An Evaluation of TensorFlow as a Programming Framework for HPC Applications

WEI DER CHIEN
An Evaluation of TensorFlow as a Programming Framework for HPC Applications

WEI DER CHIEN
Abstract

In recent years, deep-learning, a branch of machine learning gained increasing popularity due to their extensive applications and performance. At the core of these application is dense matrix-matrix multiplication. Graphics Processing Units (GPUs) are commonly used in the training process due to their massively parallel computation capabilities. In addition, specialized low-precision accelerators have emerged to specifically address Tensor operations. Software frameworks, such as TensorFlow have also emerged to increase the expressiveness of neural network model development. In TensorFlow computation problems are expressed as Computation Graphs where nodes of a graph denote operation and edges denote data movement between operations. With increasing number of heterogeneous accelerators which might co-exist on the same cluster system, it became increasingly difficult for users to program efficient and scalable applications. TensorFlow provides a high level of abstraction and it is possible to place operations of a computation graph on a device easily through a high level API. In this work, the usability of TensorFlow as a programming framework for HPC application is reviewed. We give an introduction of TensorFlow as a programming framework and paradigm for distributed computation. Two sample applications are implemented on TensorFlow: tiled matrix multiplication and conjugate gradient solver for solving large linear systems. We try to illustrate how such problems can be expressed in computation graph for distributed computation. We perform scalability tests and comment on performance scaling results and quantify how TensorFlow can take advantage of HPC systems by performing microbenchmarking on communication performance. Through this work, we show that TensorFlow is an emerging and promising platform which is well suited for a particular class of problem which requires very little synchronization.
Sammanfattning

Ethics and sustainability

In this work no personal data is used or obtained. The methods and experiments were originally developed and references for the development are cited where appropriate. This work used computing resources from KTH PDC, a power consumption aware supercomputing center where heat from cooling is re-used for building heating at KTH.
Acknowledgment

I would like to thank my family for their support while I was working on this thesis. I am grateful to my supervisor Stefano Markidis for the opportunity to work with this project and his guidance for this work as well as for publications made related to this work. I am equally grateful to my examiner Erwin Laure for his comments and feedback. I appreciate my opponent Ardhendu Shekhar Tripathi for pointing out issues and providing useful suggestions to this report. I would also take this opportunity to thank my co-worker Chaitanya Prasad Sishtla for his help in verifying and testing the software implementations in this study as well as for his help on a related publication; and my colleague Rami Karim for his assistance in a related research work and publication. Apart from that, I would like to thank my friends Sandra Karlsson and Ludwik Janiuk for their help on correcting, improving and proofreading the Swedish abstract of this work.
Abbreviations

- **AI** Artificial Intelligence
- **BLAS** Basic Linear Algebra Subprograms
- **CG** Conjugate Gradient
- **FIFO** First In First Out
- **GEMM** General Matrix Multiplication
- **GPU** Graphics Processing Unit
- **HPC** High Performance Computing
- **HTTP** Hypertext Transfer Protocol
- **MPI** Message Passing Interface
- **NCCL** NVIDIA Collective Communications Library
- **PGAS** Partitioned Global Address Space
- **RDMA** Remote Direct Memory Access
- **RDD** Resilient Distributed Dataset
- **RPC** Remote Procedure Call
- **TCP** Transmission Control Protocol
- **TPU** Tensor Processing Unit
Contents

1 Introduction 1
  1.1 Motivation ............................................ 1
  1.2 Research Questions .................................. 2
  1.3 Contribution ......................................... 3
  1.4 Outline ............................................... 3

2 Related Works 5
  2.1 Traditional programming approaches in HPC ........ 5
  2.2 Recent work on TensorFlow ........................... 6
  2.3 Emerging HPC programming framework and platforms 8
  2.4 Machine learning and AI frameworks
      adopted on HPC systems .............................. 9

3 Background 10
  3.1 Overview ............................................. 10
  3.2 Computation Graph ................................... 11
      3.2.1 Operation node categories ...................... 12
      3.2.2 Device placement ................................. 13
  3.3 Memory management .................................. 14
  3.4 Queues and dataset .................................. 15
      3.4.1 Queues ........................................... 15
      3.4.2 Dataset .......................................... 17
  3.5 Computation Kernels ................................ 17
  3.6 Distributed TensorFlow ............................... 17
      3.6.1 In-graph replication ............................... 18
      3.6.2 Between-graph replication ...................... 19
  3.7 Communication modules ............................... 21
      3.7.1 gRPC ............................................. 22
      3.7.2 GPUDirect RDMA ................................. 22
# CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.7.3 InfiniBand RDMA</td>
<td>23</td>
</tr>
<tr>
<td>3.7.4 MPI</td>
<td>24</td>
</tr>
<tr>
<td>4 Methods</td>
<td>25</td>
</tr>
<tr>
<td>4.1 Measurement of communication bandwidth</td>
<td>25</td>
</tr>
<tr>
<td>4.2 Expressing algorithms in TensorFlow</td>
<td></td>
</tr>
<tr>
<td>4.2.1 CG Solver</td>
<td>26</td>
</tr>
<tr>
<td>4.2.2 Optimized CG Solver</td>
<td>33</td>
</tr>
<tr>
<td>4.2.3 Tiled matrix multiplication</td>
<td>38</td>
</tr>
<tr>
<td>5 Experimental setup</td>
<td>41</td>
</tr>
<tr>
<td>5.1 Communication Performance</td>
<td>42</td>
</tr>
<tr>
<td>5.2 CG solver</td>
<td>42</td>
</tr>
<tr>
<td>5.3 Optimized CG solver</td>
<td>43</td>
</tr>
<tr>
<td>5.4 Tiled matrix multiplication</td>
<td>43</td>
</tr>
<tr>
<td>6 Results</td>
<td>44</td>
</tr>
<tr>
<td>6.1 Communication performance</td>
<td>44</td>
</tr>
<tr>
<td>6.2 CG solver</td>
<td>45</td>
</tr>
<tr>
<td>6.3 Optimized CG solver</td>
<td>47</td>
</tr>
<tr>
<td>6.4 Tiled matrix multiplication</td>
<td>49</td>
</tr>
<tr>
<td>7 Discussion and conclusions</td>
<td>51</td>
</tr>
<tr>
<td>7.1 Scientific closure</td>
<td>51</td>
</tr>
<tr>
<td>7.2 Discussion</td>
<td>52</td>
</tr>
<tr>
<td>7.3 Future work</td>
<td>53</td>
</tr>
<tr>
<td>Bibliography</td>
<td>56</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Motivation

In recent years, deep-learning, a branch of machine learning became extremely popular due to their extensive applications and performance. Through a minimization process, the weights of individual neurons in a neural network are determined. One of the most important operation required during a training process is dense matrix matrix and matrix vector multiplication, or simply Tensor operation [13]. In order to speed-up the training process, Graphics Processing Units (GPUs) are commonly used due to their massively parallel computation capability [30]. In addition, specialized accelerators have emerged to specifically address Tensor operations [31]. These accelerators are highly specialized for neural network training and provides considerable speed improvement. One specialty of these architecture is that they usually operate in low precision environment, as neural network training is typically resistant to precision loss up to a certain degree [16][24][37]. One example is the recently released NVIDIA Tensor Core included in the latest Volta architecture. A Tensor Core per clock cycle performs one $4 \times 4$ GEMM (GEneral Matrix Multiply) in mixed precision [32][40]. Apart from specialized hardware, software frameworks have also emerged to increase the expressiveness of neural network and Tensor operations. One typical example is TensorFlow developed by Google. In TensorFlow computation problems are expressed as Computation Graphs where nodes of a graph denote operation and edges denote data movement between operations. Once a graph is to be executed, workloads are automatically assigned to available accelerator devices [2]. In particular,
TensorFlow supports distributed training and is able to work across a cluster of computing nodes. If available, it is possible to communicate through Remote Direct Memory Access (RDMA), GPU Direct or MPI [45].

With increasing number of heterogeneous hardware platform and accelerators which might be co-existing on the same cluster system, it became increasingly difficult for users to program efficient, scalable while portable applications. TensorFlow is one such effort to bring in abstraction between platform development and application development such that users can take advantage of the power of these hardware without having to explicitly program for them. The objective of this work is to study the usability and expressiveness of using TensorFlow as a programming framework for HPC applications. One example is the distributed computation of a very large GEMM operation, which does not fit into GPU memory of one single node. Another example is to solve a large linear system with a Conjugate Gradient Solver. By expressing these problems as computation graph and through the distribution model of TensorFlow, we evaluate the performance, scalability and the usability of TensorFlow on HPC systems. We would like to understand if TensorFlow can enhance the usability of these hardware platforms by providing users a high-level expressiveness while at the same time providing high performance and scalability.

