Kernel and Launch Time Optimizations For Deep Learning Frameworks

by

Doğa Dikbayır

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 and Science

July 22, 2019

Kernel and Launch Time Optimizations For Deep Learning Frameworks

Koç University

Graduate School of Sciences and Engineering

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

Doğa Dikbayır

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:

_____

Asst. Prof. Didem Unat

_____

Assoc. Prof. Mehmet Aktaş

_____

Prof. Sibel Salman

Date: _____

to my family...

ABSTRACT

Deep learning has become a prominent tool for extracting and exploring information in a wide selection of areas ranging from computer vision to natural language processing. With the increasing availability of modern hardware accelerators like GPUs and FPGAs, deep learning researchers had the opportunity to design more powerful and complex neural network architectures. Training time and memory restrictions, however, still act as a bottleneck in deep learning research. Utilizing system resources is essential in order to maximize training efficiency. Scheduling key computational operations such as convolution and mat-mul efficiently across all processing units (e.g. CPU cores and/or GPUs) in the system while minimizing the time spent on data communication is crucial. Existing solutions addressing these restrictions are not transparent and require manual partitioning of the model limiting the training performance of the frameworks. This thesis, attacks this problem from two angles: kernel optimization and model parallelism.

First the multidimensional reduction operation which is a frequently used operation in neural network training is analyzed and optimized. The mathemetical properties of the reduction operation are used to concurrently reduce many of its dimensions and eliminate unnecessary kernel launches. We compare our kernel's performance with the optimized Knet deep learning framework and achieve up to 56x speed-up and 75% of the theoretical device bandwidth of an NVIDIA K20m GPU.

In addition, a device placement algorithm based on depth-first search is proposed to reduce the programming effort for the researcher and improve resource utilization. The partitioning algorithm creates a cost-model for the target neural network based on the computational dependencies and memory transfers between the operations involved in the network and its and layers. Then the altered framework run-

time uses this cost model and heuristics to place the operations on devices . All the complexity of assigning different parts of the network into multiple devices is handled by our runtime, with high resource utilization. The proposed runtime is implemented on top of popular deep learning framework Tensorflow for evaluation and compared with the popular graph partitioning and load balancing tools METIS and Zoltan. Different state-of-the-art neural network architectures and training datasets are used in the experiments to measure the performance.

ÖZETÇE

Derin öğrenme, bilgisayar görüşünden doğal dil işlemeye kadar pek çok alanda bilgi çıkartmak ve keşfetmek için öne çıkan bir araç haline gelmiştir. GPU ve FPGA gibi modern hızlandırıcıların artan ulaşılabilirliği sayesinde derin öğrenme araştırmacıları daha güçlü ve daha karmaşık sinir ağı mimarisi tasarlama fırsatı bulmuştur. Ancak eğitim süreleri ve hafıza kısıtlamaları, halen derin öğrenme araştırma alanında darboğaz olmaktadır. Sistem kaynaklarını kullanmak, eğitim verimliliğini iyileştirmek açısından gereklidir. Konvolüsyon ve matris çarpımı gibi anahtar sayısal işlemleri bütün işlemciler (ör: CPU çekirdekleri ve GPU) arasında planlamak bu hususta çok önemli bir hale gelmiştir. Bu kısıtlamaları hedef alan mevcut çözümler şeffaf değildir yani kullanıcının manüel parçalamasını gerektirerek derin öğrenme çerçevelerinin eğitim sürelerini kısıtlamaktadir. Bu tez, bu problemi iki farklı açıdan ele almaktadir: çekirdek iyileştirmesi ve model paralelliği.

İlk önce, sinir ağı eğitiminde sıkça kullanılan çok boyutlu indirgeme işlemi analiz edilip iyileştirilmektedir. İndirgeme işleminin matematiksel özellikleri, birden fazla boyutun aynı anda indirgenip gereksiz çekirdek programi ateşlemelerinin yok edilmesi için kullanıilmaktadır. Geliştirdiğimiz çekirdek performansıını iyileştirme yapılmış Knet derin öğrenme çerçevesiyle karşılaştırdığımızda 56 kata kadar hızlanma ve NVIDIA K20m GPU makinesinin teorik bant genişliğinin 75%'ni elde etmekteyiz.

Bunlara ek olarak, yazılımcının programlama eforunu azaltmak ve kaynak kullanımını arttırmak için bir aygıt atama algoritması geliştirilmiştir. Algoritma, ağ ve katmanlarındaki işlemsel bağlılıkları ve hafıza aktarımlarını değerlendirerek, bir maliyet modeli oluşturmaktadır. Daha sonra, değiştirilmiş çerçeve çalışma zamanı bu maliyet modeli ve bazı buluşsal koşulları kullanarak işlemleri aygıtlara yerleştirmektedir.

Ağın farklı kısımlarını birden çok aygıta yerleştirme işleminin bütün karmaşıklığı çerçevemiz taraından kontrol edilmektedir. Önerilen çalışma zamanı, bilinen bir derin öğrenme çerçevesi olan TensorFlow'un üstüne geliştirilmiştir ve sonuçları, değerlendirme için, popüler birer çizge parçalama ve yük paylaştırma kütüphaneleri olan METIS ve Zoltan sonuçları ile karşılaştırılmıştır. Yapılan deneylerde, performans öçümleri, farklı ve modern derin sinir ağlarının kullanılması ile gerçekleştirilmiştir.

ACKNOWLEDGMENTS

First of all, I would like to thank my advisor Dr. Didem Unat for being a wonderful teacher throughout my research studies. I would not be able to walk on this long and dark road without her pointing the flashlight forwards. I also thank my summer research mentor Dr. M. Esat Belviranli for supporting me when I was abroad and co-guiding me with Dr. Unat in this challenging road. I thank all my professors for inspiring me and broadening my creativity.

I thank my family for supporting me no matter what, not only now but throughout my entire life. I want to specifically thank my beautiful mother for being such an amazing and strong woman and raising me.

I thank TUBITAK for supporting me financially throughout my graduate education.

I also thank all the members of ParCoreLab group for providing me assistance whenever I needed it. I specifically thank my dear friends Ilyas and Abay for supporting me in my most stressful periods during my graduate education.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

NOMENCLATURE

- API: Application Programming Interface
- CNN: Convolutional Neural Network
- CPU: Central Processing Unit
- CUDA: Compute Unified Device Architecture
- DAG: Directed Acyclic Graph
- DFP: Depth First Placement
- DFS: Depth First Search
- EASGD: Elastic Averaging Stochastic Gradient Descent
- FPGA: Field-Programmable Gate Array
- HPC: High Performance Computing
- LRAP: Least Recently Assigned Path
- MLP: Multi-layer Perceptron
- NMT: Neural Machine Translation
- FPGA: Heterogeneous Memory
- RNN: Recurrent Neural Network

- SGD: Stochastic Gradient Descent
- TF: TensorFlow
- TPU: Tensor Processing Unit

Chapter 1

INTRODUCTION

1.1 Motivation

Deep learning is the iterative process of training an artificial neural network consisting of multiple layers and it is currently being used to address different types of problems in various fields including image classification [Krizhevsky et al., 2012] , caption generation [Xu et al., 2015], medical image analysis [Zhou et al., 2017] and many more. An artificial neural network is a computational unit consisting of layers and neurons. In each iteration, the parameters in the network are improved based on their gradients from previous iterations. This iterative approach usually converges to higher accuracy values if the complexity of the neural network, i.e the number of neurons and connections, increase. More activations on the input data and more error gradients, help the algorithm to understand the samples in greater detail and perform more sophisticated tasks. However, this requirement results in great computational power need and very long training times. Moreover, neural network models are not getting smaller in most cases. One of the first neural networks LeNet [Lecun et al., 1998] consisted of only seven layers while the recent ResNet-50 [He et al., 2015] is 152 layers deep. Still being very crucial, the computational limitations of neural networks are not the only bottleneck in the field. The same factors slowing down the training time of these structures also increase the memory requirement significantly. More complex networks have more parameters and layers and therefore require more memory. The memory bottleneck may even prevent the training process if the model does not fit into the processing unit. All these limitations and bottlenecks underline the impor-

tance of high performance deep learning methods. Increasing popularity of different hardware accelerators like GPUs, FPGAs and the more recent TPUs created a heterogeneous computational environment full of optimization opportunities. Therefore, optimizing deep learning operations for these accelerators and managing the data communication and model distribution across different worker accelerators is critical for effective deep learning research.

1.2 Optimizing Multidimensional Reduction

Many mathematical operations are being frequently utilized in a deep learning process. Two of the arguably most frequently used operations in core machine learning kernels are broadcast and reduction operations. They are used in the computation of the gradient values of a loss function of a deep neural network, which indicates the accuracy of the prediction made by the network. More specifically, the broadcast operation is used for summing the bias parameters with the weight parameters of a deep learning model. The bias values are projected over the weight tensors. Then, the reduction operation is used in the backpropagation algorithm, which computes the gradient values of a loss function, based on these parameters. For example, cross-entropy loss function calculates the weighted average of the prediction values computed by the neural network by the ground-truth values. This calculation is performed by reducing the output tensor of the network. In addition to this fundamental usage of these operations, every calculation that involves tensors and a basic associative mathematical function can use broadcast or reduction operations in its core. For instance, the recent method and the network on neural caption with visual attention proposed by Xu et al utilizes both broadcast and reduction operations [Xu et al., 2015]. In their proposed long short-term memory model, the calculation of the hidden states involves the broadcast of the output state on the memory state, by multiplication; learning stochastic and deterministic attentions in the network requires the weighted sum of the predicted attention value tensors through reduction operation.

We first define a formal mathematical definition for both broadcast and reduction

operations. Using the definition, we propose a parallel multi-dimensional reduction algorithm for CUDA enabled GPU devices. In literature, this operation is either naively implemented or optimized only for a single dimension.
In our proposed parallel tensor reduction method, we exploit the associativity property of reduction operation and minimize the necessary synchronization points in the method. Instead of launching a reduction kernel for each dimension, we merge them to reduce the overhead of kernel launch and and write-back of the temporary data to the global memory. Since the order of the elements to be added together has no importance because of the associativity, we reduce the tensor in smaller independent partitions, and thus minimize the synchronization penalty between threads. Our arrangement of thread blocks and threads also eliminates the need for inter-block synchronization. This method allows the tensor to be reduced in a fully parallel fashion, increasing performance.

1.3 Transparent Model Parallelism for Deep Learning Frameworks

As mentioned earlier in this thesis, advances in neural systems result in more complex neural network architectures. Supporting such big and complex structures on a single device might be impractical. However, distribution of the model across different worker devices is also not a trivial task. Due to the data dependencies in the computational flow of a neural network, it might be very difficult to parallelize. Most deep learning frameworks like TensorFlow, MXNet and pyTorch [Abadi et al., 2015] [Chen et al., 2015] [Paszke et al., 2017] support GPU kernels, however, they only offer manual model parallelism where the programmer must tell the framework explicitly to which device each operation should be assigned. This requires a human-expert to partition the neural network computation in order to achieve high performance. This thesis investigates the possible strategies to transparently partition a neural network across different devices while achieving high performance. We propose a high-level architecture for our method. We first extract the computational flow as a data-flow graph from TensorFlow. Then we collect profiling data for different operations and

devices. We use this profiling data as weights in a graph partitioning algorithm and generate a device placement pattern for the operations in the computational flow. Finally, we alter the standard TF run-time code in order to enforce our device placement during the deep neural network model execution. In the graph partitioning phase we experiment with different graph partitioning algorithms and tools. We also propose a novel depth-first-search based heuristic device placement algorithm that tries to minimize the execution time of the training process.

Chapter 2

BACKGROUND

2.1 Tensor Notations

We introduce the necessary terminology to describe the broadcast and reduction operations on tensors. We inherit the terminology to describe tensors from [Liu et al., 2017] in order to formally present our methodology. The terms and notations to be used in the rest of the paper are defined as follows:

Tensor: Tensor is an array that consist of multiple dimensions. We use bold lowercase letters to address vectors and bold capital letters for matrices. Higher-order tensors are represented by calligraphic letters such as $\mathcal{T}$.

Order and modes: The dimensions of a tensor are called *modes*. The number of modes in a tensor is referred as its *order*. We use the triple pipe operator to notate the order of a tensor. For example, $|||\mathbf{b}||| = 1$ for vector $\mathbf{b}$ and $|||\mathbf{M}||| = 2$ for matrix $\mathbf{M}$.

Indexing: Tensor elements are accessed using coordinate values, similar to accessing multidimensional arrays in many popular programming languages. Each coordinate value represents the position of the element at the corresponding mode. Element with coordinate values (i,j,k) of a third-order tensor $\mathcal{T}$, is accessed as $\mathcal{T}(i,j,k)$. If all the elements in a mode are accessed, the colon notation is used. For example, $\mathcal{T}(:,j,k)$ refers to all the elements in the first mode of tensor $\mathcal{T}$.

Fibers and slices: Tensors can be segmented into different partitions over one or more modes. One dimensional partitions of a tensor are called *fibers* and two dimensional segments are defined as *slices*. Figure 2.1 demonstrates the yz slices and z fibers of the third-order tensor $\mathcal{T}$.

