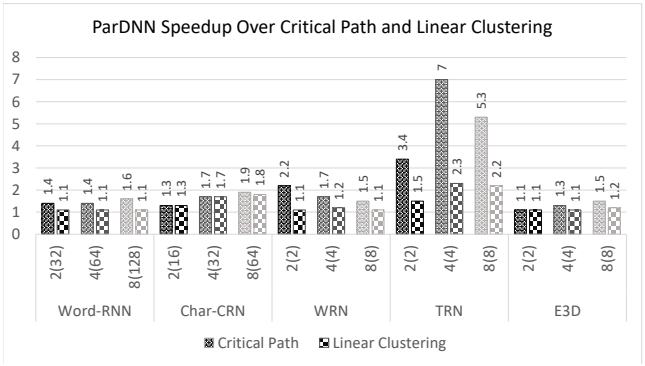


Figure 3: ParDNN speedup over Linear Clustering [25] and Critical -Path (CP) heuristic [19]. x-axis: Number of GPUs (Batch Size)

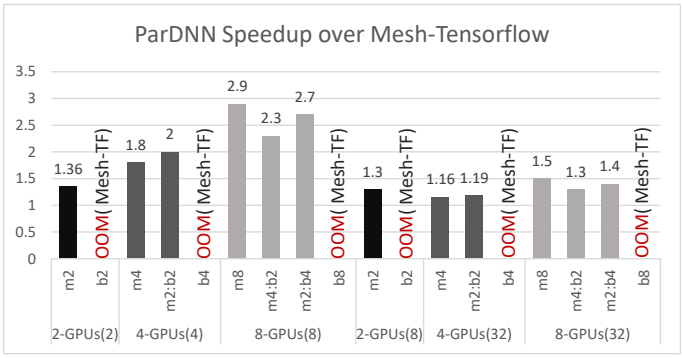


Figure 4: ParDNN speedup over Mesh Tensorflow. X-axis: number of GPUs (batch size), and the possible permutations permitted by Mesh-Tensorflow.

achieves the best results in all of their experiments. Authors performed event-based simulations to evaluate their partitioning. We applied their heuristic on our graphs, extracted the partitioning before the simulation step and fed them to Tensorflow. As Figure 3 shows, ParDNN outperforms CP using all models on all device counts.

To demonstrate that our choice of using a multi-staged approach over a high-complexity single heuristic does not harm the quality of the partitioning, we compare ParDNN with Linear Clustering (LC). To do a fair comparison, we implemented LC with GLB and Earliest Estimated Time First (EST First) [37] as a task ordering heuristic since this combination gave the best results. We post-processed the result to meet the Tensorflow placement constrains. ParDNN outperforms LC in all experiments, even though it sacrifices nodes' weighted-level recalculation after the K'th iteration, thanks to the novel mapping heuristic. Moreover, while it took ParDNN $\sim$ 2 minutes to produce the placement for the largest graph (TRN with 205K nodes), it took linear clustering $\sim$ 12 hours. CP, on the other hand, is quite fast. It partitions all the graphs within seconds. Both ParDNN and LC are far better than CP, this is because CP assigns all the nodes outside of the critical path to the least loaded devices without further grouping/clustering. In other words, it does not consider locality outside the critical path.

5.3.2. Mesh-TensorFlow

Mesh-TensorFlow [28], an extension to TensorFlow, was proposed to overcome the memory limitations of a single device. It permits specifying a general class of distributed ten-

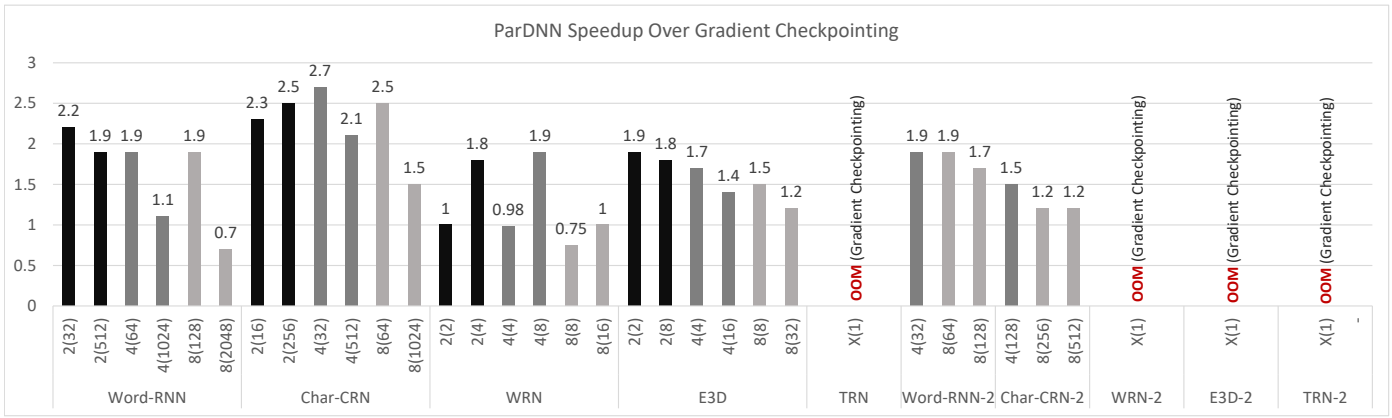


Figure 5: ParDNN speedup over gradient checkpointing combined with data parallelism to run on multiple GPUs (X-axis: Number of GPUs (Batch Size)). Two batch size scalings were used (a) doubling the batch size with the number of devices, (b) if both techniques allow larger batches, the comparison is held using the maximum commonly trainable batch size. X indicates gradient checkpointing (combined with data parallelism) could not produce a valid solution regardless of GPU count.