1.2 Research Questions

In this section, we outline three main research questions for this work.

How to express typical HPC problems as graph on TensorFlow? It is immediately obvious that one conceptual challenge of programming TensorFlow for non neural network specification is to expressive a computation problem in graph. Since Computation Graph is a high level expression, an even more challenging issue is to express these problems distributively in order to achieve scalability. We evaluate these conceptual issues by implementing two of the most commonly used tool for scientific applications: large matrix multiplication and CG solver.
How scalable and performant is TensorFlow?  We would like to study if TensorFlow, as a high level framework to the user, can still deliver high computation performance. One evaluation is by measuring the performance and comparing with similar solutions. An important aspect of software HPC system is scalability. We access this aspect by benchmarking the same applications that we are going to implement with distributed TensorFlow.

How can TensorFlow take advantage of HPC specific high performance architectures?  An important aspect of evaluating TensorFlow as an application development framework on HPC systems is if the framework is able to take advantage of HPC specific system architectures. One example is how well can distributed TensorFlow take advantage of HPC networks such as Infiniband and NVLink. The current version of TensorFlow supports gRPC through TCP, RDMA, and MPI as means of communication. Measurement programs will be developed on TensorFlow to evaluate how well can the framework take advantage of these architectures.

1.3 Contribution

This study brings the following contributions: We evaluate if TensorFlow framework is a suitable candidate for developing HPC applications by studying the characteristics of TensorFlow and elaborate on how HPC users can take advantage of the high usability and expressiveness in Tensor operations. We also look into the increased level of abstraction when programming heterogeneous accelerators such as GPUs, which can potentially ease development effort. We implement and study the performance and scalability of two common applications on TensorFlow, perform benchmarking and comment on results.

1.4 Outline

This work is organized as following. We provide an overview of related work on TensorFlow and emerging programming frameworks for developing HPC application, followed by a detail introduction of TensorFlow. We try to summarize the functionality and expressiveness of TensorFlow by introducing key concepts which are applicable. We
introduce our sample algorithms and explain the implementation effort required with TensorFlow. Finally we perform benchmarking and comment on the performance results.
Chapter 2

Related Works

In this chapter, we provide an overview of existing programming frameworks for HPC applications. We review emerging HPC programming models and frameworks. We also provide an overview of programming framework and platforms that are typically used in AI and Machine Learning communities which are being increasingly adapted on HPC systems.

2.1 Traditional programming approaches in HPC

Application programming on HPC systems often falls into two categories: for shared memory system or for distributed memory system. In the case of distributed memory system, computation, with the exception of embarrassingly parallel problem, often requires data exchanges between processes collocated on different computing nodes within the systems. One such way of programming these systems is through the Message Passing Interface (MPI).

MPI is an umbrella of standardized routines that performs message passing between processors on different systems [35]. These routines provide virtual topology, synchronization functions and communication functions between a set of processes. Since MPI is a standard, it is often language and vendor independent. MPI supports point-to-point communication between two processes; collectives, which all processes participate in communication; and one-sided communication where only one process actively communicate with another process without
the other process explicitly participating in communication. In addition to
communication of in-memory data, MPI supports parallel I/O with
processes. MPI-IO is a set of MPI functions designed to abstract I/O
management on distributed systems such that files can be accessed
in parallel by MPI processes [34][48]. The latest version of MPI is 3.0
and the two major MPI libraries are OpenMPI [22] and MPICH [36],
together with vendor specific implementations, such as Intel MPI [26]
and Fujitsu MPI [9]. Depending on the MPI implementation, support
for network protocols like InfiniBand is often included [42].

In terms of shared memory system, meaning computation within
one computing node, OpenMP is often used. OpneMP implements
a fork-join model for threading by compiler directives [17]. OpenMP
allows both data parallelism and task parallelism. Data parallelism is
achieved by spawning threads which run the same segment of code.
Task parallelism is achieved by work sharing where each thread exe-
cutes an allocated part of the code, for example the automatic unrolling
of loops. In the recent version of OpenMPI, MPI interprocess shared
memory extension (MPI SHM) is added. This allows MPI processes to
access shared memory regions though MPI window [19].

2.2 Recent work on TensorFlow

TensorFlow is a machine learning framework designed by Google with
a focus on expressing neural network problems [2]. The framework
provides a Python frontend and the runtime is developed in C++. It
was opened sourced [47] by Google in 2015 and was originally part of
the Google Brain project. DistBelief [18] was an internal Google project
which is the predecessor of TensorFlow. It is a framework which allows
distributed training of deep neural networks and allows different dis-
tributed optimization strategies. In TensorFlow, computation problems
are expressed in computation graphs for easy development and connec-
tion of neural networks components. The approach follows the use of
computation graph which was pioneered by Theano. Theano expresses
mathematical expressions as directed graphs with variable nodes and
operation nodes [8]. Furthermore Theano supports computation with
mutiple computation devices (e.g. GPU). Theano is currently discon-
tinued and version 1.0 was the final release [50]. Apart from the use of
computation graph, TensorFlow allows the use of different accelerators
by simply pinning these operation nodes of the computation graph to a
device. It also supports concurrent use of multiple different accelerators
in the same graph. Since TensorFlow was open sourced, contributions
made to the platform by the community has been rapid [15].

Support for various platform dependent optimizations, such as CPU
SIMD instructions were added. In terms of pipeline performance, Ac-
celerated Linear Algebra (XLA) was developed to improve execution
speed and memory usage. XLA eliminates overhead from TensorFlow
runtime and improve memory usage by fusing short-lived pipelined
operations. XLA supports both Ahead-of-Time and Just-in-Time compi-
lation when the computation graph is processed. Studies have also
shown that the performance of TensorFlow runtime can be improved
by optimizing scheduling [1][51].

The default communication protocol of TensorFlow in a distributed
setting is gRPC [23]. In terms of communication performance, studies
have shown that the default gRPC protocol is unable to capture the
power of high performance interconnect, which is commonly available
on HPC platforms [33]. Thus, GPUDirect and InfiniBand verbs RDMA
were developed to support the use of high speed RDMA in distributed
TensorFlow [27][45][46][52]. Apart from InfiniBand RDMA, MPI is also
supported. It is possible to use MPI as a process launcher and as an
underlying communicator [25][49]. Techniques such as all-ring-reduce
by MPI were implemented to reduce communication. Horovod is a plu-
gin framework for distributed training in TensorFlow. It uses MPI for
process distribution and NCCL by NVIDIA for reduction between pro-
cesses [41]. When adopting these communication protocols, it is likely
that gRPC is still used for administrative purposes, such as to transfer
metadata and control messages. Therefore it is important to quantify
and improve the performance of gRPC. Studies have recently been
made to quantify the performance of gRPC and various protocols [10]
by building a performance test suite.

In terms of data input, TensorFlow supports the use of various
data input mechanisms. Among the supported systems are POSIX file
systems, Google Cloud Storage System (GCS), Amazon Object Stores
(S3) as well as Hadoop File System (HDFS) [4].
2.3 Emerging HPC programming framework and platforms

Various programming models and frameworks have emerged in recent years in an attempt to address the difficulties in developing applications on HPC platforms. One such model is Partitioned Global Address Space (PGAS). PGAS is a parallel programming model which assumes a global address space for all processes whereas the space is in fact partitioned for each process such that each process owns a partition of the memory space [21]. UPC is an extension of C language which implements global address space abstraction and management [20]. It also manages data communication on shared memory and distributed memory systems. Similar languages include Chapel [12] by Cray and UPC++ [55].

Apart from PGAS, various task based programming models have emerged to support the use of multi-core and heterogeneous architectures. StarPU is a task based programming framework where tasks are represented as a series of directed graph [5]. Tasks in a graph are scheduled for execution in such a way that all dependencies are satisfied. It additionally supports the use of MPI in a distributed setting. In that case one MPI process is launched per computing node and all the CPU and GPU computation on the node will be handled by StarPU. PaRSEC is another task based programming framework. Tasks dependencies in PaRSEC [11] are represented as directed graphs and similar to StarPU, these tasks are scheduled to run in such a way that dependencies are satisfied. On a distributed memory system, MPI is used as underlying communicator.

Some other novel solutions for managing applications for shared memory system include replicated OS model. One example is Popcorn Linux [7], a replicated kernel operating system based on Linux. Multiple kernels are run on different nodes and these kernels collaborate with others. The operating system exposes a single image view to applications over all the computing nodes. The kernels on the computing nodes are responsible for consistencies of the memory view across all computing nodes.
2.4 Machine learning and AI frameworks adopted on HPC systems

Apart from TensorFlow, many of the machine learning and AI frameworks are being increasingly adopted on HPC systems. Caffe is a deep-learning framework that uses GPU for training [28]. The framework particularly focuses on Convolutional Neural Networks for computer vision applications. The framework takes a modular approach for its training pipeline, however the expressiveness is quite restricted and is neural network oriented. The original distribution of Caffe operates on one single machine. Other distribution of Caffe which supports distributed training were cdeveloped. S-Caffe supports distributed GPU training on a cluster and primarily focuses on co-designing the framework with CUDA aware MPI, which provides better interoperability between CUDA and MPI [6].