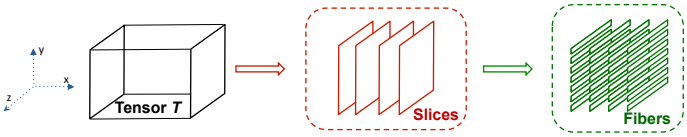

Figure 2.1: Example slices and fibers of tensor $\mathcal{T}$

Decomposition: Tensors can be decomposed into smaller partitions of lower order over $m_1, m_2, ..., m_n$ modes. The *decomposition* operation results in a decomposition list $\mathbf{D}$, consisting of d_i. Each d_i is an n-th order tensor containing the original tensor's elements over the $m_1, m_2, ..., m_n$ modes. We define these as *hyperslices*. The decomposition of a tensor is expressed using a superscript consisting of the modes of decomposition on the tensor's identifier. For example, $\mathcal{T}^{y,\,z}$, would result in all of $\mathcal{T}$'s yz slices, as it is shown in Figure 2.1.

2.2 CUDA and GPU

GPU accelerators have been utilized heavily in computer science research in the last decade. Thanks to their massively parallel architectures, they are great for solving compute-intensive problems like stencil solvers [Unat et al., 2010], 3D rendering [Mortensen et al., 2007] and many more. GPU machines are also crucial for deep learning research. Compute intensive matrix multiplications at the fully connected layers and convolution operations in the convolutional layers are great candidates for GPU computation, especially when the training input is also large. In such extreme scenarios, a few GPUs are known to outperform thousands of multi-core CPU processors in terms of training time [Cui et al., 2016].

The GPU architecture is different from the CPU architecture. CPUs have a few cores while GPU devices usually have hundreds of cores. The properties of these cores are also different. GPU cores are lightweight, they usually have small caches and computing power. Due to this high number of cores, compute intensive applications can efficiently run on a GPU device. Figure 2.2 [Ghorpade et al., 2012] compares the GPU architecture and CPU architecture. This massively parallel architecture allows

the GPU to concurrently compute millions of data points in parallel.

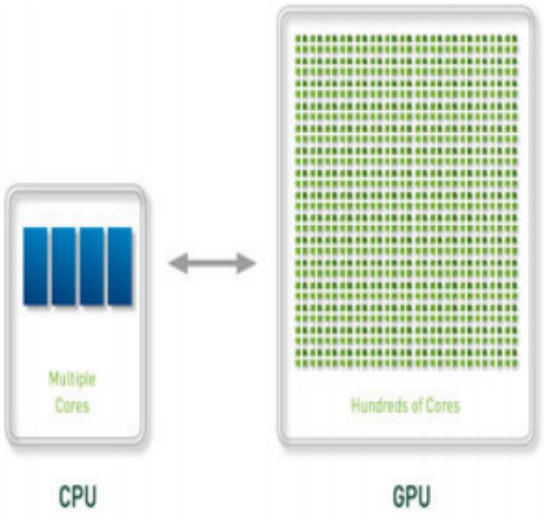

Figure 2.2: CPU (left) and GPU (right) architectures compared

In this thesis, all the experimentation is done on NVIDIA GPU devices, therefore, we will focus on NVIDIA GPU models. In order to make its devices easily programmable, NVIDIA launched the Kepler Architecture and CUDA GPU programming interface in 2007 [Nickolls, 2007]. CUDA is a C/C++ extension and an abstraction of the NVIDIA GPU architecture. In CUDA, the GPU device is composed of different components, all these components map the underlying hardware to the software stack and make the device easily programmable. In CUDA, the work is divided between threads. These threads are further grouped in different levels. The first level of grouping is a warp. A warp is a set of threads (32 in Kepler architecture) that is scheduled and managed together by the hardware. The next level of grouping consists of data structures called thread-blocks. Threads in the same thread-block are able to utilize the same shared memory and to be synchronized. Multiple thread-blocks form the kernel grid which contains all the threads running the kernel on the device. Synchronization between different thread-blocks is not possible during the same kernel execution.

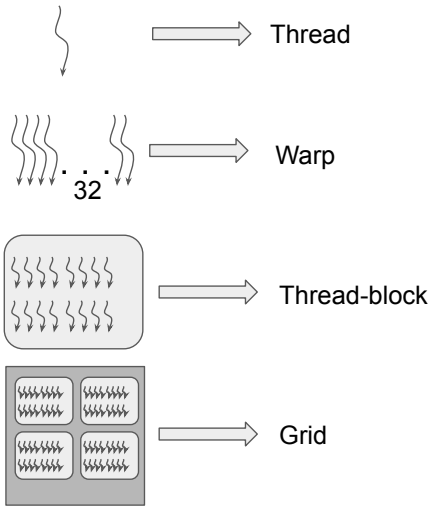

Figure 2.3: CUDA Abstraction

2.3 Deep Learning and Neural Networks

The first artificial neural network was a single neuron perceptron that was used for binary classification [Rosenblatt, 1958]. In the same research, a learning rule was established and in a later work, the perceptron convergence theorem was proposed stating that any data set which is linearly separable is solvable by the perceptron learning rule [Novikoff, 1962]. These advancements encouraged many other researchers to investigate the concept in more detail. In 1986, backpropagation algorithm was integrated into artificial neural networks and multi-layer neural network structures were able to be trained. With the increasing availability of computing power and hardware accelerators like GPU devices, researchers were able to propose larger network structures that require both much more memory and computing power [Krizhevsky et al., 2012] [He et al., 2015] [Simonyan and Zisserman, 2014]. This processing power enabled more layers to be added into the artificial network and created the concept of deep neural networks and deep learning.

Deep neural networks are multiple layers of neurons that are connected with each other through connections. Throughout the training process, these connections gain

different weights depending on their contribution to the error rate calculated using the ground truth training data. The error's gradient is calculated and back-propagated through all the layers. With this approach, each layer in the neural network learns certain patterns hidden in the training data and are therefore able to make accurate predictions about unseen data of similar properties [Rumelhart et al., 1986]. As discussed in the introductory chapter, deep learning have been successfully utilized in many different fields. The amount and variety of the training data encouraged researchers to develop specialized architectures.

It is important to understand the basic concepts and differences in neural network architectures to engineer efficient high performance computing solutions.

2.3.1 Mini-batch Stochastic Gradient Descent

Stochastic gradient descent is the core training algorithm for most deep neural networks. As mentioned earlier, the error's gradient is propagated backwards through all the layers in an iteration during the training process. We then use this gradient to update the weights (or parameters) of the neural networks. At the beginning of each iteration, the same error calculation and gradient backpropagation is performed on this new set of weights until convergence.

Stochastic gradient descent originally processes a sample at each iteration. This means the gradient is calculated over a single sample. While being computationally effective, this method may suffer from noisy data and take a long time to converge because of the number of the noisy steps taken. Moreover, most modern processors and hardware accelerators (especially GPUs) may be underutilized if a single sample is processed in each step. Both limitations can be tackled with mini-batch stochastic gradient descent. This algorithm is logically the same as gradient descent, but with a fundamental difference. Mini-batch SGD processes training samples in random equal-sized data chunks. We will use the term mini-batch and batch interchangeably in this thesis to refer to each of these chunks. Since a gradient is calculated on multiple data points this time, noisy gradients will be "softened" and the model therefore will have

good convergence properties [Shamir, 2016].

2.3.2 Multilayer Perceptrons

The earliest examples of deep neural networks were multi-layer perceptrons. These networks consist of multiple fully-connected layers. In a fully connected layer, there is an all-to-all correspondence between the neurons of consecutive layers [Murtagh, 1991]. A fully connected layer has great representational power as it considers all the connections between different neurons. However, it is also extremely expensive in terms of computational power which makes it impractical for deeper networks since it slows down the training significantly.

2.3.3 Convolutional Neural Networks

Arguably the most popular type of neural networks are convolutional neural networks. The idea was first proposed by Yann LeCunn in 1998 to propose a neural network (LeNet) able to perform accurate character recognition [Lecun et al., 1998]. It was discovered that fully connected layers were not always required to perform accurate inference. Convolutional layers were introduced to summarize the data and reduce the number of connections between consecutive layers. Pooling layers were also introduced to further compress the intermediate volume of the data. LeNet was only seven layers deep while modern networks like ResNet-152 are 152 layers deep [He et al., 2015].

2.3.4 Recurrent Neural Networks

Neural networks were being utilized for time-independent data before the invention of RNNs or recurrent neural networks. RNNs are being utilized in very complex tasks like scene labeling [Pinheiro and Collobert, 2014] and speech recognition [Graves et al., 2013]. These neural networks can memorize information during the training process and can learn time series data in opposite to multi-layer perceptrons and convolutional neural networks. LSTMs or, long shot term memory networks [Hochreiter and Schmidhuber, 1997], are good examples of neural networks that can

analyze time-series data. In recurrent neural networks, the same neuron's output is fed to itself in every iteration step. This strategy allows the network to memorize. Backpropagation is also performed over time through unrolling the network in time and considering each time-step as a different layer to be propagated backwards.

2.4 Parallel Deep Learning

The success of neural networks like LeNet, as mentioned earlier, motivated researchers to explore new and more complex neural network structures. The size of the training data sets and data set complexity also increased. The MNIST character recognition data set contains 70000 28x28 black-and-white hand-written digits and ten output classes while the Imagenet data set contains more than 14 million dynamic sized colored images and 1000 different output classes. Neural networks trained on MNIST data set are used to recognize hand-written digits while models trained on the ImageNet data set are used to do detailed object recognition. This simple example showcases how the difficulty of the task may indirectly affect the corresponding training data set and neural network size and complexity. This necessary growth in size negatively affects the training time of the network, especially in more sophisticated and large networks.

Parallel deep learning is the field that investigates fast and memory efficient deep neural network training strategies by utilizing high performance and parallel computing techniques. The complexity of deep neural networks make the optimal partitioning and scheduling of the operations inside the same network very difficult to automate. In literature, there are three known approaches to tackle this difficult problem: data, model and pipeline parallel training.

2.4.1 Data Parallelism

Data parallelism in parallel computing is to divide the input data into smaller partitions and run the same operation on the smaller pieces and combine the partial results in the last step. Data parallel algorithms provide a wider set of solutions

for devices with multiple processors, even for problems that are inherently serial [Hillis and Steele, 1986]. Data parallelism is the most commonly adopted partitioning strategy in parallel deep learning. As we discussed earlier, deep neural networks can be very large and complex therefore partitioning the network itself may not be a trivial task. Partitioning the data instead, is only dividing it into pieces. This straight-forward parallelization strategy was first used in 2009 for deep-belief networks [Raina et al., 2009] and has been successfully adapted for other networks and deep learning frameworks since then. As we can see in Figure 2.4 (left), in data-parallel deep learning, the mini-batch of samples is divided into k partitions where k is the number of workers. Then the deep neural network is replicated on each of the k workers. The mini-batch partitions are processed by their corresponding workers in each iteration. The local gradients are averaged over the workers and the average gradient value is then used to update the network parameters. This simple technique has shown remarkable benefits speeding up the aforementioned deep belief networks up to 72 times on GPU devices compared to a dual-core CPU.

Even with these promising improvements on training time, data parallelism has its own limitations. First, the mini-batch size selection plays a crucial role in convergence and inter-worker communication volume. Employing data parallelism on a large amount of workers implies that the mini-batches are also going to be divided into smaller pieces. This eventually results in under-utilization of the workers, since the input data now is too tiny. Similarly, many small gradient messages in the aggregation phase will add a significant communication overhead that will result in very poor training performance. In fact, this communication amount can exceed 90% of the total time spent on training the network [Harlap et al., 2018]. In order to tackle this problem, the size of the mini-batch can be increased, however, this strategy also has its own problems. First of all, naive yet popular training algorithms are known to converge much harder when the mini-batch size gets too big [Goyal et al., 2017]. Second, very large mini-batch sizes will make the network unable to fit on a single device's memory and won't allow the training to take place at all. These limitations

resulted in many sub-research areas to sprout and similarly acts as one of the main motivations for our research.

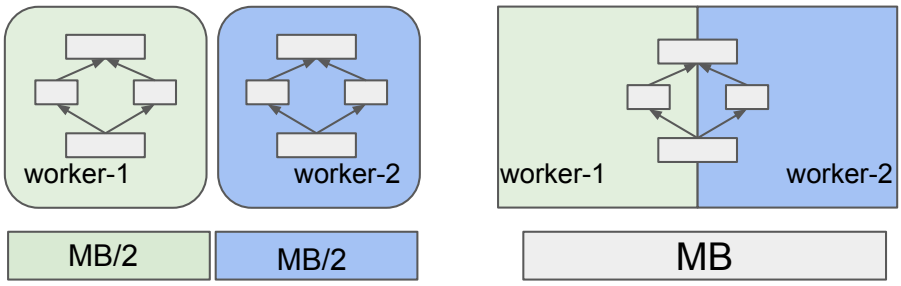

Figure 2.4: Data parallel training (left) compared to Model parallel training (right)

2.4.2 Model Parallelism

Model parallelism or task parallelism is another well-known parallel computing technique. Many popular parallel computing programming libraries such as OpenMP [Chapman et al., 2007] and OpenMPI [Gabriel et al., 2004] implement this concept. Model parallelism the process of mapping different operations to different workers considering the dependencies in between. In deep learning, model parallelism is interpreted in different ways with respect to the partitioning granularity. Whole layers or operations in the neural network can be placed to execute on different devices or a layer or operation itself can be partitioned on multiple workers. A specific version of the former interpretation is researched as pipeline parallel deep learning and will be discussed in the next subsection. The latter, was first utilized by Alex Kryznznx [Krizhevsky, 2014] to partition fully-connected layers in a deep neural network. Figure 2.4 (right) shows the high-level representation of model parallelism. The figure considers both interpretations of model parallelism; the first and the last layers of the model are both divided into half and the halves are assigned to different devices while the middle layers are executing on the workers as a whole.

