Training Memory-Constrained Deep Learning Models Using Automatic Dataflow-Graph Partitioning

by

Fareed Mohammad Fareed Qararyah

A Dissertation Submitted to the
Graduate School of Sciences and Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Master of Science

in

Computer Engineering

KOÇ ÜNİVERSİTESİ

September 15, 2020

Training Memory-Constrained Deep Learning Models Using Automatic Dataflow-Graph Partitioning

Koç University

Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a master's thesis by

Fareed Mohammad Fareed Qararyah

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Assist. Prof. Didem Unat (Advisor)

Prof. Deniz Yuret

Assist. Prof. Kamer Kaya

Date:

[To my family]

ABSTRACT

Training Memory-Constrained Deep Learning Models Using Automatic Dataflow-Graph Partitioning
Fareed Mohammad Fareed Qararyah
Master of Science in Computer Engineering
September 15, 2020

Scaling Deep Neural Networks (DNNs) has been crucial for enhancing the accuracy in many real-world applications. Hence, many state-of-the-art DNNs are becoming deeper and/or wider; having larger parameter sets and/or more layers. However, this comes at a cost, since training these models requires huge amounts of memory. The memory required to store the parameters and intermediate results exceeds what is offered by the processing elements usually used in training, i.e. Graphics Processing Units (GPUs). This necessitates seeking algorithms and techniques to enable efficient training of these memory-demanding models under the capacity constraints imposed by the underlying processing elements.

This thesis proposes to parallelize DNN (ParDNN), an automatic, generic, and non-intrusive partitioning strategy for deep learning models that are represented as dataflow graphs. Our partitioning relies on the dataflow graph representation of the DNN model because graph representations have been adopted by the most popular general-purpose deep learning frameworks. The core algorithm in our strategy partitions DNN's underlying dataflow graph among a set of processing elements so that their memory constraints are met and the training time is minimized. The proposed algorithm is completely independent of the deep learning aspects of the model to be partitioned, so it can be used with any type of emerging models. Unlike most of the distributed training frameworks, ParDNN is completely automatic and it requires the user no knowledge about the model structure. Moreover, it requires no modification neither at the model nor at the systems level implementation of the operation kernels.

The dataflow graph of a widely-used DNN contains few thousands up to hundreds of thousands of operations, and this number is subject to rapid increase. As a result, finding an efficient placement of the graph operations on a set of processing elements requires an algorithm that strikes the right balance between obtaining a good quality partitioning and having low overhead. In this thesis, we explain the shortcomings of the existing heuristics to attain this balance, and how our strategy is designed to overcome these shortcomings. It incorporates concepts from graph partitioning low-overhead algorithms and high-quality static scheduling techniques.

In this thesis, we demonstrate the design and experimental results of our strategy. Using ParDNN, we experimented with a set of models representing the major application domains of deep learning. We use TensorFlow deep learning framework in our demonstration. Our experiments have shown that the partitioning obtained by ParDNN permitted efficient training of DNNs having hundreds of thousands of operations and billions of parameters on multiple GPU devices. In these experiments, ParDNN either outperforms or provides a qualitative advantage over state-of-the-art alternatives. Moreover, ParDNN has a negligible overhead, its running time ranges between seconds to few minutes compared to the training that may last for weeks.

ÖZETÇE

Yüksek Lisans Tez Başlığı
Fareed Mohammad Fareed Qararyah
Bilgisayar Mühendisliği, Yüksek Lisans
15 Eylül 2020

Derin Sinir Ağlarını (DNN'ler) ölçeklemek, birçok gerçek dünya uygulamasında doğruluklarını artırmak için çok önemli olmuştur. Bu nedenle, son teknoloji ürünü DNN'lerin çoğu derinleşiyor ve / veya genişliyor; daha büyük parametre setlerine ve / veya daha fazla katmana sahip oluyorlar. Ancak, bu modellerin eğitimi büyük miktarda bellek gerektirdiğinden bunun bir bedeli vardır. Parametreleri ve ara sonuçları depolamak için gereken bellek, genellikle eğitmekte kullanılan işleme öğeleri, yani Grafik İşleme Birimleri (GPU'lar) tarafından sunulan belleği aşmaktadır. Bu, bellek gerektiren bu modellerin, kullanılan işlem öğelerinin getirdiği kapasite kısıtlamaları altında verimli bir şekilde eğitilmesini sağlamak için algoritmalar ve teknikler aramayı gerektirir.

Bu tez, veri akışı grafikleri olarak temsil edilen derin öğrenme modelleri için otomatik, genel ve müdahaleci olmayan bir bölümleme stratejisi olan ParDNN'ni önermektedir. Veri akışı grafiği gösterimine, evrenselliği nedeniyle ve TensorFlow, PyTorch ve MXNet gibi en popüler genel amaçlı derin öğrenme çerçeveleri tarafından benimsenmesinden dolayı bölümleme tekniğimizi veri akış grafiklerine dayandırdır. Stratejimizdeki temel algoritma, DNN'nin temelindeki veri akışı grafiğini bir dizi işleme öğesi arasında böler, böylece bellek kısıtlamaları karşılanır ve eğitim süresi en aza indirilir. Önerilen algoritma, bölümlenecek modelin derin öğrenme yönlerinden tamamen bağımsızdır, bu nedenle ortaya çıkan her tür modelle kullanılabilir. Dağıtılmış eğitim çerçevelerinin çoğunun aksine, ParDNN tamamen otomatiktir, kullanıcının model yapısı hakkında bilgi sahibi olmasını gerektirmez. Dahası, ne modelde ne de operasyon çekirdeklerinin sistem düzeyinde uygulanmasında herhangi bir değişiklik gerektirmez.

Popüler bir DNN'nin veri akışı grafiği, birkaç bin ila yüz binlerce işlem içerir ve bu sayı hızlı bir artışa tabidir. Sonuç olarak, grafik işlemlerinin bir dizi işleme elemanı üzerinde verimli bir şekilde yerleştirilmesi, çok iyi kalitede bir bölümleme elde

etmek ve düşük ek yüke sahip olmak arasında doğru dengeyi sağlayan bir algoritma gerektirir. Bu tezde önerilen strateji bu akılda tutularak tasarlanmıştır. Grafik bölümleme düşük ek yük algoritmalarından ve yüksek kaliteli statik programlama tekniklerinden kavramları içerir.

Bu tezde, stratejimizin tasarımını ve deneysel sonuçlarını ortaya koyuyoruz. ParDNN kullanarak, derin öğrenmenin başlıca uygulama alanlarını temsil eden bir dizi model denedik. Deneylerimizde Tensorflow derin öğrenme çerçevesini kullanıyoruz. Deneylerimiz, ParDNN tarafından elde edilen bölümlemenin, birden çok cihaz üzerinde yüz binlerce işlem ve milyarlarca parametreye sahip DNN'lerin verimli eğitimine izin verdiğini göstermiştir. ParDNN son teknoloji alternatiflere göre daha iyi performans gösterir veya niteliksel bir avantaj sağlar. Dahası, ParDNN 'nun ihmal edilebilir bir ek yükü vardır, çalışma süresi haftalarca sürebilen eğitime kıyasla saniyeler ile birkaç dakika arasında değişir.

ACKNOWLEDGMENTS

I would like to thank my advisor Assist. Prof. Didem Unat for the guidance and support of my MSc study and research, for her patience, motivation, and enthusiasm. I would like to acknowledge TUBITAK for funding my research. I would like to thank my family, friends, and whoever supported me during my study.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABBREVIATIONS

API	Application Program Interface
CCR	Communication to Computation Ratio
CP	Critical Path
CPU	Central Processing Unit
CUDA	Compute Unified Device Architecture
DL	Deep Learning
DNN	Deep Neural Network
DOP	Degree of Parallelism
DP	Data Parallelism
FIFO	First In First Out
GLB	Guided Load Balancing
GPU	Graphics Processing Unit
OOM	Out of Memory
RR	Round Robin
UM	Unified Memory

Chapter 1

INTRODUCTION

1.1 Motivation

Recent years have witnessed an explosion in the usage of deep learning to tackle a wide range of problems in science and engineering domains. This widespread can be attributed to the availability of more computing power and the abundance of data that can be used to train the deep learning models (DNNs) [Zhang et al., 2018]. It has been realized that Larger models produce results with higher accuracy on more complex tasks [Goodfellow et al., 2016]. An example in the computer vision field is ResNet family [He et al., 2016; Zagoruyko and Komodakis, 2016] where stacking more residual units or widening the model improves the accuracy. The same trend shows up in natural language processing with the Transformers [Vaswani et al., 2017], where consistent gains were yielded on language understanding tasks when scaling the model [Huang et al., 2018; Lepikhin et al., 2020]. This explains why, since the introduction of hidden units, DNNs have doubled in size roughly every 2.4 years. This growth is expected to continue in the coming years [Rajbhandari et al., 2019; Zahangir Alom, 2019].

Large models, with massive parameter sets or a large number of layers or both, require huge amounts of memory to train [Huang et al., 2018; Shoeybi et al., 2019; Zagoruyko and Komodakis, 2016; Sekiyama et al., 2018]. This memory is necessary to store their parameters and the intermediate results. In extreme cases, memory consumption can reach multiple Terabytes [Lepikhin et al., 2020]. However, the accelerators which are usually used to train these models, and which have participated in the renascence of deep learning, have a limited amount of memory. For example, NVIDIA Tesla V100 [NVIDIA, 2017], the most advanced GPU released by NVIDIA has only 32 GB of memory.