Spark is a cluster programming framework which focuses on fault tolerant. It introduces the concept of Resilient Distributed Datasets (RDDs). RDDs can be seen as distributed shared memory objects. In particular in the case of failure, the dataset can be reconstructed by other parts of the distribution [53][54].
Chapter 3

Background

In this chapter, we provide an overview of TensorFlow as well as essential concepts required for building applications. We first motivate the programming approach in TensorFlow with a discussion on how a computation problem can be expressed by the use of computational graph. We follow by introducing how data is consumed by the graph and how to construct an input-computation pipeline. We also discuss on how the pipeline can be replicated and distributed to multiple nodes in a computing cluster. Paradigms with distributed TensorFlow runtime will be introduced and we give a brief discussion on distributed programming models, specific to distributed TensorFlow. A large part of the introduction outlines the TensorFlow white paper [3].

3.1 Overview

TensorFlow is a machine learning framework that focuses on expressing neural network problems at scale on heterogeneous systems. It is designed by Google and was officially open sourced through Apache 2.0 license in November, 2015. The project started as part of the Google Brain project in 2011 to explore different large neural networks. The platform is a successor of DistBelief, a neural network training and inferencing system for supervised learning.

In TensorFlow, a computation problem is expressed as a computation graph through a *Python* or *C++* API, where nodes represent computation operations with one or more inputs and zero or more outputs. Edges between nodes represent data movement. Flowing between nodes, are *Tensors*, which are multi-dimensional matrices. Computation
in TensorFlow is only executed when an operation is executed by a session. When calling the TensorFlow API, instead of executing the statement, it constructs a node in the computation graph. A graph can be executed by invoking a computation node through a *session*. A TensorFlow *session* is the primary way for a client to interact with a graph which is managed by the TensorFlow runtime. When a node is invoked by a client session, the node being invoked becomes the sink and all the computation nodes between the source and the sink will be invoked in an order such that all dependencies will be satisfied.

TensorFlow allows the placement of different data and operations on different computation devices, such as CPU and GPUs. Data operations can either be pinned by the TensorFlow runtime to different accelerators such as different GPUs or manually. A communication end-point is automatically added between computation nodes if the edge goes across devices i.e. CPU and GPU. In the case of distributed TensorFlow, the underlying communication protocol is *gRPC*, a *RPC* protocol based on TCP developed by Google. If *InfiniBand* is available, RDMA through *InfiniBand* is possible. TensorFlow also supports using MPI as the underlying communication manager.

### 3.2 Computation Graph

*Computation Graph* is a directed graph with a set of nodes that represents dataflow. Computation nodes on the graph are either data source or operation [1]. Data sources are typically *Tensors* or special stateful data sources which returns a *Tensor* to downstream operations. These data sources are either mutable or immutable.

In TensorFlow, a *Tensor* is a typed multi-dimensional array which stores data between 8 to 64 bits. Additionally it supports complex number type and strings. A *Placeholder* is an important kind of node which resembles a tensor but the data content is supplied by a client session during execution of the graph. A *Constant* is an immutable Tensor which resides on a device. A *Variable* is like a *Constant* tensor but its content is mutable through operations such as *assign* and *assign_add*. In addition, *Variable* is persistent in the sense that it is able to survive through multiple execution the graph. All *Placeholder*, *Variable* and *constant* requires their shape to be defined. *Shape* refers to the dimension of the array. *Queue* and *Dataset* are special stateful data nodes which
tensors are supplied whenever requested by downstream operations.

![Figure 3.1: A simple computation graph with two tensors A and B each with the same shape.](image)

Figure 3.1 illustrates a computation graph where two Tensors $A$ and $B$ with the same shape are being added up though the operation $add$ and results in $C$. When $C$ is invoked by a client session, $A$ and $B$ will be evaluated and sent to $add$. When $add$ received both $A$ and $B$, it will perform addition and output the result to the edge connecting to $C$. $C$ will be returned to the client session.

### 3.2.1 Operation node categories

Operation nodes in a computation graph is separated into a few categories.

**Element-wise mathematical operations** These operations correspond to element-wise operations between tensors, such as $add$, $sub$, $Div$.

**Array operations** These operations concern the manipulation of tensor properties, such as shape and rank. It is also possible to extract tensor data through slicing and splitting.

**Matrix operations** At the heart of TensorFlow is matrix operations. These operations perform linear algebra operations such as $matmul$ for matrix matrix multiplication.
Stateful operations  These operations concern the manipulation of stateful nodes (i.e. Variables). Example of these includes `assign` and `assign_add` which are used to update variables.

Queue and synchronization operations  These operations concerns stateful data nodes in a graph such as `enqueue` and `dequeue`. A call to `dequeue` returns a tensor with a predefined shape from the queue to the downstream node.

Control flow operations  These operations add a relationship between disconnected nodes in such a way that a dependency relationship is established while no data flows between two operation. One typical example is to explicitly perform an update to a variable before starting computation.

### 3.2.2 Device placement

![Device placement diagram](image)

Figure 3.2: Illustration of two tensors with the same shape being placed on the CPU, which are connected to a matrix multiplication operation on a GPU.
Listing 3.1: Computing a matrix multiplication where A and B are stored in CPU while operation is done on GPU

```python
with tf.device('/cpu:0'):
    A = tf.placeholder(dtype=tf.float64, shape=[3,3])
    B = tf.placeholder(dtype=tf.float64, shape=[3,3])

with tf.device('/gpu:0'):
    C = tf.matmul(A, B)

with tf.Session() as sess:
    a = np.random.random([3,3])
    b = np.random.random([3,3])
    ret = sess.run(C, feed_dict={ A: a, B: b })
```

Figure 3.2 illustrates a computation graph when nodes of a graph are placed on different devices. A and B are two tensors being placed on a CPU and they serve as the input sources of a matrix multiplication operation which is done on a GPU. Since the nodes are not placed on the same device, a send node is added to each of the tensors and a receive node is added to the matrix multiplication operation. Listing 3.1 illustrates how a graph is created and executed in Python 3, together with the placement code. The tensors A and B are supplied by the client session through dictionary feeding and are sent to the implicit sending nodes. They will be transported to the receiving node, which attaches to the matmul operation on GPU. Finally the operation is executed on the GPU.

### 3.3 Memory management

By default, TensorFlow allocates almost all available GPU memory during initialization to avoid memory fragmentation. It is possible to initially allocate a portion of the memory during initialization and grow the size of allocation if needed. However to avoid fragmentation, allocated memory will never be freed until the end of a session. In terms of CPU host memory, it uses a `malloc` substitute called `jemalloc` which aims to reduce memory fragmentation and provide scalable concurrency support by improving allocation pattern and optimizing dirty page purging. `jemalloc` is integrated into the `libc` of FreeBSD.
3.4 Queues and dataset

3.4.1 Queues

Queue is a special stateful node which contains a list of tensors. Other computation nodes interact with a queue through `enqueue`, to put in; and `dequeue`, to extract one one or more tensors. Queues are originally intended for pre-fetching data from disks, which is a relatively expensive operation comparing to in-memory data transfer. The intention is to prevent disk I/O from becoming a bottleneck such that the computation graph can continuously consume input tensors for computation so as not to leave the devices idle. When one creates a queue, the shape of each tensors, data type and capacity must be specified. TensorFlow queues support a tuple of tensors, meaning that each entry in the queue can consist of multiple tensors. When a queue has zero element, a `dequeue` operation becomes blocking, until elements are enqueued into the queue. Similarly when a queue has a number of elements which reaches the maximum capacity, an `enqueue` operation becomes blocking, until one or more elements are dequeued.

There are four types of queues in TensorFlow:

- RandomShuffleQueue
- PriorityQueue
- FIFOQueue
- PaddingFIFOQueue

`RandomShuffleQueue` is a queue which returns a random element. `PriorityQueue` returns elements according to a priority. Both enqueue and dequeue operations of a priority queue have to include an additional 64 bit scalar at the beginning of a tuple entry which indicates the item’s priority. `FIFOQueue` is the simplest queue in TensorFlow which returns elements in a first-in-first-out order. Similar, `PaddingFIFOQueue` returns elements in a first-in-first-out order except that allows items with dynamic tensor shape.