sor computations. We compare the performance of ParDNN with Mesh-TensorFlow using the Transformer model which the original authors used to demonstrate the scaling [28]. Figure 4 shows the speedup of ParDNN over Mesh-TensorFlow using 2, 4 and 8 GPUs. We report all permutations [55] with both regular weak scaling and the maximum trainable batch size for most of these permutations. ParDNN outperforms Mesh-TensorFlow in both cases. The performance gap is much larger with smaller batch sizes. This is because when the batch size is small, GPUs are underutilized, and Mesh-Tensorflow splitting across model dimension, batch dimension or both creates even smaller kernels exacerbating underutilization. However, when the batch sizes are large enough, the benefit of the parallelism created by Mesh-Tensorflow outweighs a possible minor underutilization leading to a relatively good performance. Moreover, unlike Mesh-TensorFlow (a) ParDNN requires no knowledge about the DNN structure by the user, while with Mesh-TensorFlow it is the responsibility of the user to rewrite the model using Mesh-TensorFlow syntax. (b) ParDNN entirely automates the partitioning, while with Mesh-TensorFlow users have to manually specify the tensor-dimensions to be split across a multi-dimensional processor mesh and finding the best assignment is an NP-hard problem. (c) Mesh-TensorFlow has a non-negligible pre-run overhead which doubles when doubling the number of GPUs reaching $\sim$ 1 hour for 8 GPU assignment.

5.3.3. *Redundant Recompute: Gradient Checkpointing*

Gradient checkpointing [56] enables DNN training with a sublinear memory cost ($O(\sqrt{N})$) when training an N layer network by recomputing the activations during backpropagation, instead of holding the forward pass results. In our comparison, we use a TensorFlow-based open-source implementation [43]. Figure 5 shows the speedup of ParDNN over gradient checkpointing when combined with data parallelism to run on multiple GPUs. For ParDNN and checkpointing, we used both regular weak scaling and the common largest possible batch sizes. ParDNN outperforms gradient checkpointing in most cases. In few cases, checkpointing is better than ParDNN; this

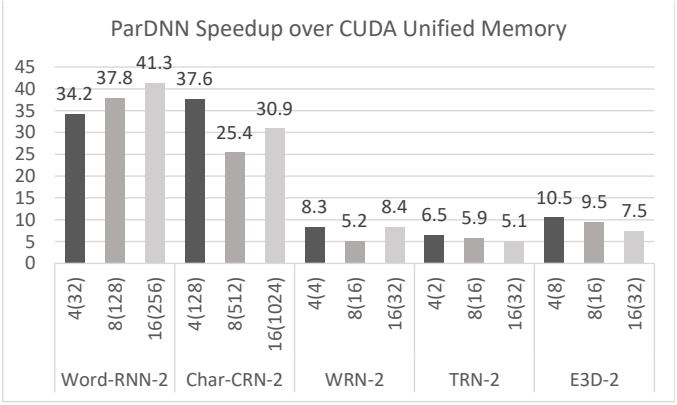
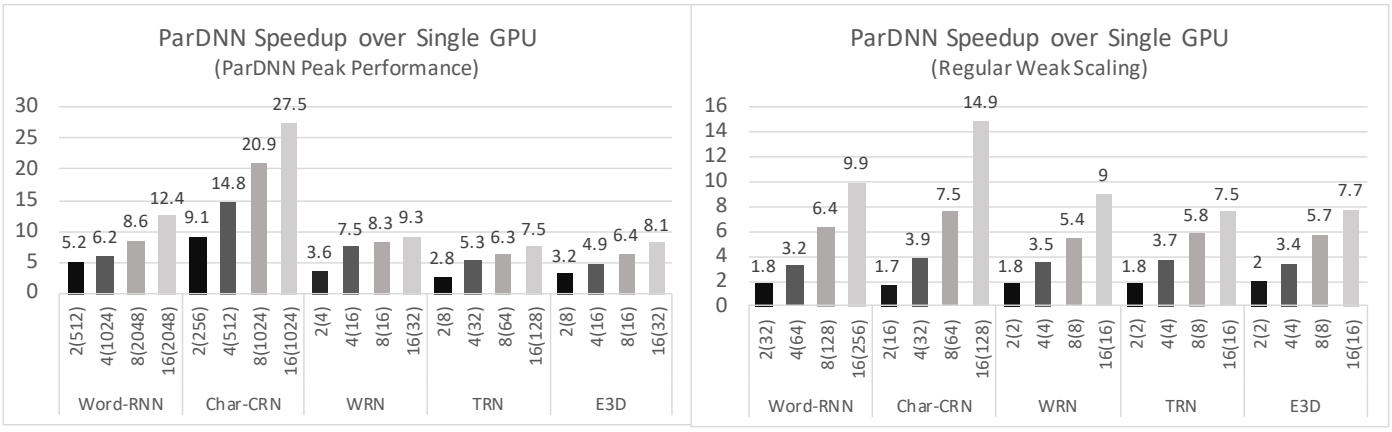


Figure 6: ParDNN speedup over CUDA Unified Memory (UM) using the large models. X-axis: Number of GPUs (Batch Size)

happens mainly when the degree of parallelism inherent in the graph is not sufficient to fully utilize all the GPUs when the model is partitioned, or when checkpointing enables pushing large enough batch-size that guarantees satisfactory utilization of these GPUs. However, more importantly, ParDNN is qualitatively superior to gradient checkpointing since it enables the training of models by using multiple GPUs where checkpointing fails to make them fit in device memory, even when using a batch size of one. For example, Figure 5(b) shows several configurations where gradient checkpointing goes out-of-memory at the batch size of one. Moreover, the overhead of gradient checkpointing can be up to 5 hours [43].