A plethora of techniques have been proposed to enable the training of memory-constrained DNNs. Some techniques focus on alleviating the memory consumption by optimizing the allocation strategy or reducing the memory consumption of the model[Meng et al., 2017; Sekiyama et al., 2018; Chen et al., 2016]. This approach may have a considerable running time overhead[Meng et al., 2017; Chen et al., 2016], or may not be sufficient to push the memory requirements under the budget [Sekiyama et al., 2018]. Another approach is to partition the model to run on multiple devices. One option here is to work at the DNN level, where the model is partitioned across multiple devices through model, pipeline, channel parallelism, or combination of them [Gholami et al., 2017; Jia et al., 2018b; Dryden et al., 2019; Huang et al., 2018; Narayanan et al., 2019; Shoeybi et al., 2019; Jia et al., 2018a]. Even though the proposed methods are successful to some extent, they suffer from either: (a) being non-generic as they target a specific class of DNNs, while [Shoeybi et al., 2019; Lepikhin et al., 2020] are targeting the transformer model, [Dryden et al., 2019; Jia et al., 2018a] are dealing with convolutional DNNs, (b) introduce non-negligible memory overhead [Narayanan et al., 2019] to maintain the statistical efficiency, or (c) can incur a high implementation cost and necessitate detailed profiling and understanding of the model to construct an accurate cost model to drive the partitioning ahead of time. Another option, which we adopt in this thesis, is to work at the systems level by partitioning the computational graph that represents a DNN and distributing it over multiple devices. However, the existing work in this direction has severe limitations. The method proposed in [Wang et al., 2019] 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 [Mirhoseini et al., 2018, 2017].

1.2 ParDNN ***Overview***

In this thesis, we propose ***Partitioning DNNs (*ParDNN*)***, a generic, deterministic, and non-intrusive partitioning strategy that avoids the drawbacks of the related work. ParDNN directly works on the computational graph, a system-level representation of neural networks that is adopted by the most popular general-purpose DL frameworks such as TensorFlow [Abadi et al., 2016], MXNet [Chen et al., 2015], and PyTorch [Paszke et al., 2019]. Operating on the computational graph level has three main advantages. First, it provides a fine-grained view of the model, which gives more parallelization options and allows better load balancing and resource utilization. Second, it isolates our strategy from the details of the learning process, which provides more generality and guarantees unaffected statistical efficiency [Narayanan et al., 2019] of the model. Third, working at the level of the graph enables us to leverage decades of work by the HPC community on graph partitioning and static scheduling. Unlike prior work [Wang et al., 2019], ParDNN does not require any changes to the implementation of the operation-kernels in the underlying graph.

ParDNN's strategy is composed of two main steps. In the first step, 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 to the partitions in a way that balances the computational load and reduces the communication cost. In the second step, we check whether the memory constraints are met in each partition. If they are not, we move to a finer granularity at which 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.

To partition graphs having hundreds of thousands of nodes in a reasonable time, ParDNN is designed to use a set of simple and efficient heuristics instead of a single sophisticated yet time-consuming algorithm. The outline of our approach is inspired by the principle of the multilevel approach used in graph partitioning [Karypis and Kumar, 1995b]. The design and algorithmic details of ParDNN include a mix of variants of static scheduling heuristics [Kim, 1988] that are mutated to reduce the time complexity, and novel techniques to address some shortcomings in the existing ones [McCreary et al., 1996; Radulescu and Van Gemund, 1998]. Our contributions are as follows:

- We propose and implement a novel computational graph partitioning technique that enables training models with large memory consumption on a set of devices with limited memory capacities. 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 or usage of specific implementations of the operation kernels.

- We conduct extensive experiments with large DNNs from the areas of computer vision, video prediction, language modeling, and translation to demonstrate ParDNN's efficiency. For memory demanding models that do not fit into single GPU memory, ParDNN is able to train 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 attaining superlinear scaling of the batch size, and in many cases, the training throughput. Moreover, ParDNN's overhead is negligible compared with the training time. On average, it takes less than 5 minutes to find a partition of a graph having hundreds of thousands of nodes, representing DNNs with billions of parameters, while training these models takes days or weeks.

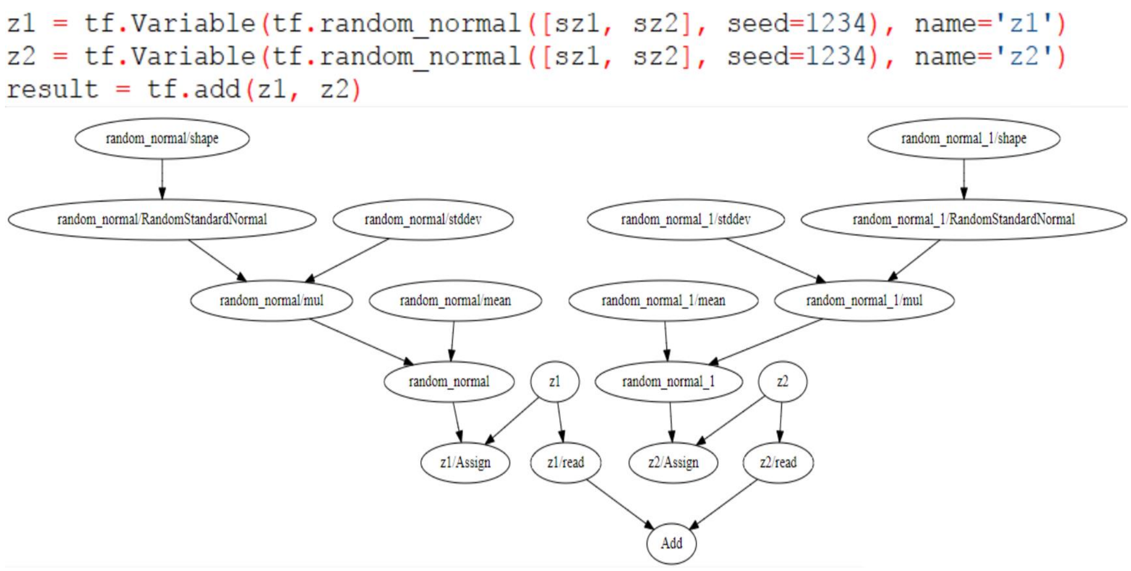
Chapter 2

BACKGROUND

In this chapter, we introduce Tensorflow dataflow graphs. Then we briefly present graph partitioning and static scheduling due to two main reasons. First, ParDNN outline is partly inspired by the concept of multilevel graph partitioning, and its design incorporates a mutated version of a static scheduling algorithm. Secondly, we explain why the direct application of these techniques either does not solve our problem or may solve it but it is impractical due to its high time complexity.

2.1 Tensor ow Data ow Graph

Modeling a computation as a directed graph has been adopted in scheduling theory [Sinnen, 2007], in parallel programming and run-time environments [Lam and Rinard, 1991; Yang and Gerasoulis, 1992; Newton and Browne, 1992; Augonnet et al., 2011], and recently in deep learning frameworks [Abadi et al., 2016; Chen et al., 2015; Bergstra et al., 2011; Paszke et al., 2019]. TensorFlow uses a stateful dataflow graph to represent a computation. It extends the classical dataflow graph model [Davis and Keller, 1982] to enable 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 dataflow between the nodes, while special edges, e.g. control dependencies, are used to enforce happens-before relationships with no data flows along them [Abadi et al., 2016]. Figure 2.1 shows a simple Tensorflow code snippet that adds two randomly generated tensors and its corresponding dataflow graph representation. We show this very simple graph since a detailed visualization of



```
z1 = tf.Variable(tf.random_normal([sz1, sz2], seed=1234), name='z1')
z2 = tf.Variable(tf.random_normal([sz1, sz2], seed=1234), name='z2')
result = tf.add(z1, z2)
```

Figure 2.1: Simple code snippet for adding two randomly generated tensors and the corresponding Tensorflow dataflow graph representation

even a simple artificial neural network does not fit into the page scale. Moreover, the shown graph is sufficient to clarify the mapping between the code and the graph representation.

2.2 Graph partitioning

2.2.1 Overview

Graph partitioning is defined as splitting the graph $G(V, E)$ into K disjoint subsets $\{P_1, \dots, P_K\}$ such that $\cup_{k=1}^{K} P_k = V$, $\forall i \in \{1, \dots, K\} : P_i \neq \emptyset$ and $\forall (i, j) \in \{1, \dots, K\}^2, i \neq j, P_i \cap P_j = \emptyset$ [Bichot and Siarry, 2011]. The constrained version of the graph partitioning aims at partitioning the graph 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 [Karypis and Kumar, 1995b]. An extension of general graph partitioning which aims to assign a set of communicating tasks to processors is called *static mapping* [Bichot and Siarry, 2011]. 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.

2.2.2 The Multilevel Method In Brief

The multilevel method [Karypis and Kumar, 1995a] is the most popular method used to solve the constrained graph partitioning problem [Bichot and Siarry, 2011]. This method gains its popularity due to the ability to partition large graphs rapidly. The main principle of this method is to group vertices together in order to deal with groups of vertices, rather than independent vertices. The method has three main levels:

- Coarsening: the goal of this level is to obtain a graph similar to the original, but whose number of vertices is lower. This will enable applying a sophisticated heuristics within a reasonable time since the problem instance has been reduced. The coarsening is an iterative process that stops when reaching a small enough graph, or when the sizes of the graphs in the last iterations are too close.

- Partitioning: at this level, the resulting graph from the coarsening phase, is partitioned using a partitioning heuristic.

- Uncoarsening and refinement: since the partitioning heuristic is applied on the coarsened graph, which gives a global view of the original one, the result may be far from being locally optimal. Thus, at this level, the graph is iteratively uncoarsened and refined to get a locally optimal version.

2.2.3 Why Graph Partitioning Is Not Suitable For Data ow Graphs

Graph partitioning is defined for undirected graphs and the existing graph partitioning tools are designed to handle undirected graphs [Pellegrini and Roman, 1996; Karypis, 1998]. To deal with a directed graph, every directed edge is converted to undirected even though this conversion loses crucial information [Bader et al., 2013]. As per the static mapping, which is an extension of general graph partitioning, it assumes that all the processes simultaneously coexist during the entire execution. In other words, it ignores the logical or the temporal dependencies . Due to these reasons, state-of-the-art graph partitioning and static mapping tools, such as Scotch static mapper [Pellegrini and Roman, 1996; Pellegrini, 2009] and MinCut optimizer result in 2 to 10 times slowdown when applied on dataflow graphs representing deep learning models [Mirhoseini et al., 2017, 2018].