To facilitate pre-fetching of data, TensorFlow provides a special tool called `QueueRunner`. Threads spawned by a `QueueRunner` will each repeatedly invoke an enqueue operation as a Python thread.
Figure 3.3 illustrates a number of queue runners continuously pushing in tuples which contain two tensors $A$ and $B$ into a queue; and a computation operation which depends on two tensors from the queue. In this example, tensor $A$ is multiplied by $x$, which is a scalar and is added to another tensor $B$, where $A$ and $B$ are obtained through a dequeue operation. Queues in Tensorflow are implemented on CPUs and all enqueue and dequeue operations will be executed on CPU while data in the queue resides in host memory.
3.4.2 Dataset

Dataset is a recent addition to the TensorFlow input pipeline. The dataset API is divided into two components: dataset and iterator. A dataset represents the source of data, which can be in memory or on disk. Transformation can be applied on the fly while defining a dataset node. Data in a dataset is consumed by an iterator, which returns the next item from the dataset. The API is currently in active development.

3.5 Computation Kernels

Operations in TensorFlow are powered by computation kernels in the TensorFlow runtime. These kernels are typically optimized for different devices. Many numerical kernels are supported. Eigen, a C++ template only numerical library is used to generate computation code for devices. Apart from Eigen, some GPU kernels (such as matmul) are supported by cuBLAS, a BLAS library developed by NVIDIA for performing BLAS operations on GPU; and cuDNN, which is used for convolution kernels for deep neural nets operation. When TensorFlow is used in CPU only environment, it is possible to use Intel MKL for computation kernels which dramatically speed up numerical computation performance on Intel CPUs.

3.6 Distributed TensorFlow

In distributed TensorFlow, a TensorFlow cluster is created and it refers to a set of jobs that together participate in executing a computation graph on TensorFlow. Typically, each job contains one or more tasks and each tasks are pinned to a respective computation node. A common idiom in distributed training in TensorFlow is to separate jobs into worker and parameter server. Parameter server is a special job dedicated to variable storage and does not perform any computation. Workers on the other hand perform computation and update variables on the parameter server. Operations can be explicitly pinned to a device of a task the same way as how operations can be pinned on a device in a single node cluster.

The two most commonly used approaches for distributed computation on TensorFlow are:
1. In-graph replication
2. Between-graph replication

Figure 3.4: A client sends in a graph through a session and dispatch operations to different worker nodes.

### 3.6.1 In-graph replication

In-graph replication is also known as model parallelism. When an extremely large neural network model cannot fit onto one device, it is possible to distribute operations to different devices across different worker nodes. Figure 3.4 illustrates an in-graph replication setting with two workers and one parameter server. Each node is responsible for one task of a job. Given sufficient resources, it is possible to run multiple tasks on the same machine. One particular client creates a graph and create a session to one particular worker to send in the graph. That particular server processes and distributes the operations of the graph to different tasks on different nodes.

Listing 3.2: Pinning operations to different workers with in-graph replication

```python
# pin variables to parameter server
with tf.device('/job:ps/task:0/cpu:0'):
    w = tf.get_variable(...)
    b = tf.get_variable(...)
```
In Listing 3.2, a loop is used to pin operations to each worker task and gradually create a graph. When the cluster is ready a client can connect to any worker, create a session and send in the graph, then invoke the operation through the session. When the operation is invoked, its upstream operations which are being placed on other computing nodes will also be executed. The approach implies that all the nodes within the cluster share the same graph.

Even though in-graph replication is extremely simple as illustrated in Listing 3.2, it is rarely used in the real world. This is due to the fact that all computation nodes share the same graph. As the problem size grows, the size of the computation graph increases as well. One particular problem, is that computation graph in TensorFlow is represented by Protobuf, a serialization library developed by Google. Protobuf cannot handle data larger than 2GB. Another problem is that, with a huge graph where operations are all placed across different devices, communication pattern becomes extremely complicated and communication cost becomes abandon as the client has to dispatch operations across different nodes.

### 3.6.2 Between-graph replication

Between-graph replication on the other hand does not create the same graph for each and every nodes in the cluster. Instead, each workers create a similar graph with partial overlapping, typically with the parameter server. Since the model is data driven in the sense that each
computing node executes a "similar" graph with same computation operations repeatedly with different input data, it is also known as data parallelism.

![Diagram](image)

**Figure 3.5: Each node creates a server and worker nodes additionally create a similar graph where the portion concerning parameter server overlaps. The clients create a session and send in the graph and invoke the operations repeatedly.**

**Listing 3.3: Pinning operations to different workers with between-graph replication**

```python
# print variable x and y on parameter server
with tf.device('/job:ps/task:0/cpu:0'):
    x = tf.get_variable(...)
    y = tf.get_variable(...,
                        initializer=tf.zeros_initializer())
```

```python
# if node that executes this code is parameter server, block
# else create its own graph, end of overlapping section
if job_name == "ps":
    server.join()
else:
    # read from data source and populate job queue
    with tf.device('/job:worker/task:%d/cpu:0' % task_index):
        queue = tf.FIFOQueue(...)
    # ..code to enqueue from file
    # create queue runner here...
```
# dequeue operations
a, b = queue.dequeue()

# create computation operations for this worker
with tf.device('/job:worker/task:%d/gpu:0' % task_index):
    new_y = tf.add(tf.matmul(a, x), b)
    # update variable in parameter server
    update_y = tf.assign_add(y,
        new_y,
        use_locking=True)

A cluster illustrated in Figure 3.5 can be created by the code snapshot in Listing 3.3. The code demonstrates between-graph replication in the sense that only the graph construction up to parameter server is the same across all computing nodes. If a computing node is the parameter server it immediately joins the server, which blocks forever. The parameter server will have no knowledge on any operations except for those that are already pinned to the device. If a node is not the parameter server, it proceeds to construct its "own" graph by specifying the job name and task index. Since the worker only construct the computation graph for itself and parameter server, it will have no knowledge of any other operations on other workers. In this example, all the workers together compute \( y = y + (a \times x + b) \), where \( a \) and \( b \) are tensors extracted from a local input queue, and \( x \) and \( y \) are global variables on the parameter server. With between-graph replication, each workers participate in their own computation by establishing sessions to themselves and repeatedly execute the operations. The approach is scalable comparing to in-graph replication in the sense that each workers own their small graph instead of sharing a global graph, thus communication is only restricted to between the parameter server and worker. Since each worker performs their independent computation cycle, there is no staling in the sense that a worker does not need to wait for workers with in-complete upstream operations. This reduces idle time and eliminate the need to coordinate between workers.

### 3.7 Communication modules

TensorFlow supports three major communication methods, namely gRPC, InfiniBand RDMA and MPI. In this section we describe their implementation.
3.7.1 gRPC

Figure 3.6: gRPC clients requesting RPC to gRPC server and getting responses.

*gRPC* is an open-source cross-platform RPC framework developed by Google. It is based on *HTTP/2.0* for bi-directional network transfer and *ProtoBuf* for serialization [23]. The framework is the default communication protocol used by distributed TensorFlow. Figure 3.6 shows the server client model of gRPC between clients and servers.

3.7.2 GPUDirect RDMA

Figure 3.7: RDMA between two GPU memory through InfiniBand.
GPUDirect RDMA on TensorFlow is implemented in the GDR module inside the TensorFlow runtime [46]. It uses gRPC for administration, such as to setup connections, and when possible, uses GPU Direct RDMA to transfer tensors to and from remote GPU memory. Figure 3.7 illustrates how the transfer of tensors through GPU Direct RDMA bypass host memory and CPU.

There are two types of messages generated by this module: read and write. A single read instruction will retrieve a tensor from a remote GPU memory, while a write will invalidate the tensor by releasing the remote buffer. gRPC is used to transfer remote buffer information, or metadata.

### 3.7.3 InfiniBand RDMA

TensorFlow runtime contains a communication module for InfiniBand RDMA with VERBS [45]. Similar to GPUDirect, gRPC is still responsible for administrative workload, such as to setup the connection and exchange metadata.

During the server setup, an RDMA manager is created to manage low-level RDMA components such as RDMA channel and RDMA adapter, an RDMA rendezvous manager is created to oversee send/recv operations between servers. Following the distributed TensorFlow design philosophy, the operation is passive, i.e. merely placing a tensor in the local out-going table. It is the receive operation that actually initiates the tensor transfer. Since TensorFlow dynamically allocate memory for tensors, it is difficult to pin memory blocks for RDMA. Therefore, the manager pre-allocate a large memory block instead and allocate tensors from there. The pool will be registered and thereafter each tensors allocated will be at a registered address, which allows RDMA to be performed.