5.3.4. *Out-of-core: CUDA Unified Memory*

Figure 6 shows the speedup of ParDNN over CUDA Unified Memory (UM). UM, to the authors knowledge, is the only out-of-core solution that has an available Tensorflow implementation. ParDNN outperforms UM in all cases. Although UM allows pushing large batch sizes what enhances GPU utilization, its performance degrades in many cases when increasing the batch size due to the larger device-host communication cost and the page faulting penalty [57].



(a) (b)

Figure 7: ParDNN speedup over a single GPU using 2, 4, 8 and 16 GPUs. X-axis: Number of GPUs (Batch Size)

Table 4: Maximum batch sizes (bsz) made possible by ParDNN. Bsz on a single GPU is the maximum that could fit without triggering OOM. Table also shows the multiplier by which ParDNN could increase the bsz over ideal data parallelism (DP). For use-cases-1, DP is assumed to applied on top of a single GPU reference point. For use-cases-2, ParDNN enables ≥ 4-GPU assignment and DP is assumed to be applied on top of 4-GPU reference point. We report the values enabled by both of the memory heuristics.

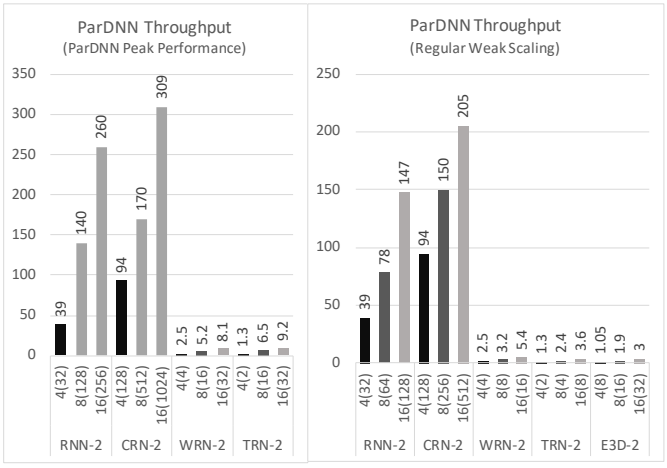
	Batch Size Scaling					Increase Over Ideal DP				
Model / #GPUs	1	2	4	8	16	1	2	4	8	16
Word-RNN	16	512	1024	2048	2048	1x	16x	16x	16x	8x
Char-CRN	8	256	512	1024	2048	1x	16x	16x	16x	16x
WRN	1	4	16	16	32	1x	2x	4x	2x	2x
TRN	1	8	32	64	128	1x	4x	8x	8x	8x
E3D	1	8	16	16	32	1x	4x	4x	2x	2x
Word-RNN-2	–	–	32	128	256	–	–	1x	2x	2x
Char-CRN-2	–	–	128	512	1024	–	–	1x	2x	2x
WRN-2	–	–	4	16	32	–	–	1x	2x	2x
TRN-2	–	–	2	16	32	–	–	1x	4x	4x
E3D-2	–	–	8	16	32	–	–	1x	1x	1x

5.4. Scaling Studies

5.4.1. Batch Size Scaling

Training with large batch sizes offers more parallelism and drastically reduces the overall training time. Authors in [58] proposed a method to scale batch sizes, which reduced the training of RESNET-50 on ImageNet to one hour. Another work harnessed very large batch sizes to reduce BERT training time from 3 days to 76 mins [59]. ParDNN enables superlinear scaling of the batch sizes while increasing the number of GPUs. Table 4 shows the batch size scaling for all of our experiments. We could increase the batch size by up to 256x for use-cases-1 and 16x for use-cases-2 going from one to 16 GPUs. This gives ParDNN a qualitative advantage even for models that fit into a single GPU since ParDNN enables training with much larger batch sizes than what can be achieved with DP.

ParDNN achieves superlinear scaling of the batch size firstly because with ParDNN, the parameters are not replicated but distributed. A large fraction of the memory consumed by the large models is to store the parameters and variables that survive through iterations. For instance, for 1.91 billion parameter *WRN*, TensorFlow allocates around 8GB for those variables. Using ParDNN these parameters are distributed, but with



(a) (b)

Figure 8: Throughput and scaling up to 16 GPUs with use-case-2 models. X-axis: Number of GPUs (Batch Size)

DP they need to be replicated. Secondly, for some operations, the memory consumption does not scale linearly with the batch size. For example, in *Word-RNN* and *Char-CRN*, the outputs of matrix multiplication operations have the largest memory consumption ratio. When doubling the batch size, the memory consumption by matrix multiplication results increases by only $\sim 25\%$. This is because the batch size might be the inner dimension for many of these multiplications, when multiplying a matrix of dimensions $a{*}batch\_size$ by another of $batch\_size{*}b$, the result has the dimensions of $a * b$ regardless of the $batch\_size$. So the memory allocated to store the output of that operation does not increase, and this effect propagates to its decedents that will take its output as their input.