2.4.3 TensorFlow

The popularity of deep learning encouraged programmers and companies to build production-level sophisticated frameworks. These frameworks help deep learning researchers by providing a practical tool to design and test new deep neural networks and training strategies. One of such frameworks that has been used intensively in the field is Google's TensorFlow. The framework was first proposed in 2010 as a large-scale deep learning framework [Dean et al., 2012]. Since the proposal of the first version in the original Google paper, major changes and additions were made to the framework. The current version of TensorFlow provides abstractions and functions for data parallel deep learning. However, automated model parallelism is not supported. If the user wants to place different operations or layers on different devices, she must instrument the neural network code manually. This process can be very challenging and require human experts to carefully plan the partitioning. With the growing size and complexity of neural networks, the process could be even more challenging. Therefore, a transparent partitioning for TensorFlow, or any other deep learning framework, has the potential of fully automating this difficult and time consuming process.

In this work, we propose a novel device placement algorithm for TensorFlow. In this section, we briefly introduce TensorFlow architecture and run time to familiarize with the terminology and the framework itself.

Overview and Architecture

TensorFlow is an extension of the DistBelief framework, improved upon user demand and requests. The improvements mostly address DistBelief's low flexibility. In TensorFlow, the users can define new mathematical operations, loss functions, training algorithms and task granulrity is mathematical operations instead of whole pre-defined layers as in DistBelief. The framework is developed in C/C++ and provides a client package in many languages including Python.

Both DistBelief and TensorFlow execute the neural networks using data-flow graphs, i.e high level representations of the operations and the dependencies in the

neural network. A data-flow graph is a directed acyclic graph (DAG). Each vertex in this DAG represents an operation in the neural network. The edges represent the data tensors communicated between these operations and also the intra-network dependencies. This data-flow graph is then either partitioned or directly executed. If the partitioning occurs, TensorFlow creates and maps an executor for each partition. Then the executors execute the sub-graphs concurrently by utilizing the dependency information provided by the graph.

Data parallelism in TensorFlow is implemented through parameter-servers. The model is first replicated on each worker device and the master. Then the parallel training algorithm, which is also configurable by the user, starts executing. The gradients in the network are averaged across all devices at the end of each iteration and the aggregation algorithm is also configurable. The aggregated and averaged gradient value is used to update the network parameters. The update rule can be configured as well. The user can choose between synchronous or asynchronous SGD or even define and use a custom update rule.

Model parallelism on the other hand, is not transparently supported. The framework allows users to place operations to different devices using the Python client. Additionally, if device placement is done by the user, the framework uses a co-location algorithm that places producer and consumer nodes on the same device to group co-locate unassigned operations in the graph with the best candidate. The placer module, however, is left open-ended in the framework code base and it is designed as an interface allowing embedding of new partitioning strategies into it.

Chapter 3

GPU KERNEL OPTIMIZATION FOR MACHINE LEARNING

As discussed in the introduction, kernel optimizations are crucially important for deep learning performance. Since deep learning is an iterative process, the same operations are invoked thousands of times and the computation latency creates a big performance problem for large number of training iterations and negatively affects the convergence time. We tried to tackle this problem by optimizing two of the most popular kernels used in deep learning: broadcast and reduction. We mathematically define the reduction and broadcast operations on tensors and their properties. We then exploit these properties to implement an efficient method for each. The details of the broadcast operation can be found in this work [Dikbayır et al., 2018], in this thesis we focus on the reduction operation. We develop a fully parallel multidimensional reduction method for tensors and implement our method with a robust work division strategy to avoid multiple kernel launches that result in extra global synchronization points and eliminate the temporary storage. We also propose an index translation algorithm to deal with the multiple dimensions of the reduction operation. Last but not least, in our evaluation, we compare the performance of our proposed implementations against Knet Deep Learning framework [Yuret, 2016], developed in Julia, an efficient programming language with just-in-time compiler and built in GPU functionalities [Bezanzon et al., 2012]. We test our proposed method involving different number of dimensions and tensor sizes and achieve up to 75% of the theoretical peak device bandwidth and significant speed-ups over the existing Knet implementation.

3.1 Mathematical Definitions

3.1.1 Reduction Operation

Reduction operation is the process of reducing one or more modes of a tensor by an associative function such as addition or maximum. The operation takes a tensor and the modes to reduce as input and produces a lower-order tensor as an output. The output tensor of the operation contains the reduced dimensions as its elements. Figure 3.1 demonstrates 1^{st}-order reductions performed on matrix $\mathbf{M}$, over its modes x and y separately by addition. If the fiber decomposition $\mathbf{M}^{\mathrm{x}}$, is taken and f_i represents the i-th fiber in $\mathbf{M}^{\mathrm{x}}$, the i-th element of the output vector is equal to $\sum_k f_i(k)$.

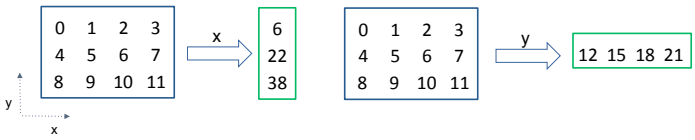

Figure 3.1: Reducing a matrix over its first mode (Left) and over its second mode (Right) by addition function

In this thesis, we use the symbol $\biguplus$ to define the reduction operation. This should, however, not be interpreted as the disjoint union operation usually used in set theory. The reduction operation is formally defined as below:

$$\biguplus_{M} \mathcal{A} = \mathcal{B}, \quad \text{where} \quad |||\mathcal{A}||| \geq n \quad , |M| = n \quad \text{and} \quad |||\mathcal{B}||| = |||\mathcal{A}||| - n \tag{3.1}$$

The modes of reduction are contained in the *mode set* M, given under the reduction operator in Equation 3.1. If a mode set is not given, the input tensor is reduced over all of its modes to a scalar value. As it is seen in the equation, the difference between the input tensor's order and number of reduction modes, or the cardinality of the mode set, is equal to the order of the output tensor.

This operation is very common in scientific computing, and some programming models such MPI and OpenMP provide primitives to perform the operation

[Makpaisit et al., 2015] [Chapman et al., 2007]. In these primitives, the input is usually reduced to a scalar value. However, in machine learning, the output of the operation can be multidimensional or the application may require a reduction over multiple modes of a tensor, making the reduction multidimensional. Neuman et al. addressed this problem in his bachelor's thesis [Neumann, 2008]. However, his implementation and evaluation is specific to Online Analytical Processing (OLAP). Moreover, many architectural and software improvements have been introduced to GPU devices since then, enabling more efficient solutions to be proposed. Therefore, in this paper, we focus more on reductions over multiple modes and reductions that result in a multidimensional output. Figure 3.2 illustrates both of these cases in a single reduction example. The figure shows sequentially, how a 2^{nd}-order reduction over modes x and z, $\biguplus_{\{x,z\}}$ is performed on a 3^{rd}-order tensor.

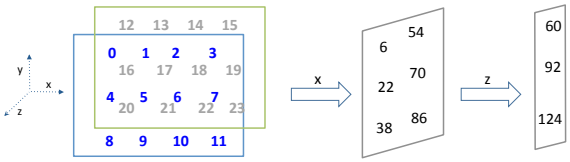

Figure 3.2: Reducing a 3^{rd}-order tensor over its x and z modes

3.1.2 Broadcast Operation

In broadcast operation, one input tensor is projected over the other by using an associative function (e.g. addition, multiplication). In the formal definition of the broadcast operation, we use the symbol $\coprod$ to notate broadcast as in Equation 3.2.

$$\mathcal{B} \coprod_{M} \mathcal{A} = \mathcal{C} \tag{3.2}$$

Different than the reduction operation, broadcast operation takes two tensors as arguments, the $\mathcal{B}$ tensor is broadcasted on the $\mathcal{A}$ tensor over modes in the mode set M and the output tensor $\mathcal{C}$ is produced. We refer to $\mathcal{B}$ as the broadcasted tensor and

$\mathcal{A}$ as the input tensor. Output tensor always has the same or larger order than both of the input tensors. In machine learning, usually tensor broadcast is utilized rather than scalar broadcast. Figure 3.3 demonstrates a broadcast example with a vector projected over a matrix, by addition. In this example, the vector $\boldsymbol{y}$ is replicated to match the order of matrix $\boldsymbol{X}$ and the elements are added to obtain matrix $\boldsymbol{Z}$. Both of the tensors, also multiple modes from each tensor can be broadcast. Figure 3.3 on the right shows the addition of a row and a column vector. It demonstrates the case where both of the vectors need to be scaled in their respective missing modes.

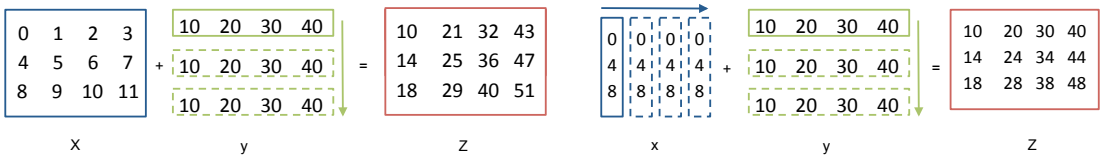

Figure 3.3: Broadcasting vector $\boldsymbol{y}$ on matrix $\boldsymbol{X}$ by addition operation (Left). Broadcasting a row over a column vector by addition operation (Right)

3.2 Reduction Implementation

An n-th order reduction operation on a tensor, can be decomposed into n different r_i-th order reductions where $\sum_{i=0} r_i = n$. For example an n-th order reduction can be written as n different first order reductions:

$$\biguplus_{M} \mathcal{A} = \biguplus_{\{m_1\}} \biguplus_{\{m_2\}} ... \biguplus_{\{m_n\}} \mathcal{A} \quad \text{where} \quad M = \{m_1, m_2, ..., m_n\} \tag{3.3}$$

In addition to the property shown in Equation 3.3, elements in a tensor are processed by an associative function, where the order of processing is not important. Therefore, if we want to reduce a higher-order tensor over n different modes, the processing order of the elements in these n modes does not affect the result of the operation. Then, the reduction operation in Equation 3.3 could also be expressed as below, where $\setminus$ represents set difference:

$$\biguplus_{M} \mathcal{A} = \biguplus_{M\setminus\{m_i\}} \biguplus_{\{m_i\}} \mathcal{A}, \quad \text{where} \quad 1 \leq i \leq n \tag{3.4}$$

Figure 3.4, demonstrates $\biguplus_{\{x,y\}} A$, performed as $\biguplus_{\{y\}} \biguplus_{\{x\}} A$ and $\biguplus_{\{x\}} \biguplus_{\{y\}} A$ respectively. As it can be seen from the figure, the order of the reduction sequence does not affect the result.

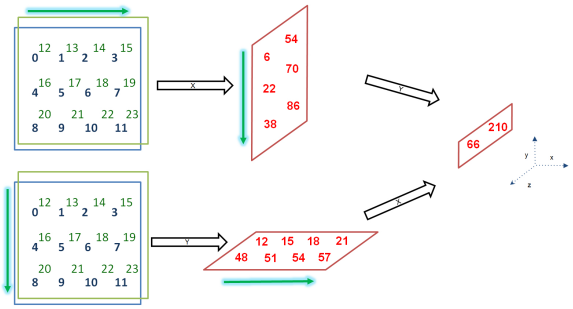

Figure 3.4: A tensor is reduced over its first and second modes in two alternative orderings.

The associativity property of reduction gives the freedom to perform the reduction in any order. However, each mode reduction in Equation 3.4 can start upon the completion of the previous mode reduction and synchronization between successive reductions is necessary. Unfortunately, CUDA does not support a global synchronization method that provides inter-thread block synchronization in a single kernel. This results in separate kernel launches for each mode reduction. This repetitive kernel launch approach causes poor utilization of memory bandwidth because all the threads are globally synchronized at the end of each kernel execution. In addition, kernel launch overheads are added to the overall execution time. To overcome these performance issues, we exploit the reduction operation's associativity property and combine the elements over each reduction mode in n-th order tensors. To obtain these hyperslices, we use the tensor decomposition described in Section 2.1. We first obtain

the decomposition $D = \mathcal{A}^{r_1, r_2, \ldots, r_n}$, consisting of hyperslices d_i. Then, each element b_i in the output tensor $\mathcal{B}$ is:

$$b_i = \biguplus d_i \tag{3.5}$$

As Equation 3.5 suggests, each hyperslice is reduced to a scalar value, thus, does not have any dependency on the reduction modes. A synchronization is not needed anymore in each mode reduction, because all of the elements in a hyperslice are simply added together. Moreover, hyperslices d_i are independent from each other, since the values of elements in $\mathcal{B}$ do not depend on each other. As a result, global synchronization is not needed either. This approach provides two benefits: temporary values are not written back to memory and overhead of multiple kernel launches is omitted because we can now perform the reduction with a single kernel launch. In the next subsections, we describe how single kernel reduction operation is implemented in detail.

Kernel Configuration

One of the challenges in obtaining high performance on GPUs, is to utilize CUDA resources properly. The number of threads per CUDA block and the total number of available blocks in a GPU are limited. The work division of the thread blocks must be carefully organized in order to achieve high performance. After giving a detailed formal explanation of our method in the previous section, we now explain our kernel configuration strategy on a 2^{nd}-order reduction that results in a multidimensional output for the sake of simplicity. In a2^{nd}-order reduction operation, the decomposition of the tensor over the reduction modes would result in slices. There are several ways to assign these slices to CUDA thread blocks. These can be categorized based on the number of slices per block.