Although graph partitioning tools yielded poor results [Mirhoseini et al., 2017], they continue to be applied to partition DNN graphs [Mirhoseini et al., 2018; Tarnawski et al., 2020]. This maybe due to the fact that they aim to balance the computational loads and minimize the communication, which are the right objectives. However, being directed or undirected completely alters the way in which these objectives are manifested in the graph. As a result, the heuristics designed to optimize for these objectives in one case may fail in the other.

Figure 2.2 shows how achieving an objective in an undirected graph might be completely irrelevant when dealing with a directed one. In part (a) of the figure, when the graph is undirected, the load balancing objective is perfectly met. However, with the existence of dependencies/directions, there is no load balancing at any moment in time. Instead, one processing element is active at a time. In part b, when the graph is undirected the second partitioning is better than the first, better here refers to the communication minimization point of view, since the second has two units of communication while the first has four. But, when the graph is directed, this is not valid anymore. What is important when having the temporal dependencies is the time purely consumed in the communication rather than the total amount of communication. While in the third graph there are two units of communication, both ow which are pure idleness, in the fourth there is only one unit of pure communication out of the four. These two cases show why the partitions suggested by the graph partitioning tools may not be of a high quality when applied on directed graphs.

In [Herrmann et al., 2017], new techniques are proposed to deal with directed graphs and [Özkaya et al., 2019] built on top of those techniques to develop clustering-based scheduling. Their goal is to produce ***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

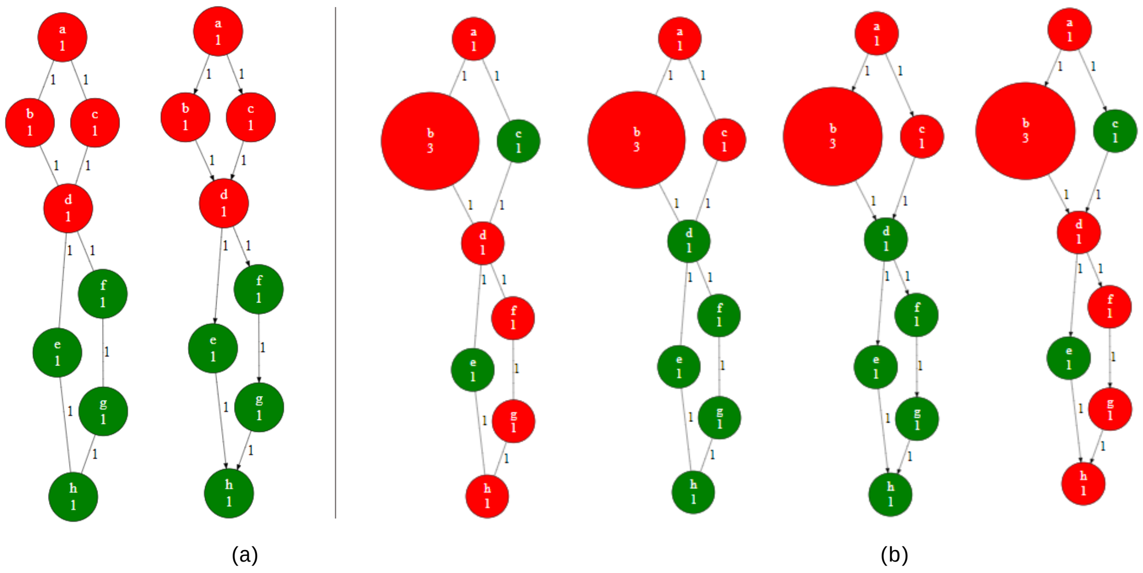


Figure 2.2: (a) Load balancing objective: Graph partitioning Algorithms ideal result (different colors represent partitions or processing elements). When the graph is undirected, loads are balanced and communication is minimized. When the graph is directed, no load balancing is achieved. (b) Communication minimization: when the graph is undirected, the second partitioning (from the left) is better than the rst since it has two units of communication as opposed to four in the rst. When the graph is directed, the notion of communication hiding is present. This means that the communication taking place while there is a work being done is not considered as it does not impose any latency. Thus the fourth partitioning is better than the third since the communications from a to c, from c to d and from d to e are hidden by the execution of the tasks b, f and g respectively.

are full of fork-joins, applying their technique to our DNN models results in highly unbalanced partitions.

2.3 Task Graph Scheduling

Finding a spatial and temporal assignment of the set of nodes in a task graph $G = (V, E)$ to a set of processors so that the end to end execution time is minimized while respecting the precedence constraints expressed by all $e \in E$ is referred to as task scheduling problem [Sinnen, 2007]. Formally, A schedule S of a task graph $G = (V, E)$ on a finite set of processing elements PE is the function pair $(ts, proc)$, where $ts : V \to Q_0^+$ is the start time function of the nodes of G; and $proc : V \to PE$ is the processor allocation function of the nodes of G to the K processing elements. The scheduling problem is to determine a feasible schedule S of minimal length for G on PE [Sinnen, 2007]. Where the schedule length, commonly known as *makespan*, is the completion time (C_t) of the last node (task) in G assuming that the execution of the graph starts at time 0. The goal is to minimize C_{tmax}, where $C_{tmax} = max_{n \in V} C_t(n)$. Task scheduling can be static or dynamic. In dynamic scheduling, the decision of which processor executes a task and when is decided by the runtime. In contrast, in static scheduling the processor allocation, and the order of tasks execution are determined at compile time. Both finding an optimal schedule and static mapping are *NP-hard* problems [Bichot and Siarry, 2011; Sinnen, 2007].

The literature is rich in sophisticated and high-quality heuristics for static task scheduling [Kwok and Ahmad, 1995, 1996; Yang and Gerasoulis, 1994; Hwang et al., 1989; He et al., 2018]. The vast majority of these algorithms were developed in the 1990s to handle small-sized graphs, and they were later evaluated using instances having tens or hundreds of nodes; less frequently exceeding 1000, and at maximum 3000 nodes [Wang et al., 2016; Liou and Palis, 1997; He et al., 2018; Wang and Sinnen, 2018; Gerasoulis and Yang, 1992]. A recent evaluation of these heuristics 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 [Özkaya et al., 2019]. In chapter5 we elaborate on the consequences of not capturing the global structure of the graph, and chapter6 we report empirical results demonstrating these heuristics high time-complexity.

Chapter 3

RELATED WORK

This chapter discusses the approaches that touche on one or more aspect of ParDNN. These mainly include methods used to enable training of memory-constrained models, as well as the well known distributed DNN training approaches even if they do not guarantee meeting the processing elements memory constraints. These approaches fall under two major categories, (a) single-device(accelerator) based, where memory optimization or augmentation techniques are applied to enable training large models using a single accelerator. (b) distributed training based, where the model is partitioned among a set of accelerators. It is important to note that we use distributed training here to refer to the case where more than one accelerator are used even if all of them are located in a single machine.

3.1 Single-device-based training of memory-constrained models

The work in this direction either reduces the memory consumption of the model so that it fits in the device memory by applying memory optimization techniques. Or augments the total memory available for the device by utilizing the host memory.

3.1.1 Memory optimization techniques

Gradient checkpointing [Chen et al., 2016] is a well-known technique to reduce the memory consumed in the training. It is a generic approach that works with both convolutional and recurrent neural networks. It enables training with a sublinear memory cost ($O(\sqrt{N})$) for an N layer network by dropping some of the intermediate results, and recovering them using an extra forward computation when needed. Checkpointing is based on trading computation for memory, since there is a need for extra computations. This incurs a non-negligible overhead of around 30% additional runtime [Chen et al., 2016].

In [Sekiyama et al., 2018], authors formulate the memory allocation problem as a special case of the two dimensional rectangle packing problems or Dynamic Storage Allocation (DSA). They propose a heuristic to optimize memory allocation. Their experiments with Chainer [Tokui et al., 2015], show that their strategy reduces the memory consumption by up to 49:5%. However, in most of the cases, the reduction was much less. More importantly, for very large models, such reduction is not sufficient. As a result, this method does not completely solve the problem.

3.1.2 Utilizing host memory

These approaches are based on offloading the device memory which is not used immediately to a slower storage, which is usually the host memory, and prefetching it when needed. They are referred to as Out-of-core techniques. The down side of this type of solutions is that it incurs an extra overhead to move the data back and forth between the device memory and the utilized extra memory. [Jain et al., 2020; Chen et al., 2016; Ito et al., 2019; Peng et al., 2020] provide different flavors and optimizations of this technique that alleviate the overhead.

3.2 Distributed training

3.2.1 Data parallelism

Due to its simplicity, data parallelism, where the training mini-batch is split across multiple workers each of which holds a replica of the model [Valiant, 1990; Chen et al., 2018], is the most commonly used type of parallelism in deep learning. It can be utilized to train memory demanding models by splitting the minibatch so that each worker has a small portion using which it can fit into its memory. A fundamental limitation here is that the model must fit entirely in one workers memory with a mini-batch size of at least 1, which might not be possible [Shoeybi et al., 2019; Wang et al., 2019; Shazeer et al., 2018; Lepikhin et al., 2020]. Even for models that fit into a single device memory but have a large number of parameters, data parallelism becomes a scalability bottleneck due to the large parameters that need to be replicated and gradients that need to be exchanged [Jia et al., 2018b; Wang et al., 2019].