5.4.2. GPU Count Scaling

Figure 7 and 8 show the speedup over a single GPU for small models and the throughput scaling of ParDNN for large models, respectively. We used two different sets of batch sizes to demonstrate scaling. The first demonstrates scaling with the batch sizes at which ParDNN achieves the peak performance,

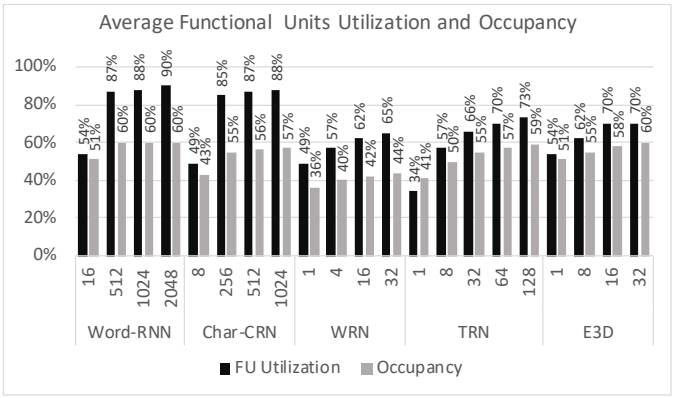


Figure 9: Average functional unit utilization and GPU occupancy with different batch sizes. X-axis: Batch Size

the vast majority of these batch sizes are the ones listed in Table 4. The second doubles the batch size with the number of GPUs (regular weak scaling).

In Figure 7 (a), ParDNN shows a substantial improvement on 2 GPUs and superlinear speedups up to 4 GPUs for all the models. The sharp performance increase happens because, in addition to the parallelism introduced by adding more GPUs, pushing larger batches while doubling the number of GPUs, improves the device utilization considerably.

Figure 9 shows the improvement in GPU functional unit utilization and GPU occupancy measured by using *nvprof* by Nvidia. We observe a similar leap where the values of these metrics increase considerably with the batch sizes used on 2, and in some cases, 4 GPUs. With the batch sizes used on 8, 16 GPUs, the utilization improves at a very small rate. Hence the scaling depends more on the inherent DoP (average degree of parallelism) in the graph and CCR (Table 5). More specifically, the models with higher DoPs scales better since introducing more GPUs exploits the parallelism. However, models with low DoP would result in small improvement in utilization since there is no parallelism inherent in the graph to be harnessed by introducing more GPUs.

Char-CRN has a large DoP, hence it continues to give superlinear speedups up to 16 GPUs. However, its scaling is not perfect due to having very high CCR, which results in having more communication links between the partitions with higher GPU counts. *Word-RNN* scales reasonably between 4 and 16 GPUs, thanks to its high DoP. However, the scaling is not perfect again due to the high CCR. *E3d* scales reasonably due to its medium CCR: the scaling is not ideal due to the relatively low DoP. Moreover, E3D experiences the lowest improvement in GPU utilization since its main operation is 3D convolution, which utilizes the GPU well enough ($>$ 50% functional unit utilization) even with a batch size of 1. Both *TRN* and *WRN* have high CCR and low DoP, hence they have the lowest improvement rate going from 4 to 16 GPUs. In Figure 7 (b), the magnitudes of the achieved speedups are less due to smaller batch sizes which means less utilization (as a magnitude) compared to their counterparts in Figure 7 (a). However, the scaling is better (going from X to 2X GPUs) up to 16 GPUs. This is because most of the used batch sizes belong to the ranges at which the utilization considerably improves with larger batches (before the plateau in Figure 9).

Table 5: Degree of Parallelism (DoP) and Communication to Computation Ratio (CCR) of the models.

Model	DoP	CCR
Word-RNN	10.45	14.5
Char-CRN	49	57
WRN	1.16	13.02
TRN	2	13.7
E3d	3.3	1.12

In Figure 8 (a), going from 4 to 8 GPUs enables much larger batches in all cases. This in turn enhances the resource utilization and results in substantial throughput improvements. *Char-CRN-2* scales linearly up to 16 GPUs due to its high DoP. The same applies for *Word-RNN-2*. *WRN-2* and TRN-2 scale modestly from 8 to 16 due to the low DoP. But the scaling is better than in Figure 7 since the batch sizes trainable with these models are located in the region at which the GPU utilization still improves with larger batch sizes (before the plateau). In Figure 8 (b) ParDNN enables linear scaling of the batch size for *E3D-2* with a performance scaling behavior similar to *E3D* for the same reasons.

5.5. *Overhead of* ParDNN

ParDNN has a negligible overhead thanks to the low complexity of each step. The longest partitioning time among all the combinations of batch sizes, GPUs and model configurations used in this work was 2 minutes in the case of partitioning *TRN-2* over 16 GPUs. The minimum time of 18 secs was taken to partition *Word-RNN* over 2 GPUs. Even though handling the memory overflow in case of memory heuristic-I takes most of the overall partitioning time, the time taken to handle memory overflow is much lower than the theoretical upper bound. This is because the complexity analysis of Step-2 of ParDNN depends on how many nodes need to be moved between clusters to address the overflow, which is much less than $|V|$ in practice. The average ratio of the nodes moved in all our experiments is 8%.