- # of slices per block > 1 : Requires within a block synchronization since a block can compute the next slice once it finishes the current slice.

- # of slices per block = 1 : Incurs no synchronization problem.
- # of slices per block < 1 : Leads to global synchronization problem since a slice is computed by more than one thread block.

The optimal way to assign thread blocks is one block per slice as it does not require any synchronization between thread blocks. However, two problems may arise with the one-slice-per-block assignment. First, when reduction slices are very small in size but large in total count, assigning multiple slices to a single CUDA thread block might be required. In this case, the number of maximum thread blocks that CUDA supports is fewer than the number of independent reduction slices in the decomposition. We overcome this issue with *grid stride loops*, where a grid stride is the number of thread blocks in the execution. In our grid stride loop, each thread block is assigned a single slice and the blocks are reassigned to the remaining slices once they finish reducing a slice. The next slice that a thread block computes is a grid stride away from the current slice.

The second problem is related to the slice size being too big. A CUDA block can contain 1024 threads at maximum. If the size of the reduction slice is too big, a single block may be insufficient to reduce a slice. Assigning multiple thread blocks to a single slice, would require synchronization between thread blocks, thus multiple kernel launches. To avoid this, we increase the number of elements to be computed by a single thread in a block. For example, if we have 1024 threads in a CUDA block and assign four elements to each thread in the block, we can reduce slices containing up to 4096 elements, without assigning an additional block to the same slice. The efficiency will naturally decrease in case of corner cases where we might have to assign too many elements to each thread, but the parallelism will not be affected.

Lastly, if the tensor is skinny and tall meaning that the slices are big but there are only few of them, our implementation would only create a small number of thread blocks, which may not be sufficient to occupy all the streaming multiprocessors. We do not have a good solution for this yet. Either one can divide the slice among

multiple thread blocks, which introduces synchronization problem, or launch parallel kernels on the device. We leave this for future work.

Virtual Coordinate Calculation

Algorithm 1: CalcCoord: Calculate Virtual Coordinates

Input : $ModeSizes$: *Sizes of reduction modes*
$HyperSliceCount$: *Number of hyperslices in decomposition*
$ThreadID$: *One dimensional global index of the thread*

Output: $Coordinates$: *Virtual coordinates*

```
1 Coordinates[0] = ThreadID % ModeSizes[0]
2 PrevModesArea = ModeSizes[0]
3 for i = 1 .. ModeSizes.length − 1 do
4 |   Coordinates[i] = (ThreadID/PrevModeArea)%ModeSizes[i]
5 |   PrevModesArea = PrevModesArea * ModeSizes[i]
6 end
7 Coordinates[ModeSizes.length] = ThreadID/PrevModesArea
```

In this section, we explain how we assign the CUDA threads to the correct tensor elements. As described in Section 2.1, in our method, a reduction over multiple modes is performed on the hyperslices of the input tensor's decomposition. The decomposition results in a virtual re-arrangement of the original tensor in a different multidimensional space that has the reduction modes as its dimensions. We do not rearrange the elements in the memory physically, therefore we cannot use the original tensor's coordinate values to access them. Instead we need to transform the linear indices to virtual coordinates.

If there are n modes to reduce the original tensor over, then the reduction space is $n+1$ dimensional. We first derive an $n+1$ dimensional coordinate system for the reduction space. Then the first n coordinates determine the target element's position in the n-th order hyperslice, while the last coordinate indicates which hyperslice the target element belongs to. We can easily access an element once its reduction space coordinates are known. However, in the actual parallel implementation, we are only

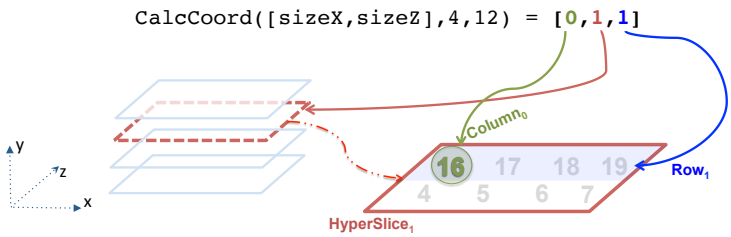

Figure 3.5: Virtual coordinates of thread 12 visualized on the tensor used in the example coordinate calculation.

given the one dimensional global index of each consecutive thread ID in a CUDA kernel. Using this global index value, we need to calculate the coordinate values in the $n+1$ dimensional reduction space in order to assign threads to the correct tensor elements. Algorithm 1 demonstrates the steps required to derive the coordinate values of the target element. The aim of the algorithm is to determine to which positions a linear index corresponds in a multidimensional space. The algorithm takes three inputs: $ModeSizes$, $HyperSliceCount$, and $ThreadID$ and produces a single output $Coordinates$. $ModeSizes$ is the list containing the size of each reduction mode, $HyperSliceCount$ is the number of hyperslices in the decomposition which is also equivalent to the size of the last dimension in the reduction space, and $ThreadID$ which is the one dimensional global index of the CUDA thread. The virtual coordinate of the target elements is then stored in $Coordinates$.

For the sake of simplicity we explain the algorithm using an example. Let's suppose that we are reducing the tensor $\mathcal{T}$ from Figure 3.2. As it is shown in the figure, we are reducing a 3^{rd}-order tensor over x and z modes. The decomposition will result in a reduction space containing three xz slices, each containing two fibers of size four. So,

$$ModeSizes = [4, 2] \quad and \quad HyperSliceCount = 3$$

Now let's find the virtual coordinate values for the thread with $ThreadID$ equal to 12. First we calculate the linear offset of the thread in a fiber as it is shown in line

1 in the algorithm: $12\%4 = 0$. Then, we initialize $PrevModesArea$ as $ModeSizes[0]$, which is 4 in this case. In line 4, two operations are combined. For the first iteration, the division operator finds the number of fibers covered by $ThreadID$, $12/4 = 3$. Then, the modulus operator calculates the offset of the linear index in the second mode, $3\%2 = 1$. In the last line of the loop, the area of the previous modes is updated with the second mode's size. This procedure is repeated for all the remaining reduction modes, and all the coordinates within the hyperslice are calculated. In this example there are only two reduction modes so the hyperslice coordinates are $[0, 1]$. Finally, we calculate to which hyperslice the thread must be assigned by finding how many hyperslices the linear index covers, $12/8 = 1$. This means that the thread will process the first element in the second row of the second hyperslice in the decomposition. After the process described above, we calculate the memory index of the element using the virtual coordinate values and the memory stride values.

Parallel Reduction Algorithm

In order to achieve good performance, it is necessary to optimize the reduction performed by each thread block. Harris et al. proposed a detailed strategy for efficient parallel reduction to a scalar value [Harris, 2007]. In their method, each thread block processes a partition of the input tensor, and the partial sums are then processed by a separate kernel or by CPU to obtain the scalar result. To calculate the thread block sums, they leveraged the on-chip memory, called *shared memory*, which is accessible by all the CUDA threads in the same thread block. Each thread in a block sums the elements assigned to it and then writes the result to a shared memory slot. Then, after the threads in the block are synchronized, the sums in the block's shared memory are added together to obtain the block sum. This approach lets the program to avoid accessing the device memory, which is a costly operation. However, in this method, block synchronization is needed before combining the partial sum in shared memory. This implementation can be further improved by utilizing the properties of structures called *warps*. Warps contain a certain number of threads that are all

Algorithm 2: Parallel Reduction Algorithm

Input : $\mathcal{A}$:Input tensor

$MemStrides$: Memory stride values

$ModeSizes$: The size of each reduction mode

$HyperSliceCount$: The number of hyperslices in decomposition

$NoEls$: Number of elements processed by each thread

Output: $\mathcal{B}$: Output tensor

```
1  Coordinates = CalcCoord(ModeSizes, HyperSliceCount, ThreadID)
2  MemIndex = Coordinates .* MemStrides
3  NumSlices = CalcSliceCount(BlockID)
4  for Slice = 0 .. NumSlices -1 do
5      for Iter = 0 .. NoEls - 1 do
6          Sum = Sum + A[MemIndex + Iter*Skip]
7      end
8      Sum = BlockReduce(Sum)
9      if first thread
10         B[ Slice*GridSize + BlockID] = Sum
11     end
12 end
```

scheduled together at the same time, thus threads in the same warp do not need to be synchronized. In addition to this, NVIDIA introduced a new instruction called *shuffle* with the Kepler architecture that allows a thread to read data from the local register of another thread in the same warp [shu, 2017]. This means that threads in the same warp can communicate with each other without any shared memory access or synchronization calls. This also implies that the synchronization calls and shared memory accesses are reduced by a factor of warp size, which is 32 for Kepler.

We adopt the block reduction method proposed by Luitjens et al. [Luitijens, 2014], that exploits the warp properties and leverages the shuffle instruction. In Luitjens' block reduction, each warp in the block performs a partial reduction using the shuffle

instruction. Then, the first thread in each warp writes the partial sum into a shared memory slot and waits for the other warps to perform their partial sums. Finally, all the partial sums are summed together by the first warp with a single warp reduction.

Now that we have an efficient block reduction method, we can construct the overall reduction kernel by incorporating the strategies from the previous sections along with the block reduction method explained in this section. Algorithm 2 demonstrates our proposed parallel reduction method in a pseudocode. Note that many details are omitted to clarify the code. We begin by calculating the virtual coordinate values and take their dot product with the memory stride values to find the memory index of the element. Then, the thread sums the elements assigned to it and passes it to the *BlockReduce* function which implements the block-level reduction with warps and shuffle instructions as explained earlier in this section. After the block sum is calculated, we write the result to the output tensor in global memory. If the thread block is assigned to more than one slice to compute, it advances to the next slice using the grid stride.

Chapter 4

TRANSPARENT MODEL PARALLELISM FOR DEEP LEARNING

4.1 Problem Definition

The importance of time and memory efficiency in deep neural network training was highlighted multiple times in the previous chapters. In this chapter, we explore different ways to partition the operations of a deep neural network across different devices. We use the TensorFlow framework to test our strategies. As we explained earlier, TensorFlow trains neural networks using data-flow graph representations. All the scheduling, partitioning, kernel launches are done using this graph. The TF data-flow graphs are fine granular, therefore the graphs can contain tens of thousands of nodes. For example, the Inception-v3 graph contains around 37 thousand nodes in its data-flow graph. Partitioning such large graphs manually is impractical, coarsening the graph is possible but the coarsening pattern must also be specific to network to exploit maximum parallelism. This problem is very similar to the well known NP-hard multiprocessor scheduling problem [Manne, 1959]. In this problem, tasks in a data-flow graph are being scheduled to different processors in such a way so that the overall execution time is minimized. Our problem is very similar, yet more complicated. The most important difference is the communication factor. Communication overhead plays an important role in overall performance. A variant of the MSP is the multiprocessor scheduling problem with communication delays. Davidović et al. proposed a mathematical programming solution for the problem [ć07towardsthe], however the task sizes are fixed and edge costs are static in their case.

We formulate the problem as a dynamic cost function:

Let $V = v_1, v_2, v_3, ..., v_N$ be the set of vertices in the data-flow graph. $t_{communication}$ is the function that computes the communication time, given the communicated data size, the source and the target devices. $t_{computation}$, on the orher hand, is the time spent to calculate the operation on the target device, given the input data size. $in(v_i)$ and $out(v_i)$ denote the size of the input and output tensors.
The total cost of scheduling the vertex set V is $C(V)$ and defined as:

$$C(V) =_{min_{v_i \epsilon V, G_j}} [C(V - \{v_i\}) + c(v_i, V - \{v_i\}, D_j)]$$

whereas the incremental cost of scheduling a single vertex v_i along with a vertex subset U on device D_j is $c(v_i, U, D_j)$ and defined as:

$$\begin{aligned} c(v_i, U, D_j) = & \; t_{computation}(v_i, D_j, in(v_i)) \\ & + \; t_{communication}(in(v_i), D_{pred(v_i).device, D_j}) \\ & + \; t_{communication}(out(v_i), D_{succ(v_i).device, D_j}) \end{aligned}$$

As we can see from the above equation, the cost of communicating data between two nodes in the data-flow graph depends on the devices that the operations are placed. This dynamic edge cost makes finding an optimal solution very difficult since the number of combinations to try to reach an optimal solution increases exponentially as the number of edges in the data-flow graph increases.

Moreover, the overhead of partitioning the data-flow graph should be small so the training phase is not delayed. In other words, the time required to generate a placement decision should be significantly lower than the time required to generate the decision. In order to tackle the problems defined above, we propose a heuristic, DFS based device placement algorithm that runs in polynomial time. In this chapter, we present our placement algorithm and our system overview.

4.2 Device Placement

Arguably, the most critical component in the proposed system is the preferred device placement algorithm. The system architecture proposed in Figure 4.2 allows any

placement or partitioning algorithm that runs on weighted graphs to be deployed. This modular structure allow us to utilize other graph partitioning tools and experiment on new algorithms with ease.

4.2.1 Depth-first Placement

The graph partitioning tools and algorithms we evaluated often gave very poor partitioning results, often ignoring the geometry of the graph, the edge directions and performing only load balancing. We try to tackle this difficult problem by designing a specifically tailored device placement algorithm, depth-first placement, for weighted DAGs. This algorithm is based on the popular depth-first search algorithm. We traverse the data-flow graph as we traverse a regular graph with DFS. But instead of only traversing, we also make placement decisions whenever local parallelism is spotted.