Tensor transfer can be done directly from source to sink without memory copy for supported devices such as CPU or GPU which supports GPUDirect. In case a tensor can not be accessed by RDMA, it will be serialized to a TensorProto in a buffer, registered, transferred and de-serialized on the receiving side.
3.7.4 MPI

The MPI module in TensorFlow takes over the responsibility of launching and transferring data between processes on different nodes [25]. Similar to VERBS, gRPC is used for administration while MPI functions are responsible for transfer of tensors. For each process, a separate thread is launched which loops through a set of operations. Once a request of a tensor is received, two callbacks are created. One to request the tensor and one to handle received data. The request is pushed into a queue and will be sent when the queue is serviced by the MPI thread.

The MPI thread probes for incoming requests. Once a request is received, the request is forwarded to TensorFlow and once the tensor is found it is placed to a sending queue. When the queue is serviced by the MPI thread it will be sent non-blockingly. When a tensor is received, the callback function which handles received data will be launched.

All send and receiving functions are non-blocking. Since gRPC is address based instead of rank based, the module creates a mapping between MPI process ID and TensorFlow process name to facilitate MPI communication between processes.
Chapter 4

Methods

In this chapter, we illustrate the implementation of two commonly used algorithms that are fundamental to many applications: tiled matrix matrix multiplication and conjugate gradient solver. We express the algorithms with computation graphs in a distributed way. We also give a general overview on communication performance of TensorFlow through various communication protocols.

4.1 Measurement of communication bandwidth

We show the communication performance of distributed TensorFlow by a simple setup with one parameter server and one worker, located on two separated nodes. A variable which contains a vector of 32-bit integers, of which the total size is equivalent to a pre-defined number of megabyte is created and pinned to the parameter server, another vector variable with the same size is pinned to the worker. An operation which performs assign add to the variable on the parameter server using the variable on the worker is created. When the operation is invoked, the variable on the worker is pushed for update in the parameter server. Therefore, when the operation is invoked a transfer of defined number of megabyte will be created. To provide a coarse-grained measurement of the transfer bandwidth, the time taken to complete the operation is measured. We detail the measurement result in MB/s and test for the supported communication protocol by TensorFlow: MPI, InfiniBand Verbs RDMA and gRPC over TCP. We do note that the test scheme is
4.2 Expressing algorithms in TensorFlow graph

It is obvious from chapter 3 that for a computation problem to be executed on TensorFlow platform it must be reformulated as a computation graph. In order to avoid synchronization, the graph ideally should be organized in such a way that all computation can be executed just by invoking a sink operation. This way the operation can be repeatedly and independently executed by computing nodes.

4.2.1 CG Solver

Conjugate gradient method is an algorithm that gives a numeric solution for a system of linear equations, namely \( Ax = b \), on the condition that the matrix is symmetric and positive-definite. CG is commonly implemented as an iterative solver. Instead of computing the solution analytically with direct methods, these algorithms approximate and iteratively improve the solution. CG in particular, is applicable to sparse linear systems that are too large to be handled by direct methods. The accuracy of solution is often determined by the residual: \( r = b - Ax \) and normally a measure, such as a norm is applied. One common measure is \( \eta = \frac{||r||}{||b||} \).

Algorithm

Algorithm 1 illustrates the iterative CG algorithm. The algorithm is divided into four parts:

1. Computation of alpha
2. Computation of \( x \) and \( r \)
3. Computation of beta
4. Computation of \( p \)
Algorithm 1: Iterative CG algorithm

Both the computation of alpha and beta require a single point of accumulation. The variables $p$, $x$, $r$, $\delta$ also depend on the computation result of the previous iteration. This implies that the algorithm has three synchronization points. Between the synchronization points it is possible to distribute computation. Figure 4.1 shows how the variables in the algorithm is split horizontally into blocks.

Firstly, the computation of $q \leftarrow Ap$ can be done individually by workers each containing a block of $A$ and a vector $p$. The matrix vector product results in a partial $q$ vector which corresponds to the portion of the task as illustrated in Figure 4.2. By performing a dot product of the partial vector $p$ and $q$ a partial alpha is derived. Due to the nature of dot product, one dot product is equivalent to the sum of multiple dot products where the two vectors are split horizontally into subvectors as illustrated in Figure 4.3.

In a distributed setting, this can be implemented in such a way that each task contains a block of $A$ matrix and a $p$ vector. The task is sent to different computation workers which individually compute their a part of $q$ and $\alpha$. Finally the resulting $q$ and $\alpha$ is sent back to the master worker for assembly.

The second part of the algorithm involves updating solution $x$ and residual $r$ as well as computation of new $\beta$ which is based on previous and updated $\delta$. The computation of new $x$, $r$ requires the result of $x$, $r$, $\delta$.
from the previous iteration and $p$, $q$ and $\alpha$ from the current iteration. This can be implemented such that each task contains a subset of the vectors and the scalar $\alpha$. Each workers after obtaining the share of task can individually compute the updated partial $x$ and $r$. Each of these computation involves a vector addition and scalar vector multiplication. With a partial $r$, the worker can additionally compute a partial $\delta$ which will later be used for computation of $\beta$. The updated $x$, $r$ and $\delta$ will be sent back to the master for combination. The individual $\delta$ computed by the workers will be accumulated. With the updated $\delta$ the $\beta$ value can also be updated.

In the final stage of the algorithm, vector $p$ is updated. This follows a similar approach for the update of $x$ and $r$ where the sub vectors of $p$ and $\beta$ are sent to individual workers and collected after computation for assembly.

Expression as computation graph

Since the algorithm involves three synchronization, the computations cannot be performed all together. Therefore, three separate disconnected sub-graph will be created for each phase of computation. We divide the tasks into two categories: parameter server and worker. The client session on the parameter server node continuously prepare tasks to enqueue and at the same time dequeue from the result queue to
obtain computation result and assemble them in-memory in a session.

Figure 4.4 illustrates how the computation graph is organized. One computation node, arbitrarily called parameter server is dedicated for queue operations. On the parameter server there are two FIFO queues for each stage, one for distributing tasks and one for gathering results. Each task is represented by a tuple of three tensors: index, partial $A$ and $p$. Index represents the index of a block such that the work can be executed by any worker and later re-assembled by the client. A dequeue operation will return the three tensors in the tuple.

To execute the graph, the clients repeatedly invoke the enqueue operation, which eventually enqueue a tuple of results, containing the index, partial vector $q$, and partial $\alpha$, into the result queue. Since the three results will be re-used during the execution on the same graph, the three values will be stored in local variables. The execution of operations in the graph automatically invoke the dequeue and assignment through explicit control dependency. Due to its complexity, it is not illustrated in the diagram, however, the concept will be demonstrated in later steps.

The enqueue operation invokes operation "matmul 1". "matmul 1" has an explicit dependency to operation "assign", which executes "matmul 0" and assign result to a local variable called $q_{local}$. The operation performs matrix multiplication of $A_p$. The reason to store the
intermediate result in a variable instead of executing the operation directly is that $q_{local}$ is required by upcoming operations. If "matmul 0" is connected to the graph directly, the operation will be repeatedly executed. Therefore it is desirable to have the result stored in a variable for later re-use. Since the graph becomes disconnected, explicit dependency is specified such that "matmul 0" will always be invoked immediately before "matmul 1" to ensure $q_{local}$ contains the up-to-date value for consumption.

Operation "matmul 1" performs dot product of $p_i$ and $q_i$ which results in partial $\alpha$. The operation on one hand extract the value from variable $q_{local}$ and on the other hand obtain the sub vector $p_i$ through a gather operation, which extract part of a tensor by given specifications. The specification is given by an index. Finally, all the required computation to perform the enqueue are executed and a tuple of results, including index, $q_{local}$ and partial $\alpha$.

The client session continuously dequeue from the result queue and perform assembly of the vectors and alpha locally until all results are received. When $\alpha$ is computed, the client session breaks the problem into tasks again and enqueue the required data into the task queue for the next phase.

Figure 4.5 illustrates the computation of the second phase, where $x, r$ and $\delta$ are updated. The graph is rather complicated. In this case, each task in the task queue contains a tuple of tensors: index, $x, r, p,$
Figure 4.4: Computation graph for the first phase of CG solver. A task is dequeued from the job queue and assigned to variables. The operations are executed such that dependencies are satisfied. Finally the resulting partial \( q \) and \( \alpha \) is enqueued to the resulting queue together with the job index.

\( q, \alpha \). Except for \( \alpha \), all tensors are partial vectors for local computation. When a dequeue operation is invoked, all tensors are copied to a local variable on the worker. Again, for simplicity, if the state of those tensors are not updated this step is omitted from the illustration.