6. Related Work

We summarize related work that touches on different aspects of ParDNN: from techniques that handle, or can alleviate, memory bottlenecks either using a single device by partitioning a DNN across multiple devices, to graph partitioning and scheduling.

Systems-level approaches: Mirhoseini et al. proposed a reinforcement learning-based method to place dataflow graphs on multiple devices [18, 17]. This approach suffers from significant time and resource consumption. The proposed policy was trained for hours using 16 workers to produce placements for models having less than $100K$ operations. A more efficient approach was proposed by Wang et al. in [16]. However, it requires a description language to specify computations and cannot describe all the operations used in DL. Moreover, it partitions all operators and tensors across all workers, resulting in poor resource utilization. In [19], authors propose a set of

practical heuristics to partition Tensorflow graphs. They concluded that critical path-based approaches yield the best performance.

DL-level approaches: Explicit model parallelism, where each worker is responsible for a subset of the layers, suffers from two major limitations: requiring complex cost models on case-by-case bases and leaving the partitioning burden to the programmer [14]. Pipeline parallelism provides good resource utilization yet some implementations requires a single layer to fit in a single device [54], which may not be the case for models with 3D inputs [60]. While in others, extra memory overhead proportional to the size of the model weights is necessary to address the statistical efficiency issue, i.e. preventing model convergence [14]. In [15, 61, 12, 9] non-generic techniques were proposed to parallelize specific types of DL models, some focusing on CNNs while others relying on Transformer in their optimizations.

Out-of-core and Recomputation: these methods either augment the device by utilizing an extra memory (Out-of-core methods) or optimize the memory consumption of the model (recomputation methods). vDNN [62] is a memory manager that virtualizes GPU memory in DNN training. ooc_cuDNN [63] extends cuDNN and applies cuDNN-compatible operators even when a layer exceeds GPU memory capacity by swapping at the granularity of individual tensor dimensions. Gradient checkpointing [56] reduces the memory needed to store the intermediate outputs and gradients with the cost of doubling the forward pass computational cost [64, 56]. PoocH [65] and Capuchin [29] propose a hybrid approach that selects either recomputing or swapping for certain layers to reduce the performance overhead based on profiling data.

Graph partitioning: To deal with a directed graph, existing graph partitioning libraries convert every directed edge to undirected even though this conversion loses crucial information [66]. Due to this reason, Scotch static mapper [22, 23] and MinCut optimizer, results in 2 to 10 times slowdown when applied on graphs of DL models [18, 17]. In [67], new techniques are proposed to deal with directed graphs and [40] built on top of those techniques for a clustering based scheduler. They aim at producing *acyclic partitioning*, where if there is a cut edge from partition a to b and another from b to a, the partition is considered cyclic, and is not acceptable. Since the graphs produced by Tensorflow are full of fork-joins, applying their technique to our DNN models results in unbalanced partitions.

Static graph scheduling: Plenty of sophisticated and high-quality algorithms were proposed [68, 69, 70, 71, 72] in this area. The vast majority of these algorithms were developed in 1990's to handle small-sized graphs, and they were later evaluated using instances having up to 3000 nodes [73, 74, 72, 37, 75]. A recent evaluation on large graphs shows that they either do not scale due to their high time-complexity, or produce low-quality allocations due to their inability to capture the global structure of the graph [40]. Table 6 shows the time complexity of some of these algorithms. Note that these heuristics are only scheduling heuristics (do not include memory handling component), hence their complexities should be compared against Step 1 of ParDNN.

Table 6: Time complexity of some of the best task scheduling algorithms

Algorithm	Time Complexity
Dynamic Critical Path [69]	$O(\|V\|^3)$
Bubble Scheduling [68]	$O(K^2\|V\|\|E\|)$
Earliest Task First [71]	$O(K\|V\|^2)$
Linear Clustering [25]	$O(\|V\|(\|V\|+\|E\|))$

7. Conclusion

ParDNN presents a lightweight and automatic approach to partition computational graphs of very large DNN models. It permits the training of models that do not fit into a single device memory, and enhances the training throughput of models barely fitting into a single device memory while being non-intrusive and generic. ParDNN is applied ahead of time and, hence the partitioning is available at the beginning of a run what enables applying any type of dynamic(runtime) optimizations on top of it. Due to the limited degree of parallelism of DNN graphs and the trend towards including more accelerators in one node, we proposed to apply ParDNN within a single node. Nevertheless, it can be used as an integral part in a large-scale training where Data Parallelism is used as an inter-node technique. The experiments on five large DNNs and comparisons with related work demonstrate its high efficiency and superlinear scaling of batch size and training throughput.

Acknowledgement

Authors from Koç University are supported by the Turkish Science and Technology Research Centre Grant No: 118E801. This work was partially supported by JST-CREST under Grant Number JPMJCR19F5. The research presented in this paper has benefited from the Experimental Infrastructure for Exploration of Exascale Computing (eX3), which is financially supported by the Research Council of Norway under contract 270053. Dr. Didem Unat is supported by the Young Scientist Awards Program by the Turkish Science Academy.

References

[1] I. Goodfellow, Y. Bengio, A. Courville, Deep learning, MIT press, 2016.

[2] S. Rajbhandari, J. Rasley, O. Ruwase, Y. He, Zero: Memory optimization towards training a trillion parameter models, ArXiv abs/1910.02054 (2019).