3.2.2 Computational graph partitioning (device placement)

Mirhoseini et al. proposed a reinforcement learning-based method to place dataflow graphs on multiple devices [Mirhoseini et al., 2017, 2018]. 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. [Wang et al., 2019], they use dynamic programming to partition the graph within few minutes . Despite being efficient, this approach has several limitations: (a) it requires writing specifications of what the operator computes using a lightweight description language that doesn't cover all the operations in TensorFlow, (b) it depends on a limited partition and reduce pattern that requires each worker to perform a coarse-grained task identical to the original computation, and (c) it always partitions all operators and tensors across all workers resulting in poor utilization of resources.

3.2.3 Model parallelism (DL-level approach)

Simple 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 [Narayanan et al., 2019]. Mest-Tensorflow [Shazeer et al., 2018] provides a more elegant way to express model parallelism, but the model still need to be modified using Mesh-Tensorflow syntax and the user is still responsible for deciding the partitioning.

Pipeline parallelism provides good resource utilization but some implementations assume that a single layer fits within a single device [Huang et al., 2019], which may not be the case for models with 3D inputs [Mathuriya et al., 2018]. While in others, extra memory overhead proportional to the size of the model weights is necessary to address the statistical efficiency issue which if not handled, it can prevent model convergence [Narayanan et al., 2019].

In [Jia et al., 2018a; Krizhevsky, 2014; Dryden et al., 2019; Shoeybi et al., 2019] non-generic techniques were proposed. These techniques are tightly coupled to certain operations or model structures, and were proposed to parallelize specific types of DNNs. While some of them are focusing on CNNs [Jia et al., 2018a; Krizhevsky, 2014; Dryden et al., 2019]; others rely on Transformer architecture in their optimization [Shoeybi et al., 2019; Lepikhin et al., 2020].

Chapter 4

PARDNN: A DNN PARTITIONING STRATEGY

4.1 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 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 4.1 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 as shown in Figure4.2. 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 three 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 by firstly merging the clusters that have no parallelism gain, then merging the rest of S using a novel load balancing algorithm. The final stage of Step-1 is a *re nement*

Figure 4.1: ParDNN Overview

of the mapping through path swapping and node switching. In Step-2, the result from Step-1 is validated against the memory constraints of the given devices. If the memory constraints are satis ed, the partition will be the nal output. Otherwise, the partition is re ned 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.

$$Max_{([0 \leq t \leq C_{tmax}],\ pe \in PE)} M_{cons}(pe, t) \leq M_{capacity}(pe) \tag{4.1}$$

Table 4.1 summarizes the time complexity of each step of ParDNN. The reported complexities after each step are relaxed ones and for some stages a tighter bounds maybe driven with amortized analysis. Splitting the partitioning strategy into a set of simple, yet e cient, sub-stages permits lowering the complexity. The nodes are grouped into clusters in the rst step, then for the most of later stages, ParDNN works at the cluster rather than node granularity, which considerably reduces the instance size it deals with.

In practice, running ParDNN on the DNN models listed in Table 5.1 takes up to 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,

Table 4.1: Complexity of Each Step of ParDNN

Step-1	Partitioning to Minimize Makespan
Graph Slicing (inc. sorting)	$O(K(\|V\|+\|E\|))$
Mapping	$O(\|V\| \log\|V\|)$
Refinement	$O(K(\|V\|+\|E\|))$
Step-2	Satisfying Memory Constraints
TensorFlow Scheduler Emulator	$O(\|V\|+\|E\|)$
Tracking Memory Consumption	$O(\|V\|)$
Addressing Overflow	$O(\|V^2\|)$
Overall complexity of ParDNN	$O(\|V\|^2)$

ParDNN offers an extremely lightweight and practical approach to partition the computational graphs of DNNs.

Next, we explain the details of each step along with the time complexity. Table 4.2 summarizes the terms and notations used during explanations.

4.1.1 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 [Rajbhandari et al., 2019]. 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 [Bichot and Siarry, 2011]. This grouping reduces the problem size and allows good quality heuristics to

Table 4.2: 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
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 (compute time)
$mem(n)$	Memory consumption of outputs of a node n
$comm(e)$	Cost of an edge e (communication time)
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 $st(n) \leq t \leq ft(n)$

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 ≠   do
 4:     heaviest_path    nd _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  f heaviest_pathg
12:     j    j + 1
13: end while
```

be applied within a reasonable time. Inspired by this method, we designed Step-1 of our partitioner in three main stages.

Graph slicing

This stage groups the nodes of the graph into disjoint clusters. It iteratively nds 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 di erent processing elements. Hence, the terms primary cluster and pe are going to be used interchangeably. After nding those primary clusters, if there are leftover nodes, we group them into *secondary clusters*. A secondary cluster, which is a linear cluster [Sinnen, 2007], is either a single node or a path. All the secondary clusters are identi ed and tagged until there is no node left on the graph that is not part of any cluster. Figure 4.2(b) shows an example.

Algorithm 1 shows the pseudo-code of the graph slicing, which takes K and graph G as inputs and outputs primary and secondary clusters. Line 2 computes the weighted level ($w\_lvl(n)$) for all 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 [Kim, 1988], 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 6.4.2 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, it rather returns a path of a heavy cost. This aims at capturing dependent and heavily-communicating nodes in one cluster to increase locality. 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 $O(|V|+|E|)$ [Sinnen, 2007]. It is done K times, resulting in an overall complexity of $O(K(|V|+|E|))$. While in linear clustering that would cost $O(|V|(|E|+|V|))$ [Wang and Sinnen, 2018]. Finding the paths given that the priorities are already specified - the nodes weighted levels - costs $O(|E|)$. This because each node is visited at most once, since the nodes are removed form 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 we have $O(|E|)$ overall steps. It is important to note that all the graphs we experimented with are sparse, having $|E| < |V|\log(|V|)$, the rest of the analyses follows this observation.

(a) (b) (c) (d) (e)

Figure 4.2: In the computational graph, vertex and edge weights indicate computation and communication costs, respectively. (a) Original computational graph, source node (A) and sink node (L) are added. (b) Shows the slicing stage when there are two pe(s). Clusters are found in the following order: $\{A; B; E; G; I; K; L\}$, $\{C; F\}$, $\{J\}$, $\{D\}$, $\{H\}$. First two are primary clusters, the other three are secondaries. (c) Cluster $\{J\}$ is merged to a primary cluster in initial merging since it has a communication of 5 units that cannot be hidden by the work within its span. (d) and (e) show LALB (ours) and GLB load balancing algorithms, respectively, after initial merging. The makespan of the LALB output is 13 units, while GLB is 15, resulting in 15% slowdown.

Mapping

This stage attaches the secondary clusters to the primaries with the objective of balancing the load among partitions and reducing communication . First, initial merging is applied to some of the secondary clusters for which execution in parallel is not advantageous. For example, in Figure 4.2(c), the cluster $\{J\}$ is merged with a primary because the total amount of communication ($comm$) incurred by the nodes in cluster $\{J\}$ can not be covered by its $potential(\{J\})$. Intuitively, the potential of a cluster measures how much parallel work is in the cluster at the time of its execution and whether or not that work is sufficient to totally hide its communication. In other words $comm(\{J\}) - potential(\{J\}) > 0$, hence there is no gain from assigning it to a distinct *pe*. Such a cluster is merged with the primary cluster with which it communicates the most.

Second, we apply a level-aware load balancing technique at which we merge the secondary clusters that are not merged by the initial merging. This process is referred to as *cluster mapping* in the scheduling literature. There are some heuristics such as wrap cluster merging [Yang, 1993], list scheduling based cluster assignment [Sarkar, 1988], and Guided Load Balancing (GLB) [Radulescu and Van Gemund, 1998]. In a comprehensive evaluation of scheduling and cluster merging algorithms in [Wang and Sinnen, 2018], 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 [Radulescu and Van Gemund, 1998]. Ignoring communication cost results in a low-quality mapping when the graph becomes very large and the number of communication operations increases. 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 issue is demonstrated in Figure 4.2(d) and (e). Ignoring the temporal aspect of load balancing causes GLB to make locally sub-optimal decision for cluster $\{D\}$. It assigns it to the less loaded *pe*, yet that *pe* has more work within the *span*($\{D\}$). This assignment pattern is not suitable especially for graphs with frequent forks and joins, where the local and the global loads become more uncorrelated. TensorFlow graphs that we have encountered so far have frequent fork-join structures.

We propose a novel time-efficient heuristic called *Level-Aware Load Balancing (LALB)*. LALB considers both communication minimization and the temporal load balancing. We temporally balance the loads by considering the workload of every *pe* within *span*(*sc*), where *sc* is the secondary cluster that is going to be merged with 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 (4.2) shows the selection criteria. In case of ties, we assign *sc* to the *pe* which has the highest communication value with it.

$$\min_{pe \in PE} \sum_{\substack{n \in pe \\ tl(n) \in span(sc)}} comp(n) + \sum_{\substack{(n,u) \in E; \\ n \in \{PE\} \setminus pe; \\ u \in sc}} comm(n,u) + \sum_{\substack{(u,n) \in E; \\ n \in \{PE\} \setminus pe; \\ u \in sc}} comm(u,n) \tag{4.2}$$

Algorithm 2 shows a high level description of the two procedures of mapping. Lines 1-7 do the initial merging of the secondary clusters to the primary clusters. The while loop (Line 9) applies our novel load balancing. The most time consuming part in this algorithm is to calculate the work within the span of the target secondary cluster *sc* (Line 11) 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 [Fenwick, 1994] as a data structure, where the tree nodes store the weights per level. This data structure allows logarithmic range summation and value updates. Line 12 calculates the cost of communication between the secondary cluster *sc* in each of the primary clusters. Line 13 performs the selection criteria defined in Equation (4.2) to select the best primary cluster to merge the *sc* with.

Complexity: Since the clusters are disjoint paths or singular nodes, and have no common nodes (a node exists only in one cluster), 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 $|V|$, so we end up with $O(|V| \log |V|)$. Before starting LALB, we sort the clusters by their weights (the heaviest clusters first due to their importance in balancing the loads), this has an upper bound of $O(|V| \log |V|)$, since the number of clusters is upper-bounded by the number of nodes. Hence, the overall complexity of the mapping stage is $O(|V| \log |V|)$.

Refinement

This stage refines the partitioning with two refinement policies. The first is responsible for coarse-grained refinement at the secondary cluster level and the second

Algorithm 2 Mapping