Similar to the computation of partial \( \alpha \), the computation is done by the client invoking the enqueue operation from the session. The operation invokes the computation of a partial \( \delta \), which invokes its dependencies, the update of \( x \) and \( r \), and in turn invoke the dequeue operation. The computation begins by a dequeue and assignment where dequeued tensors are stored locally in a variable. \( x \) and \( r \) will be updated by multiplying the vector \( p \) and \( q \) with \( \alpha \) respectively and accumulate to itself through the assign add operation. Finally, the updated partial \( \delta \) is computed by a dot product of \( r \). The updated vectors \( x, r \) and \( \delta_{new} \) will be enqueued to the result queue.
Figure 4.5: Computation graph for the second phase of CG solver. A task is dequeued from the job queue and $x$, $r$, $\delta$ are computed in such a way that dependencies are satisfied. Finally, the results together with the job index are enqueued to a result queue.

The client session continuously dequeue from result queue and accumulate for final $\delta$ and compute $\beta$. When $\beta$ is computed the client again breaks the problem into tasks and enqueue to the task queue for the next phase.

Finally, $p$ has to be updated. This is relatively simple and is illustrated in Figure 4.6. Again, client dequeue required data from the task queue and perform computation. The client session repeatedly execute enqueue operation which invokes the add operation. The operation accumulate both $p$ and $r$, after scaled by the scalar $\beta$. The result is enqueued to the result queue together with the index in a tuple. The client session dequeue the results and assemble the complete updated $p$ vector. After this phase, the iteration step is over.
4.2.2 Optimized CG Solver

During the development of the CG solver in section 4.2.1, we noticed that the performance is below expectation. The performance analysis will be detailed in chapter 6. Upon investigation, it was suspected
that the issue is due to large amount of data movement between iterations. The design decision was made due to the requirement for over-decomposition. Over decomposition is required to solve larger problems on devices with relatively small memory. However, if we are able to relax this assumption, it is possible to reduce data movement dramatically. In this sense, each device will be responsible for only their dataset and only data required for synchronization will be transferred. For this reason, we developed an optimized version of the solver which is able to take advantage of some form of data locality.

Expression as computation graph

![Computation graph](image)

Figure 4.7: Computation graph for the initialization of variables from dictionary. Values are placed in placeholders through dictionary feeding and assignment operations are executed. After the assignment each workers sign into a barrier.

Similarly, there are three computation phases in the optimized implementation. However, instead of each worker obtaining a task from the parameter server in each phase, each of the workers are provided with their share of “task” by initializing the variables through dictionary feeding. Figure 4.7 shows how such initialization is done. The reason for not initializing the variable implicitly with variable initializer is to overcome the 2GB size limit imposed by ProtoBuf. After the initialization, each worker signs into an implicit barrier. A barrier is constructed by creating one queue for each worker. When a worker signs into a barrier it enqueues one value to every queue, then immediately ask to
dequeue that many number of elements from its own queue. The only way that the worker can exit that blocking dequeue operation is if all other workers have also enqueued that many values to its queue.

Figure 4.8: Computation graph for the first phase of optimized CG solver. \( q \) is computed by performing a matrix vector product then a dot product of \( p_{local} \) and \( q \) is computed and sent to the parameter server. The parameter server receives all individual partial \( \alpha \) values and compute the updated \( \alpha \) value. The value is then distributed to all workers by populating a queue with copies of the new \( \alpha \).

In addition, instead of computing parameter values in the parameter server client session, we construct them as graphs on the parameter server worker. For this reason, all dequeued values on the parameter server only go through the graph instead of going to the client session. This avoids the inefficient communication between the client session and the graph. Additionally, this allows better performance optimization by TensorFlow as computation is done with TensorFlow instead of Python session. All operations are chained up in such a way that each client and parameter server only invokes one operation through session run per iteration. The final result can be extracted by reading the value of the \( x \) variable in their graph. The goal is to rely on the
client session for computation as little as possible and synchronization is implicitly done through blocking dequeue and enqueue operations.

![Computation graph for the second phase of optimized CG solver](image)

Figure 4.9: Computation graph for the second phase of optimized CG solver. Each workers compute their own $r$ and $x$. With the updated $r$ they compute their partial $\delta$ and send them to the parameter server. The parameter server computes the final $\delta$ and also compute $\beta$ with both $\delta_{\text{new}}$ and $\delta_{\text{old}}$. $\beta$ is distributed to the workers by populating a queue.

Each workers individually update their local $q$ by multiplying $A$ and $p$. With the new $q$ it is possible to compute their part of alpha by computing the dot product of $p_{\text{local}}$ and $q$ where $p_{\text{local}}$ represents their share of the $p$ vector. The dot product is enqueued into a partial value queue on the parameter server as in Figure 4.8. At the same time the parameter server dequeues a number of elements from the partial alpha queue where the number of dequeue equals the number of workers. The partial alphas computed by individual workers are summed up and used as a divisor of $\delta$ to compute the updated $\alpha$. Finally a number of copies of the updated alpha is enqueued to a alpha queue. The number of enqueue is equal to the number of workers. Each and every worker will ask to dequeue one element from the alpha queue. When the dequeue operation is successful the worker would update their
local copy of $\alpha$ by assignment. With the updated $\alpha$ value the workers can proceed to the next phase.

![Computation graph for the third phase of optimized CG solver.](image)

Figure 4.10: Computation graph for the third phase of optimized CG solver. Each workers dequeue a $\beta$ value from the beta queue on the parameter server and update their $p_{local}$. The send in the updated $p_{local}$ together with their worker index to the parameter server. The parameter server receives all the partial $p_{local}$ vectors and reshape them to form the new $p$. The updated $p$ is then distributed to all workers through a p queue.

Each workers individually compute their share of $x$ and $r$ with the updated $\alpha$ in Figure 4.9. When the computation is done a partial $\delta$ is computed by performing dot product of $r$ with itself. The value is enqueued into a partial delta queue on the parameter server. The parameter server dequeues all the values in the queue, sum them up and assign the result to $\delta_{new}$. The value is divided by the previous delta $\delta_{old}$ to compute the $\beta$ value. A number of copies of $\beta$ is enqueued to a beta queue.

Each workers dequeue from the beta queue as in Figure 4.10. The workers compute the updated $p_{local}$ with their own $r$ and $\beta$. The result is enqueued into a priority queue on the parameter server with their task.
index being the priority such that when the parameter server performs
dequeue many operation the vectors will be ordered by task indexes.
Since the vectors are already in order, a merge can be done by simply
reshaping the tensors. Copies of the reshaped $p$ is enqueued to the
$p$ queue and workers will dequeue one copy and update their own
variable.

4.2.3 Tiled matrix multiplication

Dense matrix multiplication is the foundation of many algorithms. A
standard algorithm that performs a multiplication of two matrices $A$
of size $n \times m$ and $B$ of size $m \times p$ which results in $C$ with size $n \times p$ is
illustrated in algorithm 2. The algorithm performs element-wise and
multiplication and summation for elements in rows of $A$ and columns
of $B$.

```
for $i := 1 \rightarrow n$ do
    for $j := 1 \rightarrow p$ do
        $sum \leftarrow 0$
        for $k := 1 \rightarrow m$ do
            $sum \leftarrow sum + A[i][k] \times B[k][j]$
        end
        $C[i][j] \leftarrow sum$
    end
end
```

Algorithm 2: Matrix multiplication of matrix $A$ of size $n \times m$ and $B$
of size $m \times p$ resulting in $C$ with size $n \times p$.

Tiled matrix multiplication is one technique in parallelizing large
dense matrix multiplication. A matrix divided into smaller blocks and
instead of element-wise multiplication and accumulation, block-wise
multiplication and accumulation is performed, where each operation
is a dense matrix multiplication and accumulation of blocks. The for-
mulation of matrix blocks are follows the same as in algorithm 2, but
$A[i][k], B[k][j]$ and $C[i][j]$ are blocks instead of elements.

Figure 4.12 details the steps of how two $4 \times 4$ matrices are decom-
posed into blocks of $2 \times 2$ and multiplied. In this particular case, the
multiplication process is decomposed into $8$ ($2^3$) matrix multiplication,
plus accumulation of the blocks.
Figure 4.11: Computation graph for dense matrix multiplication.

**Expression as computation graph**

This particular algorithm is decomposed in such a way that there are no dependencies for the computation of each blocks as they can be accumulated in different orders. Therefore the computation graph is relatively simple and tasks are only required to perform one dense matrix multiplication.