[3] Z. Alo, T. M. Taha, C. Yakopcic, S. Westberg, V. Sagan, M. S. Nasrin, M. Hasan, B. C. V. Essen, A. A. S. Awwal, V. K. Asari, A state-of-the-art survey on deep learning theory and architectures, Electronics 8 (3) (2019) 292. doi:10.3390/electronics8030292.

[4] T. Sekiyama, T. Imamichi, H. Imai, R. Raymond, Profile-guided memory optimization for deep neural networks, arXiv preprint arXiv:1804.10001 (2018).

[5] S. Zagoruyko, N. Komodakis, Wide residual networks, arXiv preprint arXiv:1605.07146 (2016).

[6] K. He, X. Zhang, S. Ren, J. Sun, Deep residual learning for image recognition, in: Proceedings of the IEEE conference on computer vision and pattern recognition, 2016, pp. 770–778.

[7] Y. Wu, M. Schuster, Z. Chen, Q. V. Le, M. Norouzi, W. Macherey, M. Krikun, Y. Cao, Q. Gao, K. Macherey, et al., Google's neural machine translation system: Bridging the gap between human and machine translation, arXiv preprint arXiv:1609.08144 (2016).

[8] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, I. Polosukhin, Attention is all you need, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems 30, Curran Associates, Inc., 2017, pp. 5998–6008.
URL http://papers.nips.cc/paper/7181-attention-is-all-you-need.pdf

[9] M. Shoeybi, M. Patwary, R. Puri, P. LeGresley, J. Casper, B. Catanzaro, Megatron-lm: Training multi-billion parameter language models using gpu model parallelism, arXiv preprint arXiv:1909.08053 (2019).

[10] A. Gholami, A. Azad, P. Jin, K. Keutzer, A. Buluc, Integrated model, batch and domain parallelism in training neural networks, arXiv preprint arXiv:1712.04432 (2017).

[11] Z. Jia, M. Zaharia, A. Aiken, Beyond Data and Model Parallelism for Deep Neural Networks, CoRR abs/1807.05358 (2018). arXiv:1807.05358.
URL http://arxiv.org/abs/1807.05358

[12] N. Dryden, et al., Channel and Filter Parallelism for Large-Scale CNN Training, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '19, 2019, pp. 46:1–46:13.

[13] Y. Huang, et al., GPipe: Efficient Training of Giant Neural Networks using Pipeline Parallelism, CoRR abs/1811.06965 (2018). arXiv:1811.06965.

[14] D. Narayanan, A. Harlap, A. Phanishayee, V. Seshadri, N. R. Devanur, G. R. Ganger, P. B. Gibbons, M. Zaharia, Pipedream: generalized pipeline parallelism for dnn training, in: Proceedings of the 27th ACM Symposium on Operating Systems Principles, 2019, pp. 1–15.

[15] Z. Jia, S. Lin, C. R. Qi, A. Aiken, Exploring hidden dimensions in parallelizing convolutional neural networks, arXiv preprint arXiv:1802.04924 (2018).

[16] M. Wang, C.-c. Huang, J. Li, Supporting very large models using automatic dataflow graph partitioning, in: Proceedings of the Fourteenth EuroSys Conference 2019, 2019, pp. 1–17.

[17] A. Mirhoseini, A. Goldie, H. Pham, B. Steiner, Q. V. Le, J. Dean, A hierarchical model for device placement (2018).

[18] A. Mirhoseini, H. Pham, Q. V. Le, B. Steiner, R. Larsen, Y. Zhou, N. Kumar, M. Norouzi, S. Bengio, J. Dean, Device placement optimization with reinforcement learning, in: Proceedings of the 34th International Conference on Machine Learning - Volume 70, ICML'17, JMLR.org, 2017, p. 2430–2439.

[19] R. Mayer, C. Mayer, L. Laich, The tensorflow partitioning and scheduling problem: it's the critical path!, in: Proceedings of the 1st Workshop on Distributed Infrastructures for Deep Learning, 2017, pp. 1–6.

[20] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, et al., Tensorflow: Large-scale machine learning on heterogeneous distributed systems, arXiv preprint arXiv:1603.04467 (2016).

[21] T. Chen, M. Li, Y. Li, M. Lin, N. Wang, M. Wang, T. Xiao, B. Xu, C. Zhang, Z. Zhang, Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems, arXiv preprint arXiv:1512.01274 (2015).

[22] F. Pellegrini, J. Roman, Scotch: A software package for static mapping by dual recursive bipartitioning of process and architecture graphs, in: International Conference on High-Performance Computing and Networking, Springer, 1996, pp. 493–498.

[23] F. Pellegrini, Distillating knowledge about scotch, in: Dagstuhl Seminar Proceedings, Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2009.

[24] G. Karypis, V. Kumar, Multilevel graph partitioning schemes, in: ICPP (3), 1995, pp. 113–122.

[25] S. J. Kim, A general approach to multiprocessor scheduling (1988).

[26] C. McCreary, M. Cleveland, A. Khan, The problem with critical path scheduling algorithms, Master's Thesis, Department of Computer Science and Engineering Auburn University, USA (1996).

[27] A. Radulescu, A. J. Van Gemund, Glb: A low-cost scheduling algorithm for distributed-memory architectures, in: Proceedings. Fifth International Conference on High Performance Computing (Cat. No. 98EX238), IEEE, 1998, pp. 294–301.