```
    InOut: pri_clusters[ ]
    In: sec_clusters[ ]
 1: for $sc \in$ sec_clusters do
 2:     if comm(sc) $-$ potential(sc) $> 0$ then
 3:         target $\leftarrow$ find_most_comm(sc, pri_clusters)
 4:         target $\leftarrow$ target + $\{sc\}$
 5:         sec_clusters $\leftarrow$ sec_clusters $-$ $\{sc\}$
 6:     end if
 7: end for
 8: comps[ ] $\leftarrow \emptyset$, comms[ ] $\leftarrow \emptyset$
 9: while sec_clusters $\neq \emptyset$ do
10:     sc $\leftarrow$ remove_next_secondary(sec_clusters)
11:     comps $\leftarrow$ calc_work_at_span(span(sc), pri_clusters)
12:     comms $\leftarrow$ calc_comms_with(sc, pri_clusters)
13:     target $\leftarrow$ find_minimal(comps, comms)
14:     target $\leftarrow$ target + $\{sc\}$
15: end while
```

does the fine-grained refinement at the node level. The first policy searches for a secondary cluster sc for which there is another secondary cluster sc^0 within its span that when swapped with sc, it results in better quality partitioning. The better quality comes from either enhancing the load balancing or reducing the total communication, or both.

The second policy handles a general issue with CP based heuristics that is discussed in [McCreary et al., 1996]. This issue arises on the communication-edges among the processing elements. When there are many costly communicating operations in G, some of them may fall outside the CP. If their effect is large enough, they will create heavier CPs in the partitioned graph. Note that the CP of the graph after partitioning is probably different than the original CP formed initially. For example, in Figure 4.2(a) the CP is initially $\{A; B; E; G; I; K; L\}$, but after partitioning, the CP becomes $\{A; C; F; G; I; K; L\}$ as in Figure 4.2(d). This is because the communication between the nodes in the same cluster is assumed to be zero. We repeatedly find the CP in the partitioned graph, as in Algorithm 1. Then we check the edges in that path that connect two different primaries, if moving a node incident to any of these edges to another primary shortens the CP, we switch that node primary. This process could be repeated as long as the CP can be shortened but since each time w_lvls need to be recalculated to nd the new CP, we choose to do it at most K times.

Complexity: For swapping, initially we sort the clusters by $tl(n)$ of their source nodes to nd the clusters within the span of a certain cluster using binary search. Since the number of disjoint clusters is bounded by the number of nodes, this process gives the complexity of $O(|V| \log(|V|))$.
Once two clusters are swapped, they are marked and not considered again leading to at most $|V|$ cluster, hence node, swaps. With each swap the binary-indexed-trees are updated to re ect the new work loads. Since each update takes $O(\log(|V|))$; overall complexity is $O(|V| \log(|V|))$. The node-level re nement is repeated K times and each time we recalculate the weighted levels and the CP $O(K(|V|+|E|))$. Upon node switching we update the trees $O(|V| \log(|V|))$. The overall time complexity of this step is $O(|V| \log(|V|))$.

4.1.2 Step 2: Validating Memory Constraints

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 con icting with other runtime optimizations. Optimizing memory consumption of production DL frameworks is an area of an ongoing development and the memory management modules of those frameworks go through continuous modi cations. Our approach is implemented separately, and could be seamlessly used with any optimization policies provided by the frameworks. ParDNN results in an allocation that meets an upper bound of memory consumption, and a dynamic policy could use that result as an initial point to further optimize at runtime. Step-2 is further divided into three stages; a scheduler emulator, memory consumption tracking, and over ow handling.

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 nish execution. Consequently, when the memory allocated for the operation inputs is released and when a new memory is allocated to hold the operation outputs. Figure 4.3 shows how the schedule participates in changing the memory consumption of the model. To estimate those values, we emulate the TensorFlow scheduler. The TensorFlow scheduler maintains a ready queue that is initially lled with nodes with no ancestors. Each node in the graph has an in-degree representing the number of dependent nodes. 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 pro ling, 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 nishing time $ft(n)$ of the nodes in the graph. The emulator has time complexity of $O(|V| + |E|)$.

Tracking Memory Consumption

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 [Abadi et al., 2016] and we refer to them as *residual nodes* (res_ns). Second, special operations that mutate the referenced tensor, of the rst 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.

(a) (b) (c)

Figure 4.3: (a) a task graph where the numbers inside the nodes indicate the memory required to store the task outputs. (b) assuming the schedule A, C, D, E, F, B, G. When G is scheduled to run we have the outputs of B and F held in the memory plus the newly allocated memory for G output totalling to 8 units; which is the peak memory consumption for this schedule. (c) assuming the schedule A, B, C, D, E, F, G, the peak consumption occurs when F is scheduled totalling to 11 units (coming from B, C, D, E and F). The fading nodes are the ones for which all the dependencies have run, hence their memory is released since it is not needed any more.

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(4.3) 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 $\le t$ 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 $\ge t$, and those nodes have finished at $\le t$ at any processing element except pe, or are non-residual that have finished at $\le t$ on that pe.

$$M_{cons}(pe; t) = \sum_{\substack{n \in pe;\\ n \in res\_ns}} mem(n) + \sum_{\substack{n \in pe;\\ n \in nor\_ns;\\ st(n) \le t \le ft(n)}} mem(n) + \sum_{\substack{n \notin (pe \cap res\_ns);\\ ft(n) \le t;\\ \exists (n;u) \in E : st(u) \ge t; u \in pe}} mem(n) \tag{4.3}$$

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 4.2) 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.

Addressing Overflow

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 [Csirik, 1991]. 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:

$$\text{minimize} \sum_{j=1}^{N} c_j x_j \quad s.t. \sum_{j=1}^{N} a_j \geq O \text{ and } x_j \in \{0, 1\} \tag{4.4}$$

In our case, O represents the amount of memory overflow, and a_j represents $M_{pot}(n;t)$. For the cost c_j, 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(4.5), which defines *move_cost*.

$$comp(n) + \forall u, v \in pe(n) \sum_{u:(u,n)\in G} comm(u,n) + \sum_{v:(n,v)\in G,v} comm(n,v) \tag{4.5}$$

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 ≥ 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 *move_cost* 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 su cient 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 over ow 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 over ow 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 over ow on that *pe*. As a result, there is no repetition and a node can enter or exit the heap once, resulting in $O(jVj \log jVj)$. When a node is moved, the new potentials and memory consumption need to be recalculated ($O(jVj)$). It may happen at most jVj times. Overall the complexity is $O(jVj^2)$. However, as shown in the next chapter, this upper bound is much larger than the practical one as the number of nodes to be moved is usually much less than jVj.

Chapter 5

IMPLEMENTATION

5.1 Implementation

Our partitioning algorithm takes as an input the number of the devices, their memory capacities, the interconnection bandwidth and latency between them, the model computational graph, pro ling data, operations metadata, and a list of the graph nodes that have no GPU implementation. The pro ling data contains execution time measurements and the size of the output of each operation-node, which is used to estimate the communication times. The operation metadata contains the operation types, which is used to identify the node types. TensorFlow standard APIs provide the pro ling information including per-node time, memory consumption, and communication sizes at the granularity of graph nodes. User-de ned operators can be supported as these operators have to be registered through the TensorFlow APIs as any regular operation.

To estimate the memory consumption, we implemented an emulator of TensorFlow's scheduler described in Abadi et al. [2016]. It is important to note that if ParDNN 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. It is important to note that our memory handling part is loosely-coupled to this emulator, and independent of its internals. It just takes the sequence in which the nodes are expected to be executed. If intended to be used with another framework, another procedure can be written to emulate its scheduler ,if needed, without touching our algorithm. From the memory lifetime point of view, TensorFlow operation-nodes fall into three main categories, residual, reference, and normal nodes, as described in Section 4.1.2. Note that the reference operations are 'co-located' by TensorFlow with the variables that they are mutating, and have to be moved together in case the variable they are referring to is moved. There is also temporary memory allocated for operation's local variables. Those are immediately released once executed. When handling memory constraints there is a trade-o between the overhead and the accuracy; static handling prioritizes overhead reduction over accuracy while dynamic handling targets the opposite. Due to the e ciency and maintainability reasons discussed in 4.1.2, we adopt the static approach. To accommodate sacri cing 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. In step 2 we constrain ourselves to the remaining 90%. In practice this threshold was su cient as all our experiments run successfully without going OOM. Nevertheless, this ratio is empirical and might need to be tuned. That being said, it is the only parameter of ParDNN that needs tuning.

As shown in Figure 4.1, the collected data is passed to our algorithm, and the output is a single le containing the 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 back-end is modi ed to read the node-to-device assignment from the placement le generated by our algorithm.