Figure 4.11 illustrates how a computation graph is formed. In each iteration, the client session invokes the enqueue operation, which in turn invokes a matrix multiplication, of which the data source is obtained through dequeueing from the task queue. After performing the multiplication, the resulting matrix together with the job index will be pushed into the result queue. The client session of the parameter server contains two threads, one to decompose the two large matrices into blocks and divide them into tasks as illustrated in Figure 4.12. The blocks, together with their respective position (indexes) in the resulting matrix are pushed into the job queue. At the same time, the other thread repeatedly dequeue from the result queue. With the resulting block the client thread accumulates the result to its respective block in memory according to the indexes. When all the tasks are completed and accumulated the thread re-shape the matrix blocks to form the final resulting matrix. All computations are done in single precision (32bit floating point operation).
\[
A = \begin{bmatrix}
a_{11} & a_{12} & a_{13} & a_{14} \\
a_{21} & a_{22} & a_{23} & a_{24} \\
a_{31} & a_{32} & a_{33} & a_{34} \\
a_{41} & a_{42} & a_{43} & a_{44}
\end{bmatrix} = \begin{bmatrix}
A_{11} & A_{12} \\
A_{21} & A_{22}
\end{bmatrix}
\]

where,

\[
A_{11} = \begin{bmatrix}
a_{11} & a_{12} \\
a_{21} & a_{22}
\end{bmatrix}, \quad A_{12} = \begin{bmatrix}
a_{13} & a_{14} \\
a_{23} & a_{24}
\end{bmatrix}, \quad A_{21} = \begin{bmatrix}
a_{31} & a_{32} \\
a_{41} & a_{42}
\end{bmatrix}, \quad A_{22} = \begin{bmatrix}
a_{33} & a_{34} \\
a_{43} & a_{44}
\end{bmatrix}
\]

\[
B = \begin{bmatrix}
b_{11} & b_{12} & b_{13} & b_{14} \\
b_{21} & b_{22} & b_{23} & b_{24} \\
b_{31} & b_{32} & b_{33} & b_{34} \\
b_{41} & b_{42} & b_{43} & b_{44}
\end{bmatrix} = \begin{bmatrix}
B_{11} & B_{12} \\
B_{21} & B_{22}
\end{bmatrix}
\]

where,

\[
B_{11} = \begin{bmatrix}
b_{11} & b_{12} \\
b_{21} & b_{22}
\end{bmatrix}, \quad B_{12} = \begin{bmatrix}
b_{13} & b_{14} \\
b_{23} & b_{24}
\end{bmatrix}, \quad B_{21} = \begin{bmatrix}
b_{31} & b_{32} \\
b_{41} & b_{42}
\end{bmatrix}, \quad B_{22} = \begin{bmatrix}
b_{33} & b_{34} \\
b_{43} & b_{44}
\end{bmatrix}
\]

\[
AB = \begin{bmatrix}
A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\
A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22}
\end{bmatrix}
\]

Figure 4.12: Representation of two \(4 \times 4\) matrices \(A\) and \(B\) in blocks.
Chapter 5

Experimental setup

In this chapter, we explain the experiment setup that evaluates the performance of algorithms in chapter 4 and detail the platform used. We perform tests to evaluate our CG solver, tiled matrix multiplication and also micro-benchmark communication performance.

We conduct the tests on Tegner, a cluster at KTH PDC. Each node has two Intel E5-2690v3 Haswell processors, 512 GB of RAM and one Nvidia Quadro K420 GPU which supports NVIDIA compute 3.0 and provides 1 GB of RAM. The operating system is CentOS 7.4.1708. The nodes are connected by EDR InfiniBand network [44]. Since Tegner is a heterogeneous system, we try to ask for nodes with same configuration by blacklisting nodes with certain properties. We compiled TensorFlow 1.7 with support of Python 3.6, CUDA, OpenMPI and InfiniBand. We used CUDA 9.1, cuDNN 7.0.5 and OpenMPI 3.0. All the tests are executed with distributed TensorFlow and the communication protocol used is InfiniBand verbs. The compiler used is GCC 6.2.0.

Since the optimized CG solver outlined in section 4.2.2 of chapter 4 does not allow over decomposition, more powerful GPUs with more memory are needed to solve the selected problem sizes. For this reason we performed that set of experiments on Kebnekaise, a system in High Performance Computing Center North (HPC2N) at Umeå. Each node has two Intel Xeon E5-2690v4 processors and 128 GB of RAM. The operating system is Ubuntu Xenial (16.04 LTS). Each node has two Nvidia K80 GPUs where each K80 card contains 2 GK210 GPU engines and each engine contains 12 GB of RAM, which means each node has 4 available GPUs. The nodes are connected by FDR InfiniBand network [29]. We compiled TensorFlow 1.7 with support of Python
3.6.4, CUDA, OpenMPI and InfiniBand. We used CUDA 9.1.85, cuDNN 7.1.2.21 and OpenMPI 2.1.2. All tests are executed with distributed TensorFlow with infiniBand verbs enabled. The compiler used ins GCC 6.4.0.

5.1 Communication Performance

We evaluate communication performance of the default gRPC, MPI and InfiniBand Verbs with our benchmarking tool discussed in section 4.1 of chapter 4 on Tegner. We allocate two nodes and dedicate one for parameter server and one for worker. Due to the limitation of this setup, the result is coarse-grained and potentially noisy, for that reason we repeat the operation for 50 times. The assign add operation is invoked 50 times by the client session on the worker node, varying the data transfer size in megabyte. We vary the transfer size for each invocation to be 1, 2, 4, 8, 16, 32, 64, 128, 256 megabytes respectively. The starting time and ending time are recorded before and after the 50 invocations and a transfer rate in MB/s is computed.

5.2 CG solver

We evaluate the performance of the solver with different problem sizes: \(16384 \times 16384\), \(32768 \times 32768\) and \(65536 \times 65536\) on Tegner. To fill up the GPU, each of the blocks responsible should have size equivalent to \(4096 \times 4096\). Therefore the block size for dividing matrix \(A\) for the three problem sizes are: \(1024 \times 16384\), \(512 \times 32768\) and \(256 \times 65536\) respectively. The number of computing nodes used is 2, 4, 8, 16. We run 10 iterations and measure the performance of the solver with runtime in seconds. We run the tests for each configuration 6 times, with the first one being the warm-up. The max, min and median is reported. The \(A\) matrix is initialized with random numbers between 0 and 1 and is flipped such that it becomes symmetric. We do however note that the convergence of the algorithm is out of the scope this study since our primary concern is scalability of the algorithm.
5.3 Optimized CG solver

We evaluate the performance of the solver similarly as in section 5.2 on Kebnekaise. However since the implementation does not allow over decomposition, the workload of each worker is determined by the problem size as well as the number of workers. We run strong scaling tests with different problem sizes: 16384 × 16384, 32768 × 32768 and 65536 × 65536 with 2, 4, 8 and 16 GPUs. We run 500 iterations and repeat each tests 6 times where the first being the warm-up. The max, min and median in seconds is reported. Since each node has multiple GPUs and each GPU has two engines, we run multiple instance of TensorFlow per node and only expose each instance to only one GPU engine by manipulating the \texttt{CUDA\_VISIBLE\_DEVICES} environmental variable.

5.4 Tiled matrix multiplication

We study the performance of the setup with different problem sizes: 16384 × 16384, 32768 × 32768 and 65536 × 65536 on Tegner and vary the number of computing nodes used to 2, 4, 8 and 16. The tile size is fixed at 4096 × 4096 due to the limitation of memory on the GPU. InfiniBand Verbs is used as the communication protocol. We run the tests 6 times, with the first one being the warm-up execution. Gflops/s is used as the figure of merit for the measurements. The two matrices to be multiplied are initialized by random numbers between 0 and 1. The number of flops is computed with $2MNL - ML$, where $M, N, L$ are boundary sizes of the matrices. The final value is derived by $\frac{(2MNL-ML)}{\text{seconds}} \times 10^9$. 
Chapter 6

Results

In this chapter, we discuss the results obtained in chapter 5 on selected systems. We analyse the performance results between the two tested algorithms and attempt to explain the reasons behind.

6.1 Communication performance

We run the tests in section 5.1 of chapter 5 on Tegner. We run the operation which invoked the transfer of data for 50 times and varied transfer size for each invocation to be 1, 2, 4, 8, 16, 32, 64, 128, 256 megabytes respectively.

Figure 6.1 shows the measurement result of communication performance. It is very obvious that gRPC was not able to take advantage of the high speed interconnect due to the limitation of underlying transport layer protocol as gRPC is based on HTTP/2.0 which in turn is based on TCP. The bandwidth saturates at around 100 MB/s. InfiniBand verbs RDMA on the other hand, provides extremely high bandwidth. The bandwidth scales from around 1000 MB/s to around 6000 MB/s where it saturates when the transfer size is increased to 256 MB. A surprising result from the measurement with MPI is that the bandwidth saturates at between 500-1000 MB/s, which is extremely low when comparing to the result from InfiniBand verbs RDMA. Even though both method should eventually be based on InfiniBand transfer, the implementation of TensorFlow with verbs further supports RDMA of tensors between nodes, which allows a request to bypass the remote node’s system and read from memory directly. Another possible factor would be overhead due to the design of the MPI module. Both tensor
requests and data are stored in queues and is only transferred when the queue of the process is serviced by a thread periodically. The InfiniBand implementation however, directly make use of InfiniBand verbs for point to point transfer.