[28] N. Shazeer, Y. Cheng, N. Parmar, D. Tran, A. Vaswani, P. Koanantakool, P. Hawkins, H. Lee, M. Hong, C. Young, R. Sepassi, B. Hechtman, Mesh-tensorflow: Deep learning for supercomputers, in: Proceedings of the 32nd International Conference on Neural Information Processing Systems, NIPS'18, Curran Associates Inc., Red Hook, NY, USA, 2018, p. 10435–10444.

[29] X. Peng, X. Shi, H. Dai, H. Jin, W. Ma, Q. Xiong, F. Yang, X. Qian, Capuchin: Tensor-based gpu memory management for deep learning, in: Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS '20, Association for Computing Machinery, New York, NY, USA, 2020, p. 891–905. doi:10.1145/3373376.3378505.
URL https://doi.org/10.1145/3373376.33785kwok1995bubble05

[30] O. Sinnen, Task scheduling for parallel systems, Vol. 60, John Wiley & Sons, 2007.

[31] M. S. Lam, M. C. Rinard, Coarse-grain parallel programming in jade, in: Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming, 1991, pp. 94–105.

[32] T. Yang, A. Gerasoulis, Pyrros: static task scheduling and code generation for message passing multiprocessors, in: ACM International Conference on Supercomputing 25th Anniversary Volume, 1992, pp. 163–172.

[33] P. Newton, J. C. Browne, The code 2.0 graphical parallel programming language, in: Proceedings of the 6th international conference on Supercomputing, 1992, pp. 167–177.

[34] C. Augonnet, S. Thibault, R. Namyst, P.-A. Wacrenier, Starpu: a unified platform for task scheduling on heterogeneous multicore architectures, Concurrency and Computation: Practice and Experience 23 (2) (2011) 187–198.

[35] J. Bergstra, F. Bastien, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, D. Warde-Farley, I. Goodfellow, A. Bergeron, et al., Theano: Deep learning on gpus with python, in: NIPS 2011, BigLearning Workshop, Granada, Spain, Vol. 3, Citeseer, 2011, pp. 1–48.

[36] C.-E. Bichot, P. Siarry, Graph partitioning, Wiley Online Library, 2011.

[37] H. Wang, O. Sinnen, List-scheduling versus cluster-scheduling, IEEE Transactions on Parallel and Distributed Systems 29 (8) (2018) 1736–1749.

[38] T. Yang, Scheduling and code generation for parallel architectures, Ph.D. thesis, Citeseer (1993).

[39] V. Sarkar, Partitioning and scheduling parallel programs for execution on multiprocessors. (1988).

[40] M. Y. Özkaya, A. Benoit, B. Uçar, J. Herrmann, Ü. V. Çatalyürek, A scalable clustering-based task scheduler for homogeneous processors using dag partitioning, in: 2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS), IEEE, 2019, pp. 155–165.

[41] P. M. Fenwick, A new data structure for cumulative frequency tables, Software: Practice and experience 24 (3) (1994) 327–336.

[42] J. Csirik, Heuristics for the 0-1 min-knapsack problem, Acta Cybernetica 10 (1-2) (1991) 15–20.

[43] Y. Bulatov, gradient-checkpointing, https://github.com/cybertronai/gradient-checkpointing (Jan. 2018).

[44] J. J. Sung Kim, Multi-layer Recurrent Neural Networks (LSTM, RNN) for word-level language models in Python using TensorFlow (Dec. 2017).
URL https://github.com/hunkim/word-rnn-tensorflow

[45] A. Karpathy, tinyshakespeare (May 2015).
URL https://github.com/karpathy/char-rnn/tree/master/data/tinyshakespeare

[46] Y. Kim, Y. Jernite, D. Sontag, A. M. Rush, Character-aware neural language models, in: Thirtieth AAAI Conference on Artificial Intelligence, 2016.

[47] M. Marcus, G. Kim, M. A. Marcinkiewicz, R. MacIntyre, A. Bies, M. Ferguson, K. Katz, B. Schasberger, The penn treebank: annotating predicate argument structure, in: Proceedings of the workshop on Human Language Technology, Association for Computational Linguistics, 1994, pp. 114–119.

[48] A. Krizhevsky, G. Hinton, et al., Learning multiple layers of features from tiny images (2009).

[49] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, I. Polosukhin, Attention is all you need, in: Advances in neural information processing systems, 2017, pp. 5998–6008.

[50] M. Cettolo, N. Jan, S. Sebastian, L. Bentivogli, R. Cattoni, M. Federico, The iwslt 2016 evaluation campaign, in: International Workshop on Spoken Language Translation, 2016.