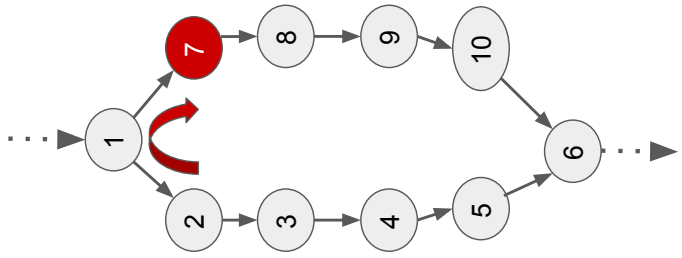

Figure 4.1: A turn-point node (in red). The labels on the nodes denote the DFS traversal order. The dotted arrows represent the rest of the network.

DFS traverses the nodes in a graph in a "downwards" fashion, meaning we first go as deep as we can from the root node and then return back to its next child node. This next node is also the sibling of the visited first child node, i.e it has the potential to be placed on a different device. These special points are when the device placement decision is done by our algorithm and we refer to them as *turn-*

point nodes. In non-turn-point nodes, the operation is simply placed on its parent's device. By adopting this idea, we will only make a placement decision when there is parallelism since assigning consecutive nodes to different devices may introduce unnecessary communication overhead. Figure 4.1 shows an example turn-point node in a graph for clarification. Algorithm 3 shows the pseudo code for the traversal logic. The algorithm is a variant of the original DFS algorithm with a two differences: we call the *assign_device* function (line 16) at each turn-point node; we introduce a global stack object *retraction_stack* to keep the least recently assigned paths in the iteration memory.

Algorithm 3: DFP

Input : G : Weighted data-flow graph
parent_node : Parent node
current_node: Current node
is_turn_point: True iff the current node is a turn-point node

Output: $\mathcal{B}$: Device Placement File

```
1  if current_node == ROOT
2  |   current_node.device = 0
3  else
4  |   current_node.device = assign_device(G, current_node, parent_node)
5  end
6  for EVERY child in current_node.children do
7  |   if child.visited == False
8  |   |   DFP(G, current_node, child, (child != current_node.children[0]))
9  |   end
10 end
11 current_node.visited = True
12 retraction_stack.append(current_node)
```

Algorithm 4: assign_device

```
Input  : G : Weighted data-flow graph
         parent_node : Parent node
         current_node: Current node
Output: dev: Assigned Device
seen_path, cur_path = list()
intersection_found = False
current_path.append(current_node)
inters_candidate = current_node.last_visited_child()
while intersection_candidate == NULL
    unplaced_successor = current_node.first_unvis_child()
    current_path.append(unplaced_successor)
    inters_candidate = unplaced_successor.last_visited_child
end
while intersection_found == False
    if inters_candidate IN retraction_stack
        intersection_node = inters_candidate
        intersection_found = True
        popped = retraction_stack.pop()
        while popped != inters_candidate
            seen_path.append(popped)
            popped = retraction_stack.pop()
        end
        retraction_stack.append(popped)
        i = current_path.length - 1
        while i ≥ 0
            if current_path[i].visited == True
                retraction_stack.append(current_path[i])
            end
            i = i - 1
        end
    else
        current_path.append(inters_candidate)
        inters_candidate = inters_candidate.last_visited_child()
    end
end
cand_devices = list()
cand_devices.append_mult(parent_node, intersection_node)
LU_dev = seen_path.least_used_device()
cand_devices.append(LU_dev)
return find_min_cost_dev(cand_devices, seen_path, current_path, parent_node, intersection_node)
```

Whenever we go deeper in the graph, we insert the nodes we traverse into the retraction stack. We will later utilize this data structure to effectively extract the

current and least recently seen paths. Algorithm 4 lists our device assignment function. We first initialize the current path *current_path* and least recently assigned path *seen_path* that are going to be compared. Then, we initialize the candidate search loop by setting the intersection candidate *inters_candidate* to the last visited child of the current node to determine whether our intersection candidate is an immediate child of the current node. In the loop, each time we cannot find a recently visited child of the *unplaced_successor*, we insert this node into our current path and repeat until we find a valid candidate. Now, we need to extract our least recently assigned path using retraction_stack and inters_candidate. We will pop nodes from our retraction_stack and add them into seen_path until we find inters_candidate in retraction_stack. At this point there is a subtlety. During the development, we first did not use an intersection node candidate and assumed our intersection node was always going to be in the global retraction stack. However, when we ran this version of our placement algorithm on a real-world neural network, the retraction_stack starved and caused the program to crash. We first thought this might be a corner case but it turned out it is very common in the networks we have experimented with. To address this problem, we introduced inters_candidate and a few additional steps to keep the correct seen_path in memory and avoid starvation. The approach is very simple. In Line 11, we check if our candidate is in the global retraction stack and if it actually makes sense to enter the first if statement. If the stack does not contain our candidate, we first append the current candidate to our current path and simply assign the most recently placed child of our current candidate as the new candidate. We repeat the process until our candidate is finally in the retraction stack. When it is, we start extracting our least recently used path by popping the nodes in the retraction stack until we encounter the intersection node, which was determined in the previous loop. Finally, we pop nodes from our current path into the retraction stack to update the least recently assigned path of nodes. With this technique, we generalize our algorithm and successfully execute the algorithm on neural network data-flow graphs.

The previous function 3 and majority of function 4 form the skeleton of the algorithm. Now we have a parallelism-aware graph traversal algorithm that tries to encompass local parallel patterns in the graph. Now the problem is to make a cost analysis and comparison using the current and least recently used paths collected in the previous step. In the cost comparison phase, we will calculate an overall decision cost of assigning a turn-point node into a specific device. We will try multiple combinations and pick the least costly option as our placement. We first use some intuitions to form a candidate device list. We only consider three devices in our device selection policy: parent node's, intersection node's and the least used device in the least recently used path. The reasoning behind selecting the parent and intersection devices as candidates is because of the dynamic communication cost behaviour at these points, which will be explained in detail in the next paragraph. The reason why we also consider the least frequently used device in the LRAP is because we want to exploit as much parallelism as we can since the chance of waiting for the least frequently used device are the lowest.

It is also important to note that CPU-GPU data transfers are much slower than the GPU-GPU ones. Especially if the GPU devices are inter-connected with NVLink [Foley and Danskin, 2017] or similar technologies the difference can be very large. This makes the assignment problem dynamic, since we do not have a static communication cost between the nodes. All the edge costs however, are not dynamic. For example, the unvisited nodes consecutive nodes in the current path are all placed on the same device, therefore the cost does not depend on the placement decision of our turn-point node. Instead, we need to look at the boundaries of our current path. This means we only experience dynamic cost behaviour in the first and terminal nodes of our current path: the turn-point node and the parent of the intersection node. We use different bandwidth values for different devices and use the tensor size information to estimate the data transfer cost between two possible nodes. We then use these values to correctly add the boundary communication costs to our overall decision cost. We use a heuristic overall decision cost to minimize the idleness among the devices. The

remaining part of the decision cost is calculated by adding all the operation costs together of least recently used path's nodes that are serial to the current path. In other words, we aggregate all the node costs in the least recently assigned path if they are placed on the same device we considering putting our current turn point node.

DFP is an iterative algorithm. Therefore, the iterative complexity of DFP is the same as DFS. At turn-point nodes, additional computation is done in DFP. However, this step does not include any calculations that have an exponential time complexity. The candidate device set size is constant since we always pick the parent, intersection and least frequently used device as candidates for placement. Therefore, this extra step with constant time complexity allows the algorithm to execute in polynomial time, which makes it very convenient to use.

In summary, we propose a novel graph traversal and device placement algorithm for DAG represented task workflows. Our evaluation is done on deep neural networks and TensorFlow, however, one can easily adapt our proposed solution in another scientific workflow that has similar properties to TensorFlow data-flow graphs.

4.3 Proposed System Overview

TensorFlow has two different run-time modules: common and distributed. The common run-time is for neural networks trained in a single system and not distributed across a cluster of computers. This system may still have access to multiple accelerators, but they are all in the same rack and not in a cluster connected through a network. In this thesis, we will focus on the common run-time, but our strategy should also be easily applicable for the distributed version. In the common run-time, the user first defines her target network using the operations and functions provided by the TF framework's client library, as discussed in Chapter 2.4.3. Then, the framework generates a data-flow graph for the neural network and assigns an executor object to run it. All the internal scheduling and memory management is handled transparently by TF. The high level blueprint of the proposed system is shown in Figure 4.2. We propose two additional modules to be integrated into the TF run time: an offline pro-

filer and a device placement module. The offline profiler collects the data necessary to run placement algorithms and the device placement module generates a placement decision and executes it on the data-flow graph when the network is trained.

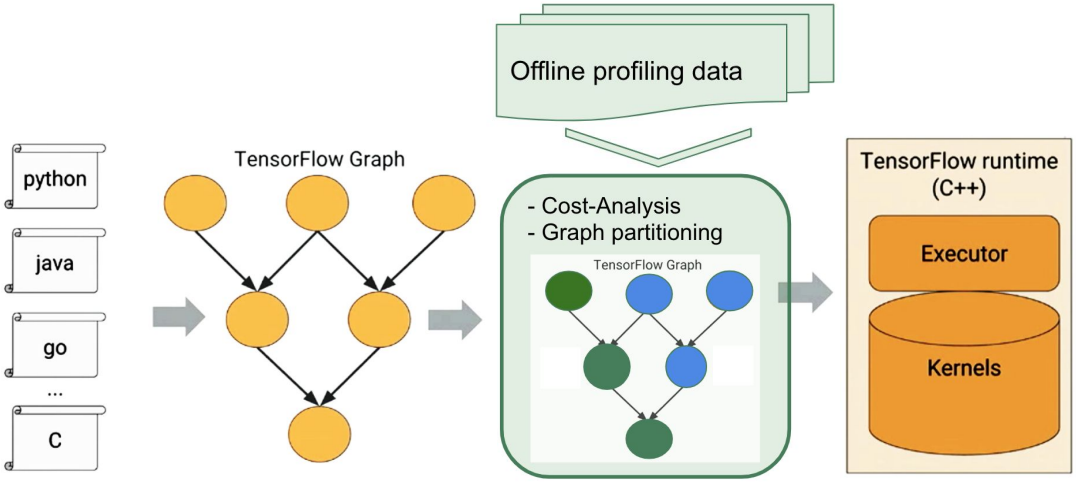

Figure 4.2: Proposed system overview. The green components in the figure are the proposed modules

4.4 Offline Profiler and Preparation of Data

Our algorithm and the graph partitioning libraries we will experiment with in this chapter, all work for weighted graphs. Unfortunately, TensorFlow's data-flow graphs only contain logical information for resolving dependencies and visualization purposes. This requires us to first assign weights to TF's data-flow graph. In order to achieve this, an offline profiling phase is required. The profiling is offline because it happens for a few iterations before actual training begins. We use TensorFlow's profiling API to instrument the model code and run the target network under this setup for a certain number of iterations. The TF profiler module generates JSON files that contain running time statistics for all of the operations in the data-flow graph. Similarly, we also use the profiler to export edge weights, size of the communicated tensors, to a separate file. Finally, we extract the neural network into a dot file [Gansner et al., 2006] to easily process it in the next phase with the proposed algorithm.

After node and edge weight profiling is completed, the data must be loaded into memory for our algorithm to process. First we read the dot file into an adjacency list. Then, we initialize a graph object using this list. The node weights are then populated using the values in the previously collected JSON files. Similarly, edges are populated with the tensor size values from their corresponding files. It is important to note that in TF, every node in the data-flow graph maps to a mathematical operation and its kernel for a specific type of input and device. Every operation defined in the framework have one or more kernels for different scenarios. For example an operation can both have a float32 GPU kernel and a float64 CPU kernel. Naturally, the performance of these operations is affected by the input and device types. Because of this, we collect data for each of the different kernels during the profiling phase and store them in different files. We load the performance data from these files into node weight vectors. Similarly, the edges have weight vectors. In their case, the vectors contain two elements: first one denoting GPU-CPU transfer time and the second denoting the GPU-GPU transfer time. The transfer times are calculated by dividing the tensor size values to the reported bandwidth rates for the device and connection type. After all of the data preparation is done, the weighted DAG is ready to be processed by our algorithm.

Chapter 5

EVALUATION

In this section, we will present our experiment results for the multi-dimensional reduction kernel and automated model parallelism algorithm.

5.1 Multi-dimensional Reduction Operation

5.2 Evaluation

In this section, we discuss the experiments we have performed to measure the performance of our implementations against Knet and how the performance is affected by different parameters. In our experiments, we measure the effective bandwidth rate achieved by our implementations because both reduction and broadcast operations are known to be memory bandwidth limited and have very low arithmetic intensity [Harris, 2007]. We calculate the effective bandwidth by dividing required data movement by the elapsed time. Here the required data movement is the minimum amount of read and write operations required to perform broadcast or reduction. Since the required data movement is the same for both Knet and our implementations, essentially the effective bandwidth is proportional to the inverse of elapsed time.

We conduct our measurements with CUDA 9.0 on a Tesla K40m GPU accelerator. The device has 12 GB of GDDR5 on-board memory, supports PCI Express 3.0 and its peak memory bandwidth rate is 288 Gb/s [NVIDIA, 2013]. The Nvidia bandwidth test in the CUDA toolkit measures 215 GB/s on device-to-device pinned memory. The CPU on the host is a 2x Intel Xeon E5 2695 v2 with 256 GB of memory. All the experiments are done with double precision floating point arithmetic.