Extending ParDNN to multiple nodes: We assume that processing elements are identical GPU devices connected to a common host. It is worth mentioning that 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 20th fastest system in November 2019 Top500 list). Despite the capability of designing ParDNN to use GPUs on di erent nodes, we argue that a hybrid approach of data parallelism across compute nodes (i.e. samples split between nodes) and using ParDNN inside the compute node is a practical choice. This approach bene ts from the e ciency and non-invasiveness of our method in tackling the memory capacity issue at the node-level, while also harnessing the data parallelism desirable weak scaling properties across the nodes. This hybrid approach

Table 5.1: Speci cations 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 Sung Kim [2017] / Tiny Shakespeare Karpathy [2015]	Word-RNN Word-RNN-2	8 8	2048 4096	28 25	0.34 1.28	10578 10578
			CHSD	ED		
Character-Aware Neural Language Models Kim et al. [2016] / Penn Treebank (PTB) Marcus et al. [1994]	Char-CRN Char-CRN-2	8 32	2048 2048	15 15	0.23 1.09	22748 86663
			#RU	WF		
Wide Residual Net. Zagoruyko and Komodakis [2016] / CIFAR100 Krizhevsky et al. [2009]	WRN WRN-2	610 304	101 50	14 28	1.91 3.77	187742 79742
			HSD	MD		
Transformer Vaswani et al. [2017] / IWSLT'16 German{English corpus Cettolo et al. [2016]	TRN TRN-2	24 48	5120 8192	2048 2048	1.97 5.1	80550 160518
		HSD	FS	P_SZ		
Eidetic 3D LSTMWang et al. [2018] / Moving MNIST digits Srivastava et al. [2015]	E3D E3D-2	320 512	5 5	4 8	0.95 2.4	55756 55756

was applied or suggested for further scaling by many state-of-the-art works Shoeybi et al. [2019]; Rajbhandari et al. [2019]; Huang et al. [2018].

Chapter 6

RESULTS

All of our experiments are conducted on a NVIDIA DGX-2 that has 16 Tesla V100 SXM3 32GB GPUs connected via NVSwitch with 300 GB/sec bandwidth between them. The measurements are conducted over the interval between the 100th and the 150th training iterations to get stable results. We used TensorFlow 1.14, and CUDA 10.0.

We summarize our key ndings of each part as follows:

- Scaling: (i) ParDNN enables a superlinear scaling in the mini-batch size. For the same number of GPUs, ParDNN enables the use of more than 9x batch size over the maximum possible with data parallelism on average. (ii) Superlinear speedup in many models and con gurations is observed going from one GPU to 16 GPUs.

- Comparison with Related Work: (i) ParDNN achieves similar performance to the distributed tensor computation framework, Mesh-TensorFlow [Shazeer et al., 2018] but provides much higher user productivity. (ii) ParDNN outperforms Gradient Checkpointing [Bulatov, 2018] combined with data parallelism in many cases, yielding up to 2.8x speedup. More importantly, ParDNN enables training models where applying Gradient Checkpointing results in out of memory (OOM) even with a batch size of 1. (iii) ParDNN outperforms CUDA Uni ed Memory for all con gurations and GPU counts.

- Overhead and Fidelity: (i) Empirical overhead of ParDNN is no more than 2 minutes for the largest model over 16 GPUs. (ii) Replacing any of ParDNN steps with other heuristics or using alternative approaches result in signi cant drop in performance or huge increase in overhead, hence, demonstrating and justifying the design choices and e ciency of ParDNN 's algorithmic steps.

6.1 Models and Datasets

To demonstrate our results we experimented with ve large models representing the main tracks of DL applications: image classi cation, video prediction, language modeling and translation. All the models and datasets used in our experiments are listed in Table 5.1. Our analysis focuses on the performance of ParDNN, rather than pursuing the accuracy as ParDNN has no e ect on the learning aspect of the model. More speci cally, the convergence and accuracy are mainly a ected 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 merely 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 [Sutskever et al., 2011]. Character-Aware Neural Language Models (Char-CRN) [Kim et al., 2016]. Both models can be enlarged by increasing the number of layers or the hidden state size. Wide Residual Network (WRN) [Zagoruyko and Komodakis, 2016] is a widened version of the residual network model. In WRN the width of the convolutional layers can be con gured. The model size grows quadratically when widened. WRN has achieved better accuracy when the model is widened [Zagoruyko and Komodakis, 2016]. TRN (Transformer) [Vaswani et al., 2017] is a widely used model that had a signi cant in uence on the design of SoTA Transformer-based models in the NLP domain such as GPT-2 [Radford et al., 2019] and Megatron-LM [Shoeybi et al., 2019]. Transformer can be enlarged by increasing the number of layers, which deepens the model, and by widening the inner-layer dimensionality. Deeper [Huang et al., 2019] and wider [Vaswani et al., 2017] con gurations of Transformer are shown to give higher accuracy. E3D is Eidetic 3D LSTM [Wang et al., 2018] for video prediction. This model achieved state-of-the-art performance in future frame prediction. E3D is closely related to convolutional recurrent networks, where the dimensions of memory states are increased, and 3D-Convs are adopted as the basic operators for state transitions. E3D can be enlarged by increasing the number of the hidden state channels on the memory dimensions.

6.2 Performance and Scaling Studies

We experimented with models under two main use-cases of ParDNN. First, model instances that t into a single device memory only with very small batch sizes. 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 di erent GPUs that hold the replicas of the model. The second use-case is model instances that do not t into a single GPU memory even with a mini-batch size of 1. These are larger variants of each model with up to 5:1B parameters in TRN-2.

6.2.1 Mini-batch Size Scaling

Training with large mini-batch sizes o ers more parallelism and drastically reduces the overall training time. For example, authors in [Goyal et al., 2017] 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 [You et al., 2019]. ParDNN enables superlinear scaling of the batch sizes while increasing the number of GPUs. Table 6.1 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 64x for use-cases-2. This gives ParDNN a qualitative advantage even for models that t into a single GPU because ParDNN enables training with much larger batch sizes than what can be achieved with DP. Table 6.1 also shows the potential advantage of ParDNN over ideal DP scaling. ParDNN achieves superlinear scaling of the batch size due to two main reasons. Firstly, 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

Table 6.1: Maximum batch sizes (bsz) made possible by ParDNN . Bsz on a single GPU is the maximum that could t 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.

	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	32	64	1x	2x	4x	4x	4x
TRN	1	32	64	128	256	1x	16x	16x	16x	16x
E3D	1	4	16	32	64	1x	2x	4x	4x	4x
Word-RNN-2	{	{	32	1024	2048	{	{	1x	16x	16x
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
WRN-3	{	{	2	4	8	{	{	1x	1x	1x

(a) (b)

Figure 6.1: (a) ParDNN speedup using 2, 4, 8 and 16 GPUs over one GPU. (b) The throughput scaling of ParDNN with larger models on 4, 8 and 16 GPUs

parameters are distributed, but with DP they need to be replicated, which wastes considerable amount of memory.

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 scaling from batch size of 1024 to 2048, the memory consumption by matrix multiplication results increases by only 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.

6.2.2 Throughput Scaling and Speedup

For use-case-1, Figure 6.1(a) shows the speedup achieved using the batch sizes shown in Table 6.1 over a single GPU. ParDNN shows a substantial improvement on 2 GPUs and a superlinear speedups using up to 4 GPUs for all the models. For large models (use-case-2), Figure 6.1(b) shows the throughput and scaling of training achieved by ParDNN. In all cases, ParDNN throughput always improves going from 4 to 16 GPUs while increasing the batch size.

To analyze performance scaling, we need to consider three factors; (a) the maximum resource utilization that is achieved at the reference point, (b) the inherent-

Figure 6.2: The average degree of concurrency, or the degree of parallelism in the graphs.

degree of parallelism, also referred to as the average degree of concurrency which is the ratio of the whole wok in the graph to the work in the critical path [Kumar et al., 1994], in the graph, which indicates how much bene t can be gained when adding more workers (i.e. GPUs) and (c) the ratio of total communication cost to total computation cost (CCR) of the graph [Özkaya et al., 2019; Sinnen, 2007]. The sharp performance improvements of ParDNN happen because when a model ts into single GPU memory with small batch size, the resources are extremely under-utilized, and pushing larger batches, while doubling the number of GPUs improves the device utilization considerably.

In Figure 6.1(a) after 4 GPUs, for all the models, the batch size could only be doubled when doubling the GPU count. The performance behavior beyond this point depends on the inherent degree of parallelism and CCR of the graph. Figure 6.2 shows the degree of parallelism for the rst set of the models, the shown values are calculated using the 4-GPU minibatch sizes for all the graphs. It is important to note that these values of the DoP are representative for all con gurations. Increasing the batch size has a marginal e ect on the DoP since the pattern of the tasks weights is similar. Clearly, both Word-RNN and Char-CRN have large degrees of parallelism, as a result, they continue to give superlinear speedups up to 8 and 16 GPUs, respectively. Note that the speedup is obtained due to both high DoP and improving GPUs utilization.

TRN exhibits modest improvement going from 4 to 16 GPUs even though its graph has higher degree of parallelism in comparison to WRN. This is because it

Figure 6.3: ParDNN speedup over Mesh-TensorFlow using 4 GPUs and Transformer (TRN) model

has a larger CCR (e.g. on 4-GPU con gurations the CCR of TRN is 1:58 compared to 0:59 of WRN), hence a considerable amount of time is spent on communication. E3D scales better going from 4 to 16 GPUs in comparison to WRN due to having higher degree of parallelism. However, the achieved speedup, compared to a single GPU, is less because E3D's main operation is 3D convolution, which is a heavy kernel utilizing the GPU even with small batch sizes.

In Figure 6.1(b), going from 4 to 8 GPUs enables much larger batches in all cases except with E3D-2. This in turn enhances the resource utilization and results in the substantial throughput improvements. In TRN-2 it is the main factor of the throughput improvement. Char-CRN-2 perfectly scales up to 16 GPUs due to its high degree of parallelism. Word-RNN-2 and WRN-2 scale modestly from 8 to 16 because the batch size of 8 was su cient to saturate the GPUs for Word-RNN-2, while in case of WRN-2, the modest scaling is due to the low degree of parallelism.