[51] Y. Wang, L. Jiang, M.-H. Yang, L.-J. Li, M. Long, L. Fei-Fei, Eidetic 3d lstm: A model for video prediction and beyond (2018).
[52] N. Srivastava, E. Mansimov, R. Salakhudinov, Unsupervised learning of video representations using lstms, in: International conference on machine learning, 2015, pp. 843–852.
[53] I. Sutskever, J. Martens, G. E. Hinton, Generating text with recurrent neural networks, in: Proceedings of the 28th international conference on machine learning (ICML-11), 2011, pp. 1017–1024.
[54] Y. Huang, Y. Cheng, A. Bapna, O. Firat, D. Chen, M. Chen, H. Lee, J. Ngiam, Q. V. Le, Y. Wu, et al., Gpipe: Efficient training of giant neural networks using pipeline parallelism, in: Advances in Neural Information Processing Systems, 2019, pp. 103–112.
[55] A. Vaswani, S. Bengio, E. Brevdo, F. Chollet, A. N. Gomez, S. Gouws, L. Jones, L. Kaiser, N. Kalchbrenner, N. Parmar, R. Sepassi, N. Shazeer, J. Uszkoreit, Tensor2tensor for neural machine translation, CoRR abs/1803.07416 (2018).
URL http://arxiv.org/abs/1803.07416
[56] T. Chen, B. Xu, C. Zhang, C. Guestrin, Training deep nets with sublinear memory cost, ArXiv abs/1604.06174 (2016).
[57] A. A. Awan, C.-H. Chu, H. Subramoni, X. Lu, D. K. Panda, Oc-dnn: Exploiting advanced unified memory capabilities in cuda 9 and volta gpus for out-of-core dnn training, in: 2018 IEEE 25th International Conference on High Performance Computing (HiPC), IEEE, 2018, pp. 143–152.
[58] P. Goyal, P. Dollár, R. Girshick, P. Noordhuis, L. Wesolowski, A. Kyrola, A. Tulloch, Y. Jia, K. He, Accurate, large minibatch sgd: Training imagenet in 1 hour, arXiv preprint arXiv:1706.02677 (2017).
[59] Y. You, J. Li, S. Reddi, J. Hseu, S. Kumar, S. Bhojanapalli, X. Song, J. Demmel, K. Keutzer, C.-J. Hsieh, Large batch optimization for deep learning: Training bert in 76 minutes, in: International Conference on Learning Representations, 2019.
[60] A. Mathuriya, et al., Cosmoflow: Using deep learning to learn the universe at scale, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage, and Analysis, SC '18, IEEE Press, Piscataway, NJ, USA, 2018, pp. 65:1–65:11.
[61] A. Krizhevsky, One weird trick for parallelizing convolutional neural networks (2014), arXiv preprint arXiv:1404.5997 (2014).
[62] M. Rhu, N. Gimelshein, J. Clemons, A. Zulfiqar, S. W. Keckler, vdnn: Virtualized deep neural networks for scalable, memory-efficient neural network design, in: The 49th Annual IEEE/ACM International Symposium on Microarchitecture, IEEE Press, 2016, p. 18.
[63] Y. Ito, R. Matsumiya, T. Endo, ooc_cudnn: Accommodating convolutional neural networks over GPU memory capacity, in: 2017 IEEE International Conference on Big Data, BigData 2017, Boston, MA, USA, December 11-14, 2017, 2017, pp. 183–192. doi:10.1109/BigData.2017.8257926.
URL https://doi.org/10.1109/BigData.2017.8257926
[64] P. Jain, A. Jain, A. Nrusimha, A. Gholami, P. Abbeel, J. Gonzalez, K. Keutzer, I. Stoica, Breaking the memory wall with optimal tensor rematerialization, in: Proceedings of Machine Learning and Systems 2020, 2020, pp. 497–511.
[65] Y. Ito, H. Imai, T. D. Le, Y. Negishi, K. Kawachiya, R. Matsumiya, T. Endo, Profiling based out-of-core hybrid method for large neural networks: poster, ArXiv abs/1907.05013 (2019).
[66] D. A. Bader, H. Meyerhenke, P. Sanders, D. Wagner, Graph partitioning and graph clustering, Vol. 588, American Mathematical Society Providence, RI, 2013.
[67] J. Herrmann, J. Kho, B. Uçar, K. Kaya, Ü. V. Çatalyürek, Acyclic partitioning of large directed acyclic graphs, in: 2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID), IEEE, 2017, pp. 371–380.
[68] Y.-K. Kwok, I. Ahmad, Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures, in: Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing, IEEE, 1995, pp. 36–43.
[69] Y.-K. Kwok, I. Ahmad, Dynamic critical-path scheduling: An effective technique for allocating task graphs to multiprocessors, IEEE transactions on parallel and distributed systems 7 (5) (1996) 506–521.
[70] T. Yang, A. Gerasoulis, Dsc: Scheduling parallel tasks on an unbounded number of processors, IEEE Transactions on Parallel and Distributed Systems 5 (9) (1994) 951–967.
[71] J.-J. Hwang, Y.-C. Chow, F. D. Anger, C.-Y. Lee, Scheduling precedence graphs in systems with interprocessor communication times, SIAM Journal on Computing 18 (2) (1989) 244–257.
[72] K. He, X. Meng, Z. Pan, L. Yuan, P. Zhou, A novel task-duplication based clustering algorithm for heterogeneous computing environments, IEEE Transactions on Parallel and Distributed Systems 30 (1) (2018) 2–14.
[73] J. Wang, X. Lv, X. Chen, Comparative analysis of list scheduling algorithms on homogeneous multi-processors, in: 2016 8th IEEE International Conference on Communication Software and Networks (ICCSN), IEEE, 2016, pp. 708–713.
[74] J.-C. Liou, M. A. Palis, A comparison of general approaches to multiprocessor scheduling, in: Proceedings 11th International Parallel Processing Symposium, IEEE, 1997, pp. 152–156.
[75] A. Gerasoulis, T. Yang, A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors, journal of parallel and distributed computing 16 (4) (1992) 276–291.