5.2.1 Reduction Performance

We divided our experimental setup into two parts. First, we observe how the structure of the input tensor affects the performance and in the second part we test our method with different reduction scenarios and compare the performance to Knet's reduction method. In the first part, we change the width and length of the input tensor to observe in which cases our method achieves the best performance. We perform a second order reduction on a 4^{th} order tensor, so the decomposition results in slices. We first test our algorithm on a tall and skinny 4^{th} order tensor which has high number of slices in its decomposition but only few elements in each slice. Then we increase the number of elements in each slice while keeping the total number of elements in the tensor fixed to shorten and fatten the tensor.

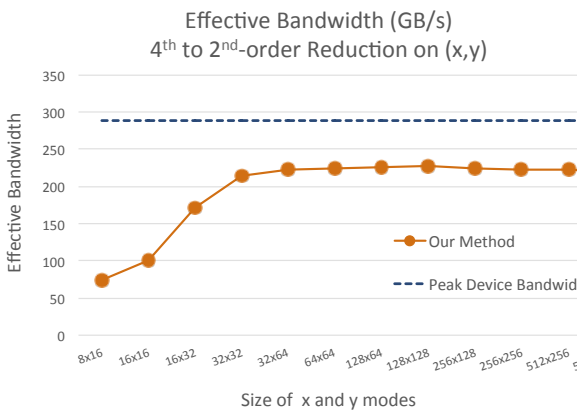

Figure 5.1: Performance of our method with increasing # elements in each slice in the decomposition

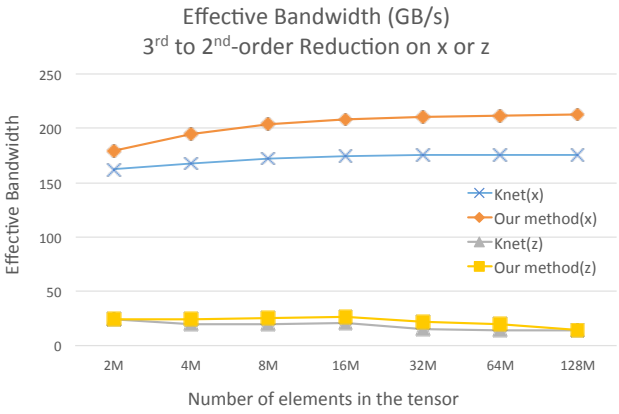

Figure 5.2: Comparing our method against Knet with different tensor sizes, reducing third order tensor over its x or z modes.

Figure 5.1 shows the effective bandwidth rate changing with the number of elements in each slice in the decomposition. The number of elements in the input tensor is fixed around 64 million. The number of elements in a slice starts at $8x16$ and goes up to $512x512$. The slice count decreases in the same range since the total number of elements in the tensor is fixed. We observe that for skinny and tall tensors the performance is low, only achieving about 20% of the peak device bandwidth. This

is mainly because each thread block has few number of threads and processes few number of elements since each block is assigned to a single slice, which results in poor usage of stream multiprocessors. However, we observe that almost no performance degradation with the increase in slice size (decrease in slice count) since there are enough slices for all thread blocks. As expected this experiment suggests that our method works best when the slices in the decomposition are fat enough to keep the stream multiprocessors occupied.

In the second part of our experiments, we measure how the order of the reduction and choice of reduction modes affect the performance. To measure these effects, we select the first modes and last modes in a tensor; the former represents the continuous memory accesses in the fast changing dimensions and the latter represents the largest strided memory accesses in the slowest changing dimensions. As baseline, we use Knet's reduction implementation. We keep the minimum size of a hyperslice at 4096 elements to ensure that the stream multiprocessors are not underutilized. We increase the total number of elements in the input tensor from $2M$ up to $128M$. We measure reduction performance on tensors $\mathcal{A}$, $\mathcal{B}$ and $\mathcal{C}$, where we reduce 1, 2 and then 3 modes, respectively. $\mathcal{A}$ is a 3^{rd} order tensor and we perform $\biguplus_{\{x\}} \mathcal{A}$ and $\biguplus_{\{z\}} \mathcal{A}$. Then, we perform $\biguplus_{\{x,y\}} \mathcal{B}$ and $\biguplus_{\{y,z\}} \mathcal{B}$ on the 4^{th} order tensor $\mathcal{B}$ and finally we perform $\biguplus_{\{x,y,z\}} \mathcal{C}$ and $\biguplus_{\{z,t,k\}} \mathcal{C}$ on the 5^{th} order tensor $\mathcal{C}$.

Figure 5.2 shows the effective bandwidth of reductions on tensor $\mathcal{A}$. We provide very similar performance to Knet when reduction is performed on a single mode because there is only a single mode to reduce and not much additional benefit of our approach that combines multiple modes into a single kernel. The performance drastically decreases for both our method and Knet when the tensor is reduced over its z mode. This is expected because elements over the z mode of the tensor are far from each other in the memory thus require strided accesses by the threads.

Figure 5.3 and Figure 5.4 show the reduction on tensor $\mathcal{B}$ with 2 mode reduction and tensor $\mathcal{C}$ with 3 mode reduction, respectively. First of all, similar to single mode reduction, we clearly observe that strided memory accesses affect the perfor-

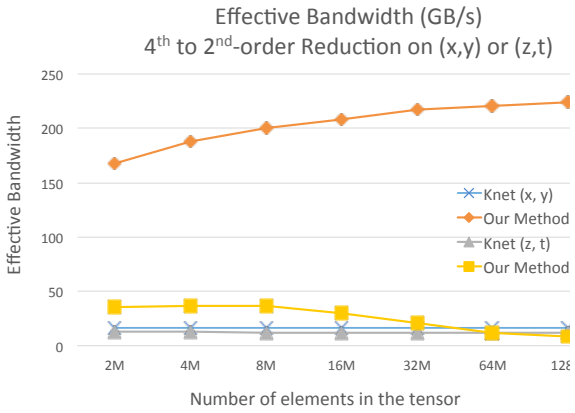

Figure 5.3: Reducing fourth order tensor over its (x, y) or (z, t) modes

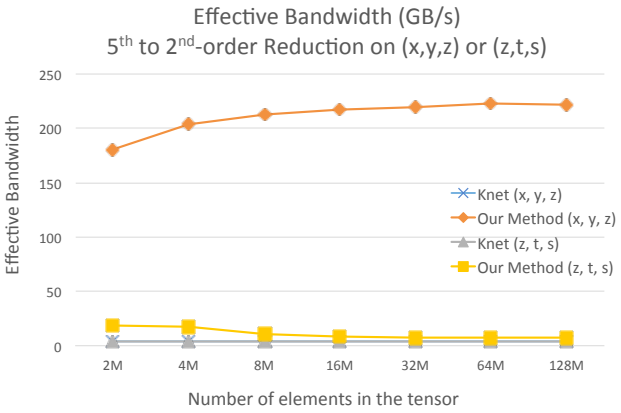

Figure 5.4: Reducing fifth order tensor over its (x, y, z) or (z, t, s) modes

mance of all the implementations negatively. However, different from Figure 5.2, we achieve a significant speedup over Knet when the tensors are reduced over their first modes. This indicates that while an iterative implementation is critically affected by the number of modes in the reduction, our approach enables to perform multidimensional reduction operations without losing performance. Our implementation is mainly bounded by strided accesses to the memory and not affected by the number of reduction modes. Even though it is not shown in the figure for clarify, reduction modes in between the fastest and slowest changing dimensions result in an effective bandwidth somewhere in highest and lowest bandwidth rates.

5.3 Transparent Model Parallelism

In this section, we will evaluate the proposed device placement algorithm on different neural networks. We will also compare our preliminary performance results and placement patterns with the ones produced by graph partitioning tools METIS and Zoltan. In our experiments, we use the NVIDIA DGX-2 workstation. The node is equipped with four NVIDIA Volta 32GB GPU devices. The GPUs are inter-connected with each other through NVLink 2.0 high-speed connection. The main neural network we have experimented with is Inception-v3. Due to its large size, high accuracy and wide

structure, this network is a very good candidate to perform experiments. As we mentioned earlier, Inception-v3 is a very successful convolutional neural network designed to perform image-classification. In order to reduce the memory requirements of the network, the convolutions summarize much more data but multiple convolutions are also performed at the same layer to prevent information loss. This concurrent pattern of this neural network allows us to analyze the placement patterns more easily.

5.3.1 Graph Partitioning Algorithms

In order to form a baseline, we use well known graph partitioning tools METIS [Karypis and Kumar, 1995] and Zoltan [Devine et al., 2000]. These graph partitioning and load balancing tools implement many different well-performing graph partitioning algorithms. We will use the multilevel k-way partitioning , multi-constraint multilevel recursive bisection , Zoltan multilevel partitioning and Zoltan hypergraph partitioning [Karypis and Kumar, 1996][Karypis and Kumar, 1998] [Devine et al., 2002] [Devine et al., 2006]. The algorithms all use coarsening and refinement approaches to efficiently partition the graph while achieving load balancing.

We first populate the node and edge weights using the profiling files generated by our offline profiler. Then we convert the graph dot file into the formats required by METIS and Zoltan. After this pre-processing step, we simply feed the correctly formatted data into the algorithms and collect placement files to be fed into the modified TensorFlow.

We first present the preliminary performance results for DFP and the graph partitioning algorithms. It is important to note that, however, when performing these experiments, the TensorFlow placer module was still enabled. Therefore, some placement decisions of the algorithms is replaced by the placer with a new decision. However, these results still give us intuition about the placements and when we will analyze some of the corresponding placement patterns at the same time, we can con....

The first performance experiments are done on the aforementioned Inception-v3 neural network. The vanilla version of TensorFlow is trained for six hundred iterations

with 32 as its minibatch size on a single GPU device. The same training setup will be used to collect performance data of the graph partitioning and DFP algorithms.

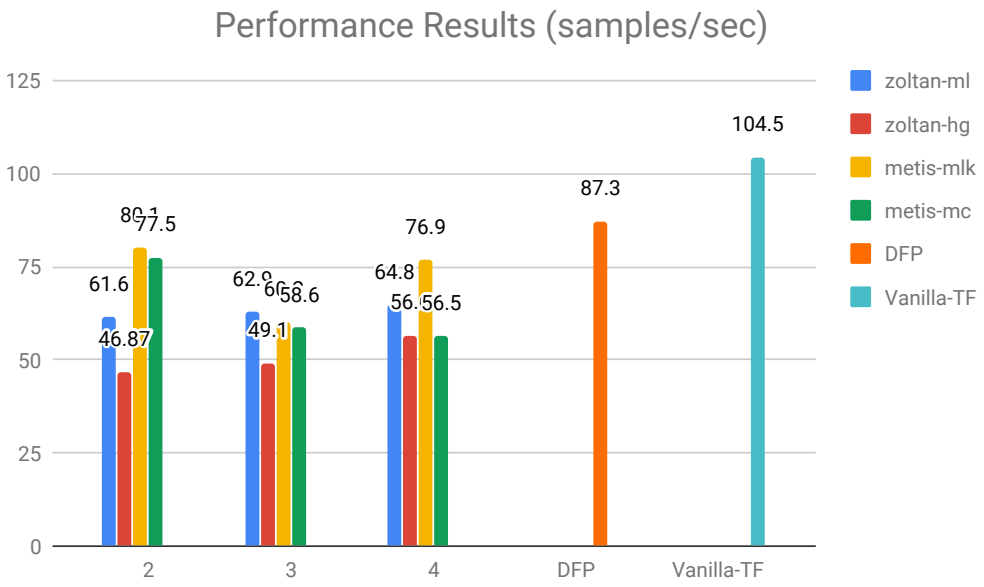

Figure 5.5: The preliminary performance results for DFP and graph partitioning algorithms for Inception-v3

If we look at Figure 5.5, we can see that DFP algorithm's placement decision outperforms all of the existing graph partitioning algorithms, but still cannot reach the performance of the Vanilla single GPU TensorFlow. The DFP placement in this experiment is almost equally partitioned across devices, however, the TF placer module makes some nodes migrate to other devices because of the colocation contraints. The same situation is also applicable for the graph partitioning algorithms, but they still perform very poorly compared to DFS. The reason behind this is probably because data-flow graphs are DAGs. These graph partitioning algorithms are originally designed for undirected graphs therefore, they do not consider the concept of edge direction. Because of this issue, the network is partitioned by only considering node and edge weights and therefore cannot exploit the parallelism between the operations.

However, Inception-v3 boasts a very large data-flow graph, so we continue ex-

perimenting with a smaller architecture: VGG-16. VGG-16 is also a well known convolutional neural network. It has a good accuracy rate, however, it's operations are memory hungry yet it has less layers and is thinner than Inception-v3. Since there is less parallelism in this model, we would not expect better results than Inception-v3. Figure 5.6 shows the performance results for METIS k-way multilevel partitioning results and DFP placement results on VGG-16 with batch-size of 128. Since the other graph partitioning methods performed significantly worse, we did not include them into Figure 5.6. Again, we can see that the DFP algorithm outperforms all the METIS placements but it still cannot beat Vanilla TF performance.