Analysis of CCR for all ve models showed the same pattern: an increase in the CCR ratio as the batch size decreases, e.g. CCR of Word-RNN drops from 0.67 to 0.17, using the batch size of 256 and 2048, respectively. This is an expected outcome given that decreasing the batch size reduces the work per GPU and causes the strong scaling e ect of overheads taking over.

6.3 Comparison with Related Work

We compare ParDNN to three di erent state-of-the-art approaches used to circumvent the memory limitation when training DNNs. We compare with (i) Mesh-TensorFlow [Shazeer et al., 2018] for distributed training, (ii) gradient checkpointing [Bulatov, 2018] in combination with data parallelism for redundant recompute and (iii) CUDA Uni ed Memory for out-of-core computing. Although there exists other graph-based solutions, we cannot directly compare either because we are not aware of any open source implementation [Mirhoseini et al., 2018] or the implementation is available for MXNet only [Wang et al., 2019]. It is worth mentioning, however, that ParDNN takes no more than 2 minutes for the largest con guration we tested, in comparison to 10s of hours reported by the other graph-based methods, in addition to ParDNN working on models 2.3x as large as what the other methods experimented with [Mirhoseini et al., 2018].

6.3.1 Mesh-TensorFlow

Mesh-TensorFlow [Shazeer et al., 2018], an extension to TensorFlow, was proposed to overcome the memory limitations of a single device and permits specifying a general class of distributed tensor computations. We compare the performance of ParDNN with Mesh-TensorFlow using the Transformer model which the original authors used to demonstrate the scaling [Shazeer et al., 2018]. Figure 6.3 shows the speedup of ParDNN over Mesh-TensorFlow using 4 and 8 GPUs. We report all permutations [Vaswani et al., 2018] possible with the maximum trainable batch size for Mesh-TensorFlow. ParDNN is on par with Mesh-TensorFlow, however, 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 nding the best assignment is an NP-hard problem. (c) Mesh-TensorFlow has a non-negligible

(a) (b)

Figure 6.4: (a) ParDNN speedup using 2, 4, 8 and 16 GPUs over one GPU. (b) The throughput scaling of ParDNN with larger models on 4, 8 and 16 GPUs

pre-run overhead which doubles when doubling the number of GPUs reaching 1 hour for 8 GPU assignment.

6.3.2 Redundant Recompute: Gradient Checkpoint

Gradient checkpointing [Chen et al., 2016] is a general approach that works with both convolutional and recurrent neural networks. It 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 [Bulatov, 2018]. Figure 6.4 shows the speedup of ParDNN over gradient checkpoint when combined with data parallelism to run on multiple GPUs. For ParDNN and checkpointing, we used the common largest possible batch sizes. ParDNN outperforms gradient checkpointing in most cases. In few cases, checkpointing is better than ParDNN ; this happens mainly when the degree of parallelism inherent in the graph is not su cient to utilize all the GPUs. However, more importantly, ParDNN is qualitatively superior to gradient checkpointing since it enables the training of models where checkpointing fails to make them t in device memory, even when using a batch size of one. For example, Figure 6.4 shows several con g-urations where gradient checkpointing goes out-of-memory at the batch size of one.

(a) (b)

Figure 6.5: (a) ParDNN speedup using 2; 4; 8 and 16 GPUs over one GPU. (b) The throughput scaling of ParDNN with larger models on 4, 8 and 16 GPUs

Moreover, the overhead of gradient checkpointing can be up to 5 hours [Bulatov, 2018].

6.3.3 Out-of-core: CUDA Uni ed Memory

Figure 6.5 shows the speedup of ParDNN over CUDA Uni ed Memory (UM). UM, to the best of authors knowledge, is the only out-of-core solution that has an available Tensor ow implementation. In all cases, ParDNN throughput always improves going from 4 to 16 GPUs while increasing the batch size. UM performance in this case degrades when increasing the batch size due to the page faulting penalty [Awan et al., 2018].

6.4 Overhead and Fidelity of ParDNN

6.4.1 Analysis of ParDNN Algorithmic Steps

Step-2 of ParDNN has to be done to insure that the model will run successfully without going OOM. On the other hand, one might argue that Step-1 could be replaced by a naive partitioning approach. To analyze the impact of slicing-mapping-re nement stages of Step-1, we replace Step-1 with a naive approach, which simply

Figure 6.6: Makespan of ParDNN over that of Linear clustering (lower is better). Results are rounded to the 2nd digit after the decimal point.

distributes the graph-nodes to the devices in a round-robin fashion (RR), where the nodes of the graph are iterated in their topological order. Figure 6.6 shows the performance improvement by ParDNN. To breakdown the performance of step1 more, we applying slicing and mapping which are necessary to be done together to give a valid partitioning and compare the results with applying step1 fully. Note that slicing only without mapping would give an invalid partitioning since there will be clusters more than the available processing elements. Compared to RR results, ParDNN doubles the training throughput on average. Applying re nement has a non-negligible e ect contributing to 5-25% improvement.

ParDNN has a negligible overhead thanks to the low complexity of Step-1 ($O(K(|V|+|E|))$). The longest partitioning time among all the combinations of batch sizes, GPUs and model con gurations used in this work was 117 secs 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 over ow takes most of the overall partitioning time, the time taken by handling memory over ow is much lower than the theoretical upper bound. This is because the performance of Step-2 of ParDNN depends on how many nodes need to be moved between clusters to address the over ow, which is much less than $|V|$ in practice. The average ratio of the moved nodes in all our experiments is 8%.

Figure 6.7: Makespan of ParDNN over that of Linear clustering (lower is better). Results are rounded to the 2nd digit after the decimal point.

6.4.2 ParDNN vs Linear Clustering

ParDNN is not a standalone scheduling algorithm, rather it is designed to nd a spacial allocation of the nodes and leaves the order of execution decision to the dynamic scheduler. However, it can still serve as an e cient phase in static scheduling. To show this feature and the advantage of our multi-staged approach over a high-complexity single heuristic, and to demonstrate that scarifying a high-complexity did not harm the quality, we compare ParDNN with linear clustering (LC). To do a fair comparison, we implemented LC with GLB and Earliest Estimated Time First (EST First) [Wang and Sinnen, 2018] as a task ordering heuristic since this combination gave the best results. For ParDNN, we used EST First as well to derive the execution order of tasks and omit the memory constraints (Step 2).

Figure 6.7 shows that in all the experiments ParDNN is on par with or outperforms LC except for (TRN, $K = 4$), where ParDNN's makespan over LC's was $1{:}0009$. In particular, ParDNN produces much better results than LC when the degree of parallelism is high as in Char-CRN and Word-RNN. For the smallest graph, namely Word-RNN with 12K nodes it takes ParDNN 6 secs to partition while it took LC, which has $O(|V|(|V|+|E|))$ time complexity, 192 secs. For the largest graph, however, WRN with $190K$ nodes, it took ParDNN 36 secs while LC took about 4.5 hours.

Chapter 7

CONCLUSION

In this thesis, we presented a practical, automatic, generic, and lightweight approach to parallelize the training of memory-constrained Deep Learning models using data ow-graph partitioning. Our approach, realized in ParDNN, enables e cient training of models that do not t into a single accelerator memory, as well as those which t using small mini-batch sizes.

ParDNN does an o ine partitioning, it operates on the graph representation of a DNN annotated with a pro ling data including computation, communication, and memory consumption of the graph operation-nodes. As the graphs are growing in size, ParDNN seeks to quickly partition the graph while addressing the locality-parallelism trade-o and meeting the used processing elements' memory constraints. ParDNN incorporates a set of heuristics that could be used combined or independently in designing static scheduling and directed graph partitioning algorithms.

We conducted scaling experiments on ve representative DL models from multiple domains. Using these models, we compared ParDNN performance to memory optimization techniques(Gradient Checkpointing), out-of-core methods (CUDA Uni- ed Virtual Memory), and expert-guided distributed training (Mesh-Tensor ow). The experiments demonstrate ParDNN scalability and competence. The experiments show a superlinear scaling of the mini-batch size and a superlinear followed by a continuous scaling in the training throughput as long as the graph DOP supports it. They show that ParDNN outperforms the state-of-the-art automatic training alternatives, and has equivalent performance to the ones requiring expert intervention.

ParDNN does not have to be used exclusively. As the partitioning is done o ine, it can be integrated with any dynamic memory or performance optimization techniques where its output forms a starting point solution. ParDNN assumes a single node of fully connected processing elements but can be integrated with Data Parallelism to scale to multiple nodes.

BIBLIOGRAPHY

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

Augonnet, C., Thibault, S., Namyst, R., and Wacrenier, P.-A. (2011). Starpu: a uni ed platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187{198.

Awan, A. A., Chu, C.-H., Subramoni, H., Lu, X., and Panda, D. K. (2018). Oc-dnn: Exploiting advanced uni ed 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)*, pages 143{152. IEEE.

Bader, D. A., Meyerhenke, H., Sanders, P., and Wagner, D. (2013). *Graph partitioning and graph clustering*, volume 588. American Mathematical Society Providence, RI.

Bergstra, J., Bastien, F., Breuleux, O., Lamblin, P., Pascanu, R., Delalleau, O., Desjardins, G., Warde-Farley, D., Goodfellow, I., Bergeron, A., et al. (2011). Theano: Deep learning on gpus with python. In *NIPS 2011, BigLearning Workshop, Granada, Spain*, volume 3, pages 1{48. Citeseer.

Bichot, C.-E. and Siarry, P. (2011). *Graph partitioning*. Wiley Online Library.

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

Cettolo, M., Jan, N., Sebastian, S., Bentivogli, L., Cattoni, R., and Federico, M. (2016). The iwslt 2016 evaluation campaign. In *International Workshop on Spoken Language Translation*.