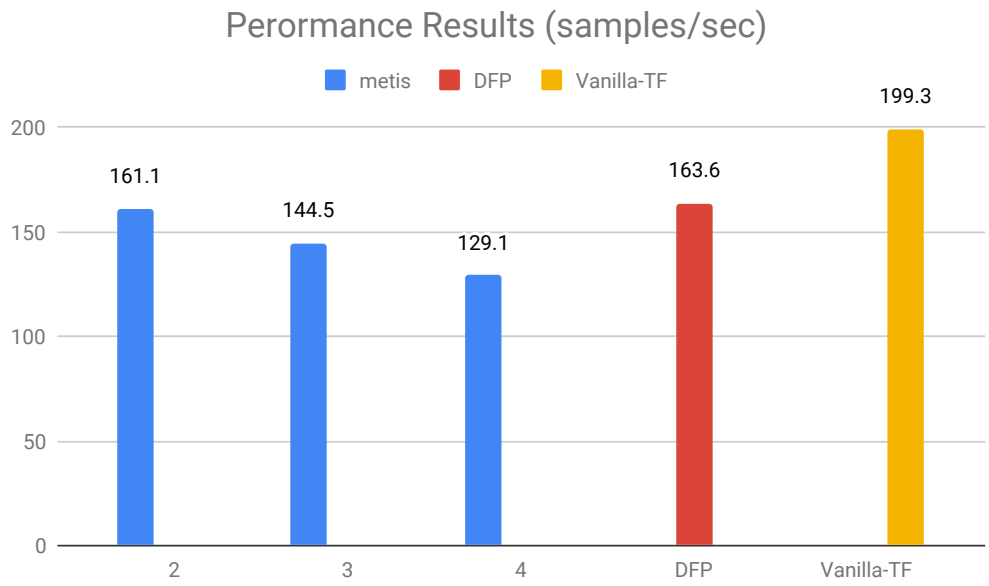

Figure 5.6: The preliminary performance results for DFP and graph partitioning algorithms for VGG-16

We first suspect underutilization of the resources. In order to test our suspicions, we perform a stress test to see if our relative performance increases when the minibatch size increases for the Inception-v3 model. We profile Inception-v3 with 64 and 128 as its minibatch size and generate new placement files to collect performance data.

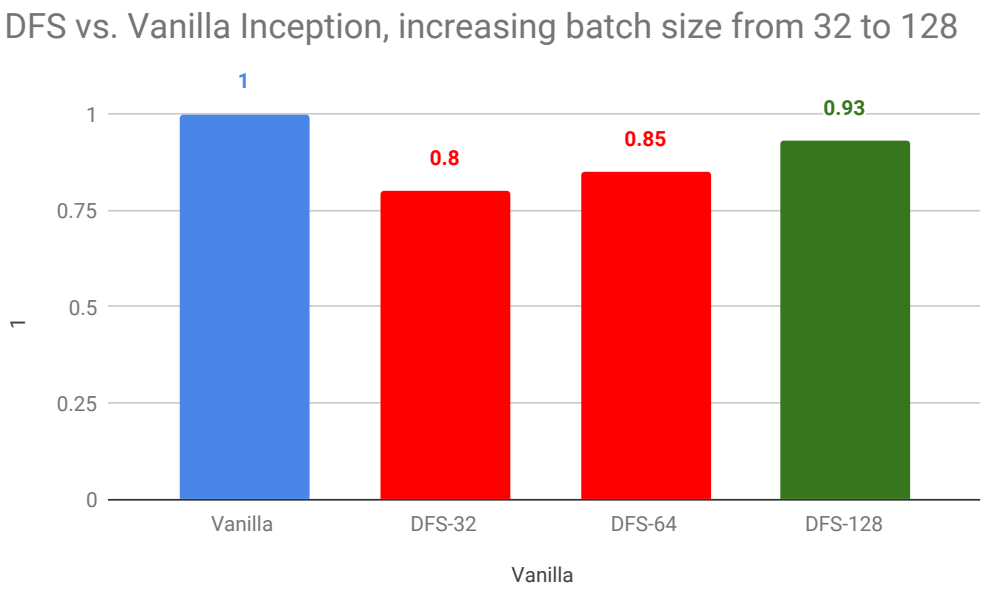

Figure 5.7: The effect of batch size to the relative DFP performance w.r.t Vanilla-TF

As we can see from Figure 5.7, increasing the batch size, or increasing device utilization, allows us to close the gap between ours and Vanilla performance. This also suggests that our algorithm could benefit from very large graphs that have data-hungry nodes more efficiently. The preliminary VGG-16 experiment also seems supports the same claim. Even if parallelism is much less frequent in VGG-16, DFP placement only loses 3% of relative Vanilla performance with VGG-16.

Chapter 6

RELATED WORK

Like we discussed before, parallel deep learning is a heavily researched topic and has many sub-fields of active research. The limitations that we have mentioned earlier in Chapter 2, motivated researchers both from high performance computing and deep learning backgrounds to conduct interdisciplinary high performance deep learning research. In this chapter, we summarize our findings about the different related work in the area. Both theoretical and system optimizations have been utilized to perform high performance deep learning.

6.1 Training Algorithms and System Optimizations

In this thesis, we focused on speeding up neural network training through GPU kernel optimizations and transparent model parallelism. Both approaches are well known practices in parallel computing and high performance computing as discussed in Chapter 2. HPC is not the only way of having faster training. Neural network training time is mostly related to convergence rate. And convergence rate is completely dependent on deep learning theory and mathematics. The work in this field can be both purely theoretical and utilize strong engineering concepts to boost up the performance even more. With the spread of multicore machines and GPUs, parallel setups for deep learning started to becoming more attractive to researchers. Therefore, parallelizing the training algorithm have become inevitable for scalable performance. The first serious attemps to achieve this were done with Google's DistBelief deep learning framework. We already mentioned DistBelief in Section 2.4.3 but only explained its architecture. The framework claims to support both data and full model parallelism. DistBelief handles data parallelism by using a parameter-server. The model parame-

ters are stored in the parameter-server machine. The framework automatically creates k replicas on k different workers. Then, the proposed training algorithm, Downpour SGD, is used to train the neural network in parallel. In Downpour SGD, each replica computes a different partition of the minibatch and the parameter updates to the parameter server are done every few steps rather than every step. This strategy reduces the synchronization points required to do parameter updates to the server while preserving the convergence properties due to stochasticity. In addition, every replica is only responsible of updating its own parameters reducing the communication volume significantly and allowing the training algorithm to scale on many workers without being affected negatively. The authors are able to scale moderate-sized networks up to 2.2x on 8 machines and achieve 12x speed-up with 81 machines on larger models.

While proposing a training algorithm and utilize theoretical concepts in their work, DistBelief authors mostly applied strong engineering concepts inspired by some theoretical properties of the SGD training algorithm. There are other training algorithms that are heavily inspired by theoretical concepts in stochastic learning such as The Elastic Averaging SGD algorithm [Zhang et al., 2015]. EASGD is a distributed training algorithm for deep neural networks that avoids communication by allowing each worker to explore the nearby parameter space. The exploration range is determined by an elastic force proposed by the authors. The force should be set in such a way that the gradient directions of the workers should be towards the global minima. Greater elasticity, allows the variance of the gradients to increase. At first, this may seem odd since randomness increases but surprisingly, EASGD is able to outperform Downpour SGD when the elasticity is set to high values.

Deep neural networks may also contain very large number of parameters. The recently proposed Amoeba network variant Amoeba [Huang et al., 2018] contains around one billion parameters. Such huge structures have enormous communication volumes during training. As we discussed in Section 2.4.1, this high communication rate prevents the training from scaling on multiple workers. There are multiple strategies for reducing communication in deep learning. One of the obvious ones is to

reduce the size of the messages that are being communicated. However, the messages are gradient values, therefore compressing them may result in precision loss if not handled carefully. Seide et al proposed a 1-bit SGD training algorithm to reduce the communication and perform efficient data-parallel training [Seide et al., 2014]. The authors compress the gradients to a single bit denoting only if the gradient is greater than 0 or not. This information loss is normally not tolerable by the convergence rate, but the authors discover that propagating the quantization through the training steps allows the accuracy to stay stable. The quantization error is added to the next minibatch's gradient. By tracking this quantization error and using delayed gradient updates, the authors are able to achieve up to around 6x speed-up by using two machines with 4 GPUs each.

Since most modern accelerators have very good computing power, they should be properly utilized in order to avoid stalls and delays in the concurrent computation. This means that each worker would need more data to process if their current input data volume is too small and causes the underutilization of the device. To prevent this, many researchers worked on large mini-batches. As we already discussed, a mini-batch is a collection of input data points that are processed together at each iteration step in the training phase. In other words, each worker processes a mini-batch of examples every other step. If the volume of this mini-batch is too small, the device will be underutilized. However, increasing the mini-batch size also has its own disadvantages. The classic update rule where a fixed learning rate is used, causes the algorithm to have poor convergence time when large batches are used. Facebook researchers propose a novel strategy for adjusting the learning rate as the minibatch size increases to prevent accuracy loss [Goyal et al., 2017]. The learning rate is increased as the minibatch size increases. This simple change and communication optimizations allow the proposed system to achieve 90% scaling efficiency when moving from 8 GPUs to 256 GPUs. From their experiments they observe that their method works with up to 8192 as the minibatch size without losing any accuracy. With 256 GPUs the system trains Resnet-50 on ImageNet in 1 hour compared to 29 hours with 8 GPUs.

This work was then improved by You et al. [You et al., 2017] by utilizing a different learning rate for each different layer in the network. This is logical because each layer may have completely different statistics and using the same learning rate may hurt the local convergence of the layer and, naturally, the overall performance too. The proposed strategy allows the evaluation neural networks to scale up to 32K mini-batches compared to the previous 8k.

6.2 Parallel Deep Learning Frameworks

In addition to developing new training algorithms and find ways to tackle communication and underutilization, optimized parallel deep learning systems can improve the time performance and memory consumption of the target network significantly. These systems usually use system optimizations such as data-parallelism, model-parallelism or pipeline parallelism.

Earlier, we mentioned DistBelief, TF's initial version. DistBelief supports data-parallel deep learning through parameter servers. A similar framework is GeePS [Cui et al., 2016]. The authors propose a smart data manager for a multiple GPU PS architecture. A distributed deep neural network is divided across different machines, but uses global parameters to update its state. This requires lots of communication between the workers and also between the CPU and the workers. Naively designed frameworks can suffer from this communication amount and may not scale when the number of workers increase too much. In addition, the GPU memory itself might be too small for large models. Existing PS' store the parameter server on CPU. This results in extra communication between CPU and GPU in every step of the training process. In order to prevent this issue, the authors distribute the parameter server across GPU workers and reduce the communication amount and also allow hiding in-GPU communication with GPU computation and reducing GPU idle-time. This reduction in communication allows the system to scale well. Another important issue with multi GPU deep learning is device memory limits. The authors address this issue by swapping some of the parameters that are not being used to the CPU memory.

This data movement is also hidden with training computation to further improve performance. The data movement between the CPU and GPUs is managed by using buffers. Whenever data is moved between CPU and GPU, a buffer in GPU memory is created and the pointer is passed to the application instead of directly copying the data. Also the intermediate states (non-parametric local data) can be stored in the parameter server. This data will not be shared among the workers. For local reads, there is no need for a separate buffer, therefore reads in this fashion will be much faster. GeePS achieves near-linear scaling up to 16 GPUs compared to Caffe single-GPU baseline. Moreover, the proposed framework achieves a bigger throughput with only 4 GPUs than a CPU-only setup is achieving with 108 machines.

Data parallel methods have significant positive impact on training time, however they suffer from large number of workers as the communication step becomes too heavy and dominates the overall execution time. Data parallelism scales up the batch size in order to utilize each worker efficiently. However, different hyper-parameters are required for different mini-bath sized neural networks. On the other hand, model parallelism may suffer from networks having not enough parallelism to distribute. Authors, therefore, focused on pipeline parallelism to increase the throughput of each worker while concurrently calculate different steps. Google presented GPipe, a scalable deep learning system that can be used to train gigantic networks. It leverages recomputation of forward passes in back-propagation phase in order to minimize memory usage (redundant calculations). The implemented system is able to train a 557 million parameter deep neural network AmoebaNet and achieve new state-of-art accuracy for the ImageNet 2012 dataset. Their method works well with smaller datasets CIFAR-10 and CIFAR-100 too and break the state-of-art accuracy for both. The importance of partitioning the layers of the pipeline is underlined by the authors multiple times in the paper and suggest that our work could be integrated into this pipeline parallel scheme to achieve even more performance.

Machine learning itself was also proposed to generate efficient device placements for TF. Google proposed a reinforcement learning model that is able to generate

device placement patterns for the TF nodes that result in faster execution time [Mirhoseini et al., 2018]. The proposed model consists of two separate components: a grouper and a placer. The grouper learns which nodes to collapse and the placer learns the best devices to place the operations. The proposed method is able to achieve up to 60% speed-up for large models like the neural machine translation [Bahdanau et al., 2014]. However, the evaluation is not done for modern GPUs such as P100 and V100 models and may cause underutilization when tried with them. Moreover, training the proposed model takes 12.5 hours for NMT which is a significant overhead.

Chapter 7

CONCLUSION

In this thesis, we explore the different optimization approaches that can be performed to speed up neural network training. We first implement the multidimensional version of a frequently used deep learning kernel and optimize it for the GPU architecture. Our performant implementation avoids extra synchronization points and kernel launches and performs 56x better than the baseline.

Secondly, we propose an automated model parallelism strategy for TensorFlow. We develop an offline profiler and device placement module. We compare our proposed algorithm's performance with popular graph partitioning tools METIS and Zoltan. Our method outperforms the mentioned libraries, however it cannot surpass Vanilla-TF performance. We suspect underutilization as its reason and present supporting preliminary findings. By disabling the TensorFlow placer module completely and performing a detailed performance analysis of our actual DFP placements, we will continue to improve the strategy and achieve speed-ups, especially for larger and heavier networks.

BIBLIOGRAPHY

[shu, 2017] (2017). Warp shuffle functions. http://docs.nvidia.com/cuda/cuda-c-programming-guide/warp-shuffle-functions. [Online; accessed 4-February-2018].

[Abadi et al., 2015] Abadi, M., Agarwal, A., Barham, P., Brevdo, E., Chen, Z., Citro, C., Corrado, G. S., Davis, A., Dean, J., Devin, M., Ghemawat, S., Goodfellow, I., Harp, A., Irving, G., Isard, M., Jia, Y., Jozefowicz, R., Kaiser, L., Kudlur, M., Levenberg, J., Mané, D., Monga, R., Moore, S., Murray, D., Olah, C., Schuster, M., Shlens, J., Steiner, B., Sutskever, I., Talwar, K., Tucker, P., Vanhoucke, V., Vasudevan, V., Viégas, F., Vinyals, O., Warden, P., Wattenberg, M., Wicke, M., Yu, Y., and Zheng, X. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. Software available from tensorflow.org.

[Bahdanau et al., 2014] Bahdanau, D., Cho, K., and Bengio, Y. (2014). Neural machine translation by jointly learning to align and translate. cite arxiv:1409.0473Comment: Accepted at ICLR 2015 as oral presentation.

[Bezanzon et al., 2012] Bezanzon, J., Karpinski, S., Shah, V., and Edelman, A. (2012). *Julia: A Fast Dynamic Language for Technical Computing.*

[Chapman et al., 2007] Chapman, B., Jost, G., and Pas, R. v. d. (2007). *Using OpenMP: Portable Shared Memory Parallel Programming (Scientific and Engineering Computation).* The MIT Press.

[Chen et al., 2015] Chen, T., Li, M., Li, Y., Lin, M., Wang, N., Wang, M., Xiao, T., Xu, B., Zhang, C., and Zhang, Z. (2015). Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. *CoRR*, abs/1512.01274.

[Cui et al., 2016] Cui, H., Zhang, H., Ganger, G. R., Gibbons, P. B., and Xing, E. P. (2016). Geeps: Scalable deep learning on distributed gpus with a gpu-specialized parameter server. In *Proceedings of the Eleventh European Conference on Computer Systems*, EuroSys '16, pages 4:1–4:16, New York, NY, USA. ACM.

[Dean et al., 2012] Dean, J., Corrado, G. S., Monga, R., Chen, K., Devin, M., Le, Q. V., Mao, M. Z., Ranzato, M., Senior, A., Tucker, P., Yang, K., and Ng, A. Y. (2012). Large scale distributed deep networks. In *NIPS*.

[Devine et al., 2002] Devine, K., Boman, E., Heaphy, R., Hendrickson, B., and Vaughan, C. (2002). Zoltan data management services for parallel dynamic applications. *Computing in Science Engineering*, 4(2):90–96.

[Devine et al., 2000] Devine, K., Hendrickson, B., Boman, E., John, M. S., and Vaughan, C. (2000). Design of dynamic load-balancing tools for parallel applications. In *Proc. Intl. Conf. on Supercomputing*, pages 110–118, Santa Fe, New Mexico.

[Devine et al., 2006] Devine, K. D., Boman, E. G., Heaphy, R. T., Bisseling, R. H., and Catalyurek, U. V. (2006). Parallel hypergraph partitioning for scientific computing. In *Proceedings 20th IEEE International Parallel Distributed Processing Symposium*, pages 10 pp.–.

[Dikbayır et al., 2018] Dikbayır, D., Çoban, E. B., Kesen, I., Yuret, D., and Unat, D. (2018). Fast multidimensional reduction and broadcast operations on GPU for machine learning. *Concurrency and Computation: Practice and Experience*.

[Foley and Danskin, 2017] Foley, D. and Danskin, J. (2017). Ultra-performance pascal gpu and nvlink interconnect. *IEEE Micro*, 37(2):7–17.

[Gabriel et al., 2004] Gabriel, E., Fagg, G. E., Bosilca, G., Angskun, T., Dongarra, J. J., Squyres, J. M., Sahay, V., Kambadur, P., Barrett, B., Lumsdaine, A., Castain,

R. H., Daniel, D. J., Graham, R. L., and Woodall, T. S. (2004). Open MPI: Goals, concept, and design of a next generation MPI implementation. In *Proceedings, 11th European PVM/MPI Users' Group Meeting*, pages 97–104, Budapest, Hungary.

[Gansner et al., 2006] Gansner, E., Koutsofios, E., and North, S. (2006). Drawing graphs with dot. Technical report.

[Ghorpade et al., 2012] Ghorpade, J., Parande, J., Kulkarni, M., and Bawaskar, A. (2012). GPGPU processing in CUDA architecture. *CoRR*, abs/1202.4347.

[Goyal et al., 2017] Goyal, P., Dollár, P., Girshick, R. B., Noordhuis, P., Wesolowski, L., Kyrola, A., Tulloch, A., Jia, Y., and He, K. (2017). Accurate, large minibatch SGD: training imagenet in 1 hour. *CoRR*, abs/1706.02677.

[Graves et al., 2013] Graves, A., Mohamed, A., and Hinton, G. E. (2013). Speech recognition with deep recurrent neural networks. *CoRR*, abs/1303.5778.

[Harlap et al., 2018] Harlap, A., Narayanan, D., Phanishayee, A., Seshadri, V., Devanur, N. R., Ganger, G. R., and Gibbons, P. B. (2018). Pipedream: Fast and efficient pipeline parallel DNN training. *CoRR*, abs/1806.03377.

[Harris, 2007] Harris, M. (2007). Optimizing cuda. *Supercomputing Conference, Reno, NV*.

[He et al., 2015] He, K., Zhang, X., Ren, S., and Sun, J. (2015). Deep residual learning for image recognition. *CoRR*, abs/1512.03385.

[Hillis and Steele, 1986] Hillis, W. D. and Steele, Jr., G. L. (1986). Data parallel algorithms. *Commun. ACM*, 29(12):1170–1183.

[Hochreiter and Schmidhuber, 1997] Hochreiter, S. and Schmidhuber, J. (1997). Long short-term memory. *Neural Comput.*, 9(8):1735–1780.

[Huang et al., 2018] Huang, Y., Cheng, Y., Chen, D., Lee, H., Ngiam, J., Le, Q. V., and Chen, Z. (2018). Gpipe: Efficient training of giant neural networks using pipeline parallelism. *CoRR*, abs/1811.06965.

[Karypis and Kumar, 1995] Karypis, G. and Kumar, V. (1995). Metis – unstructured graph partitioning and sparse matrix ordering system, version 2.0. Technical report.

[Karypis and Kumar, 1996] Karypis, G. and Kumar, V. (1996). Parallel multilevel k-way partitioning scheme for irregular graphs. In *Proceedings of the 1996 ACM/IEEE Conference on Supercomputing*, Supercomputing '96, Washington, DC, USA. IEEE Computer Society.

[Karypis and Kumar, 1998] Karypis, G. and Kumar, V. (1998). Multilevel algorithms for multi-constraint graph partitioning. In *SC '98: Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, pages 28–28.

[Krizhevsky, 2014] Krizhevsky, A. (2014). One weird trick for parallelizing convolutional neural networks. *CoRR*, abs/1404.5997.

[Krizhevsky et al., 2012] Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In Pereira, F., Burges, C. J. C., Bottou, L., and Weinberger, K. Q., editors, *Advances in Neural Information Processing Systems 25*, pages 1097–1105. Curran Associates, Inc.

[Lecun et al., 1998] Lecun, Y., Bottou, L., Bengio, Y., and Haffner, P. (1998). Gradient-based learning applied to document recognition. In *Proceedings of the IEEE*, pages 2278–2324.

[Liu et al., 2017] Liu, B., Wen, C., Sarwate, A. D., and Dehnavi, M. M. (2017). A unified optimization approach for sparse tensor operations on gpus. *2017 IEEE International Conference on Cluster Computing, CLUSTER 2017, Honolulu, HI, USA, September 5-8, 2017*, pages 47–57.

[Luitijens, 2014] Luitijens, J. (2014). Faster parallel reductions on kepler. https://devblogs.nvidia.com/faster-parallel-reductions-kepler. [Online; accessed 4-February-2018].

[Makpaisit et al., 2015] Makpaisit, P., Ichikawa, K., Uthayopas, P., Date, S., Takahashi, K., and Khureltulga, D. (2015). Mpi_reduce algorithm for openflow-enabled network. *2015 15th International Symposium on Communications and Information Technologies (ISCIT)*, pages 261–264.

[Manne, 1959] Manne, A. S. (1959). On the Job Shop Scheduling Problem. Technical report.

[Mirhoseini et al., 2018] Mirhoseini, A., Goldie, A., Pham, H., Steiner, B., Le, Q. V., and Dean, J. (2018). Hierarchical planning for device placement.

[Mortensen et al., 2007] Mortensen, J., Khanna, P., and Slater, M. (2007). Light field propagation and rendering on the gpu. pages 15–23.

[Murtagh, 1991] Murtagh, F. (1991). Multilayer perceptrons for classification and regression. *Neurocomputing*, 2(5):183 – 197.

[Neumann, 2008] Neumann, A. B. (2008). Parallel Reduction of Multidimensional Arrays for supporting Online Analytical Processing (OLAP) on a Graphics Processing Unit (GPU).

[Nickolls, 2007] Nickolls, J. (2007). Gpu parallel computing architecture and cuda programming model. In *2007 IEEE Hot Chips 19 Symposium (HCS)*, pages 1–12.

[Novikoff, 1962] Novikoff, A. B. (1962). On convergence proofs on perceptrons. In *Proceedings of the Symposium on the Mathematical Theory of Automata*, volume 12, pages 615–622, New York, NY, USA. Polytechnic Institute of Brooklyn.

[NVIDIA, 2013] NVIDIA (2013). Nvidia k40m active board specifications. https://www.nvidia.com/content/PDF/kepler/Tesla-K40-Active-Board-Spec-BD-06949-001$_v$03.*pdf*. [*Online*; *accessed*28 − *January* − 2018].

[Paszke et al., 2017] Paszke, A., Gross, S., Chintala, S., Chanan, G., Yang, E., DeVito, Z., Lin, Z., Desmaison, A., Antiga, L., and Lerer, A. (2017). Automatic differentiation in pytorch.

[Pinheiro and Collobert, 2014] Pinheiro, P. and Collobert, R. (2014). Recurrent convolutional neural networks for scene labeling. In Xing, E. P. and Jebara, T., editors, *Proceedings of the 31st International Conference on Machine Learning*, volume 32 of *Proceedings of Machine Learning Research*, pages 82–90, Bejing, China. PMLR.

[Raina et al., 2009] Raina, R., Madhavan, A., and Ng, A. Y. (2009). Large-scale deep unsupervised learning using graphics processors. In *Proceedings of the 26th Annual International Conference on Machine Learning*, ICML '09, pages 873–880, New York, NY, USA. ACM.

[Rosenblatt, 1958] Rosenblatt, F. (1958). The perceptron: A probabilistic model for information storage and organization in the brain. *Psychological Review*, pages 65–386.

[Rumelhart et al., 1986] Rumelhart, D. E., Hinton, G. E., and Williams, R. J. (1986). Learning Representations by Back-propagating Errors. *Nature*, 323(6088):533–536.

[Seide et al., 2014] Seide, F., Fu, H., Droppo, J., Li, G., and Yu, D. (2014). 1-bit stochastic gradient descent and application to data-parallel distributed training of speech dnns. In *Interspeech 2014*.

[Shamir, 2016] Shamir, O. (2016). Without-replacement sampling for stochastic gradient methods. In Lee, D. D., Sugiyama, M., Luxburg, U. V., Guyon, I., and

Garnett, R., editors, *Advances in Neural Information Processing Systems 29*, pages 46–54. Curran Associates, Inc.

[Simonyan and Zisserman, 2014] Simonyan, K. and Zisserman, A. (2014). Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556*.

[Unat et al., 2010] Unat, D., Cai, X., and Baden, S. (2010). Optimizing the aliev-panfilov model of cardiac excitation on heterogeneous systems. Talk at Para 2010: State of the Art in Scientific and Parallel Computing in Reykjavik on June 6-9, 2010.

[Xu et al., 2015] Xu, K., Ba, J., Kiros, R., Cho, K., Courville, A. C., Salakhutdinov, R., Zemel, R. S., and Bengio, Y. (2015). Show, attend and tell: Neural image caption generation with visual attention. *CoRR*, abs/1502.03044.

[You et al., 2017] You, Y., Gitman, I., and Ginsburg, B. (2017). Scaling SGD batch size to 32k for imagenet training. *CoRR*, abs/1708.03888.

[Yuret, 2016] Yuret, D. (2016). Knet: beginning deep learning with 100 lines of julia. *Machine Learning Systems Workshop at NIPS 2016*.

[Zhang et al., 2015] Zhang, S., Choromanska, A. E., and LeCun, Y. (2015). Deep learning with elastic averaging sgd. In Cortes, C., Lawrence, N. D., Lee, D. D., Sugiyama, M., and Garnett, R., editors, *Advances in Neural Information Processing Systems 28*, pages 685–693. Curran Associates, Inc.

[Zhou et al., 2017] Zhou, S., Greenspan, H., and Shen, D. (2017). *Deep Learning for Medical Image Analysis*. Elsevier and MICCAI Society book series. Elsevier Science & Technology Books.