Chen, C.-C., Yang, C.-L., and Cheng, H.-Y. (2018). E cient and robust parallel dnn training through model parallelism on multi-gpu platform. arXiv preprint arXiv:1809.02839.

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

Chen, T., Xu, B., Zhang, C., and Guestrin, C. (2016). Training deep nets with sublinear memory cost. ArXiv, abs/1604.06174.

Csirik, J. (1991). Heuristics for the 0-1 min-knapsack problem. Acta Cybernetica, 10(1-2):15{20.

Davis, A. L. and Keller, R. M. (1982). Data ow program graphs.

Dryden, N. et al. (2019). 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, pages 46:1{46:13.

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

Gerasoulis, A. and Yang, T. (1992). A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. journal of parallel and distributed computing, 16(4):276{291.

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

Goodfellow, I., Bengio, Y., and Courville, A. (2016). Deep learning. MIT press.

Goyal, P., Dollar, P., Girshick, R., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch sgd: Training imagenet in 1 hour. arXiv preprint arXiv:1706.02677.

He, K., Meng, X., Pan, Z., Yuan, L., and Zhou, P. (2018). A novel task-duplication based clustering algorithm for heterogeneous computing environments. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):2{14.

He, K., Zhang, X., Ren, S., and Sun, J. (2016). Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 770{778.

Herrmann, J., Kho, J., Uçar, B., Kaya, K., and Çatalyürek, Ü. V. (2017). Acyclic partitioning of large directed acyclic graphs. In *2017 17th IEEE/ACM international symposium on cluster, cloud and grid computing (CCGRID)*, pages 371{380. IEEE.

Huang, Y., Cheng, Y., Bapna, A., Firat, O., Chen, D., Chen, M., Lee, H., Ngiam, J., Le, Q. V., Wu, Y., et al. (2019). Gpipe: E cient training of giant neural networks using pipeline parallelism. In *Advances in Neural Information Processing Systems*, pages 103{112.

Huang, Y. et al. (2018). GPipe: E cient Training of Giant Neural Networks using Pipeline Parallelism. *CoRR*, abs/1811.06965.

Hwang, J.-J., Chow, Y.-C., Anger, F. D., and Lee, C.-Y. (1989). Scheduling precedence graphs in systems with interprocessor communication times. *SIAM Journal on Computing*, 18(2):244{257.

Ito, Y., Imai, H., Le, T. D., Negishi, Y., Kawachiya, K., Matsumiya, R., and Endo, T. (2019). Pro ling based out-of-core hybrid method for large neural networks: poster. *ArXiv* , abs/1907.05013.

Jain, P., Jain, A., Nrusimha, A., Gholami, A., Abbeel, P., Gonzalez, J., Keutzer, K., and Stoica, I. (2020). Breaking the memory wall with optimal tensor rematerialization. In *Proceedings of Machine Learning and Systems 2020*, pages 497{511.

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

Jia, Z., Zaharia, M., and Aiken, A. (2018b). Beyond Data and Model Parallelism for Deep Neural Networks. CoRR, abs/1807.05358.

Karpathy, A. (2015). tinyshakespeare.

Karypis, G. (1998). Metis, a software package for partitioning unstructured graphs, partitioning meshes, and computing ll-reducing orderings of sparse matrices version 4.0. http://glaros. dtc. umn. edu/gkhome/metis/metis/download.

Karypis, G. and Kumar, V. (1995a). Analysis of multilevel graph partitioning. In Supercomputing'95: Proceedings of the 1995 ACM/IEEE conference on Supercomputing, pages 29{29. IEEE.

Karypis, G. and Kumar, V. (1995b). Multilevel graph partitioning schemes. In ICPP (3), pages 113{122.

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

Kim, Y., Jernite, Y., Sontag, D., and Rush, A. M. (2016). Character-aware neural language models. In Thirtieth AAAI Conference on Arti cial Intelligence.

Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks (2014). arXiv preprint arXiv:1404.5997.

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

Kumar, V., Grama, A., Gupta, A., and Karypis, G. (1994). Introduction to parallel computing, volume 110. Benjamin/Cummings Redwood City, CA.

Kwok, Y.-K. and Ahmad, I. (1995). Bubble scheduling: A quasi dynamic algorithm for static allocation of tasks to parallel architectures. In Proceedings. Seventh IEEE Symposium on Parallel and Distributed Processing, pages 36{43. IEEE.

Kwok, Y.-K. and Ahmad, I. (1996). Dynamic critical-path scheduling: An e ective technique for allocating task graphs to multiprocessors. IEEE transactions on parallel and distributed systems, 7(5):506{521.

Lam, M. S. and Rinard, M. C. (1991). Coarse-grain parallel programming in jade. In Proceedings of the third ACM SIGPLAN symposium on Principles and practice of parallel programming, pages 94{105.

Lepikhin, D., Lee, H., Xu, Y., Chen, D., Firat, O., Huang, Y., Krikun, M., Shazeer, N., and Chen, Z. (2020). Gshard: Scaling giant models with conditional computation and automatic sharding. arXiv preprint arXiv:2006.16668.

Liou, J.-C. and Palis, M. A. (1997). A comparison of general approaches to multiprocessor scheduling. In Proceedings 11th International Parallel Processing Symposium, pages 152{156. IEEE.

Marcus, M., Kim, G., Marcinkiewicz, M. A., MacIntyre, R., Bies, A., Ferguson, M., Katz, K., and Schasberger, B. (1994). The penn treebank: annotating predicate argument structure. In Proceedings of the workshop on Human Language Technology, pages 114{119. Association for Computational Linguistics.

Mathuriya, A. et al. (2018). Cosmo ow: 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, pages 65:1{65:11, Piscataway, NJ, USA. IEEE Press.

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

Meng, C., Sun, M., Yang, J., Qiu, M., and Gu, Y. (2017). Training deeper models by gpu memory optimization on tensor ow. In Proc. of ML Systems Workshop in NIPS.

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

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

Narayanan, D., Harlap, A., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., Gibbons, P. B., and Zaharia, M. (2019). Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15.

Newton, P. and Browne, J. C. (1992). The code 2.0 graphical parallel programming language. In *Proceedings of the 6th international conference on Supercomputing*, pages 167–177.

NVIDIA, T. (2017). V100 gpu architecture: The world's most advanced datacenter gpu. Technical report, Tech. Rep., NVIDIA, Also available at https://images. nvidia. com/content . . . .

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

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. (2019). Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037.

Pellegrini, F. (2009). Distillating knowledge about scotch. In *Dagstuhl Seminar Proceedings*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

Pellegrini, F. and Roman, J. (1996). 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*, pages 493–498. Springer.

Peng, X., Shi, X., Dai, H., Jin, H., Ma, W., Xiong, Q., Yang, F., and Qian, X. (2020). 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, page 891–905, New York, NY, USA. Association for Computing Machinery.

Radford, A., Wu, J., Child, R., Luan, D., Amodei, D., and Sutskever, I. (2019). Language models are unsupervised multitask learners. *OpenAI Technical Report*.

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

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

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

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

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

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

Sinnen, O. (2007). *Task scheduling for parallel systems*, volume 60. John Wiley & Sons.

Srivastava, N., Mansimov, E., and Salakhudinov, R. (2015). Unsupervised learning of video representations using lstms. In *International conference on machine learning*, pages 843–852.

Sung Kim, J. J. (2017). Multi-layer Recurrent Neural Networks (LSTM, RNN) for word-level language models in Python using TensorFlow.

Sutskever, I., Martens, J., and Hinton, G. E. (2011). Generating text with recurrent neural networks. In *Proceedings of the 28th international conference on machine learning (ICML-11)*, pages 1017–1024.

Tarnawski, J., Phanishayee, A., Devanur, N. R., Mahajan, D., and Paravecino, F. N. (2020). Efficient algorithms for device placement of dnn graph operators. *arXiv preprint arXiv:2006.16423*.

Tokui, S., Oono, K., Hido, S., and Clayton, J. (2015). Chainer: a next-generation open source framework for deep learning. In *Proceedings of workshop on machine learning systems (LearningSys) in the twenty-ninth annual conference on neural information processing systems (NIPS)*, volume 5, pages 1–6.

Valiant, L. G. (1990). A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111.

Vaswani, A., Bengio, S., Brevdo, E., Chollet, F., Gomez, A. N., Gouws, S., Jones, L., Kaiser, L., Kalchbrenner, N., Parmar, N., Sepassi, R., Shazeer, N., and Uszkoreit, J. (2018). Tensor2tensor for neural machine translation. *CoRR*, abs/1803.07416.

Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. (2017). Attention is all you need. In *Advances in neural information processing systems*, pages 5998–6008.

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

Wang, J., Lv, X., and Chen, X. (2016). Comparative analysis of list scheduling algorithms on homogeneous multi-processors. In *2016 8th IEEE International*

Conference on Communication Software and Networks (ICCSN), pages 708–713. IEEE.

Wang, M., Huang, C.-c., and Li, J. (2019). Supporting very large models using automatic dataflow graph partitioning. In *Proceedings of the Fourteenth EuroSys Conference 2019*, pages 1–17.

Wang, Y., Jiang, L., Yang, M.-H., Li, L.-J., Long, M., and Fei-Fei, L. (2018). Eidetic 3d lstm: A model for video prediction and beyond.

Yang, T. (1993). *Scheduling and code generation for parallel architectures*. PhD thesis, Citeseer.

Yang, T. and Gerasoulis, A. (1992). Pyrros: static task scheduling and code generation for message passing multiprocessors. In *ACM International Conference on Supercomputing 25th Anniversary Volume*, pages 163–172.

Yang, T. and Gerasoulis, A. (1994). Dsc: Scheduling parallel tasks on an unbounded number of processors. *IEEE Transactions on Parallel and Distributed Systems*, 5(9):951–967.

You, Y., Li, J., Reddi, S., Hseu, J., Kumar, S., Bhojanapalli, S., Song, X., Demmel, J., Keutzer, K., and Hsieh, C.-J. (2019). Large batch optimization for deep learning: Training bert in 76 minutes. In *International Conference on Learning Representations*.

Zagoruyko, S. and Komodakis, N. (2016). Wide residual networks. *arXiv preprint arXiv:1605.07146*.

Zahangir Alom, Tarek M. Taha, C. Y. S. W. V. S. M. S. N. M. H. B. C. V. E. A. A. S. A. V. K. A. (2019). A state-of-the-art survey on deep learning theory and architectures. *Electronics*, 8(3):292.

Zhang, L., Wang, S., and Liu, B. (2018). Deep learning for sentiment analysis: A survey. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 8(4):e1253.

Appendix goes here.