6.2 CG solver

We run the tests in section 5.2 of chapter 5 on Tegner at PDC for problem sizes $16384 \times 16384$, $32768 \times 32768$ and $65536 \times 65536$ on 2, 4, 8 and 16 computing nodes respectively with one extra node as parameter server. Figure 6.2 shows the median runtime of the experiments. The error bar on the histogram denotes the variation (minimum and maximum) during the test runs. For each of those problem sizes, matrix $A$ is decomposed into smaller blocks in tasks with number of elements being equal to $4096 \times 4096$. The result shows that distributing workload to workers provide very little to no scaling at all.

To understand why the performance is so bad, we perform a timeline profiling. Timeline is a tool in TensorFlow that returns metadata and record of a session execution. The profile can be visualized us-
Figure 6.2: Performance of distributed CG solver for different problem sizes on different number of computing nodes, measured in seconds.

Figure 6.3 illustrates the timeline of an overall run. It is immediately obvious that the majority of runtime was wasted on data transfer. Upon zooming in it is clear that the particular idle time was spent at the beginning/end of an iteration. While at the same time the actual computation only takes into account fraction of the entire execution time. Similarly, Figure 6.4 shows zoomed in timeline of the execution of computation of $x$, $r$ and partial $\delta$ of a particular task. Again, receiving the task from the queue took the majority of the running time. However, it can be seen that the computation pipeline itself is efficient with little idle time. It is obvious that data transfer time dominates the execution. Since the algorithm has three synchronization per iteration, the slowest phase dominates and thus eliminate any speedup bring by other phases, if any. Also, apart from the computation of $\alpha$, most of the computation involves only mainly vector computation instead of
dense matrix computation. Therefore it is likely that the workload per tasks are insufficient to keep the pipeline busy. Yet at the same time the amount of data transferred is large enough to congest communication such that the communication time dominates. On the other hand, the number of tasks is only handful. For the case of $65536 \times 65536$ with $256 \times 65536$ per tile only decompose to 256 tasks.

### 6.3 Optimized CG solver

We run the tests in section 5.3 of chapter 5 on Kebnekaise at HPC2N for problem sizes $16384 \times 16384$, $32768 \times 32768$ and $65536 \times 65536$ on $2$, $4$, $8$ and $16$ GPUs respectively with one extra process instance as parameter server.

Figure 6.5 shows the median runtime of the tests in seconds. Comparing to the un-optimized version not only is the runtime performance extremely good, the performance also scales as the number of worker GPU increases. For $16384 \times 16384$ the scaling is moderate. This is likely
Figure 6.5: Performance of optimized CG solver for different problem sizes on different number of computing nodes, measured in seconds. It was impossible to run problem size $65536 \times 65536$ with only 2 and 4 GPUs due to memory requirement.

due to under utilization of the GPUs as the problem size is relatively small comparing to the GPUs’ capacity. For $32768 \times 32768$ the scaling is obvious particularly when increased from 2 to 4 workers. The scaling flats out thereafter when 16 workers are used as expected from strong scaling test. Problem size $65536 \times 65536$ cannot be executed with less than 8 workers due to insufficient memory. Since this implementation does not allow over decomposition, it is impossible to further break down the tasks into small ones. When the number of workers increases from 8 to 16, scaling is observed.

We perform timeline profiling of the solver to compare the differences between the optimized and un-optimized version. Figure 6.6 shows the profiling result of a run with two GPUs and problem size $32768 \times 32768$. It can be seen that the timeline pattern comparing to the un-optimized version is less sparse. It can also be seen that by offloading all parameter computation operations to TensorFlow it is possible to further optimize the execution. The runtime is able to fully utilize many CPU cores for overlapping computation.

Figure 6.7 zooms into to a particular execution phase of a particular
Figure 6.6: Timeline profiling result of an execution with problem size $32768 \times 32768$ and two workers.

Figure 6.7: Zoomed in timeline profiling result of a particular phase of a particular task. It can be seen that the pipeline is rather efficient in the sense that the operations tightly follows each other. In this sense the performance is similar to that of the un-optimized version as the implementation itself is similar. From the profiling it can be observed that performance gain is mostly due to reduction of time spent on data movement and optimization of parameter computation.

### 6.4 Tiled matrix multiplication

We run the tests in 5.4 in chapter 5 on Tegner at PDC for problem sizes $16384 \times 16384$, $32768 \times 32768$ and $65536 \times 65536$ on 2, 4, 8 and 16 computing nodes respective with one extra node as parameter server.
Figure 6.8: Performance of tiled matrix multiplication for different problem sizes on different number of computing nodes, measured in Gflops/s. The dotted line shows the baseline performance of one GPU performing GEMM on two $4096 \times 4096$ matrices with cuBLAS.

Figure 6.8 shows the median performance of the experiment. The error bar on the histogram denotes the variation (minimum and maximum) during the test runs. We additionally performed a test with simple cuBLAS GEMM on two $4096 \times 4096$ matrices for a number of iterations on one GPU. The average Gflops/s is plotted as a reference line to illustrate the baseline performance with least possible overhead.

From the result, it can be seen that with 4 computing nodes and 1 parameter server, the performance is around $3 \times$ of the baseline performance around double than that of the performance when only 2 computing nodes are used. The performance for problem size $65536 \times 65536$ scales particularly well. For both problem size $32768 \times 32768$ and $65536 \times 65536$ saturates at around 1000-1200 Gflops/s. The performance in terms of scalability is very satisfying.
Chapter 7

Discussion and conclusions

In this chapter, we conclude this work by giving a summary of the results and insights achieved by this study and discuss the usability of TensorFlow as a programming model for HPC applications. With the information in-hand, we propose future work and direction where suitable.

7.1 Scientific closure

In this section, we attempt to answer the scientific questions raised in section 1.2 of chapter 1.

How to express HPC problems as graph on TensorFlow? Computation problems are expressed as computation graphs on TensorFlow. Furthermore TensorFlow provides special operations which are stateful and supports various input pipelines. Problems must be decomposed into sub-problems and have to be formulated in various form of matrix multiplications. In a distributed setting, the recommended approach is between-graph replication where multiple different graphs are created for different workers. Clients create sessions to their respective nodes, initialize their own graph and performs repeated operations.

How scalable and performant is TensorFlow? Test result on the performance of the two sample application is two folded. It shows that the implementation of input pipeline is of paramount importance to the performance of TensorFlow. We tested two applications: tiled matrix multiplication and CG solver. The two applications are very different
in nature. One requires synchronization and one does not. The result shows that synchronization and imbalanced workload seriously hamper computation performance as it disrupts the input pipeline. As it was originally designed for deep neural network training, TensorFlow is data driven in the sense that it requires large amount of computation and data flow within workers, parameter server and input source through the computation pipeline. In order to hide the cost of communication, workloads of tasks must be sufficient and the execution of tasks must be asynchronous. At the same time, size of data transferred should be small enough such that it can be hidden by overlapping computation. As demonstrated with our optimized version of CG solver, by properly balancing computation load and communication load it is possible to create acceptable scalability. Given a properly crafted pipeline and large amount of data, TensorFlow performs extremely well on our testing platform. Another observation through developing the two CG solver is that it is important to offload as much computation workload to the TensorFlow runtime as possible to take advantage of TensorFlow’s scheduling optimization. In the optimized CG solver, not only is communication cost between client session and the graph eliminated, execution is optimized to take advantage of multi-core systems.

Can TensorFlow take advantage of HPC specific high performance architectures? The programming abstraction of TensorFlow enables the use of different heterogeneous accelerators with ease. The use of heterogeneous accelerators is increasingly common on computing clusters. TensorFlow uses optimized computing kernels with such as Intel MKL and NVIDIA cuBLAS as backend of operations. These libraries are often custom made for processors commonly used on HPC platforms. Furthermore, our tests shows that TensorFlow is capable of harvesting the bandwidth provided by InfiniBand verbs RDMA, which is commonly used to interconnect HPC clusters.

7.2 Discussion

As discussed in chapter 6, we observed that TensorFlow performs particularly well in tiled matrix multiplication but poorly with a distributed CG solver. From timeline profiling it can be seen that this is likely
due to computing resources being left idle because of data movement. Since TensorFlow is originally designed for data driven neural network training in mind, the model is inherently data-driven. Therefore in order to efficiently program TensorFlow, one must carefully tune the algorithm such that data movement does not become the bottleneck of the pipeline. Furthermore, the number of tasks should be sufficiently large such that continuous and asynchronous computation can hide latency due to data movement. We showed that by an optimized CG solver which requires relatively little data movement by not allowing task over decomposition. The performance compared to the original version is extremely well.

7.3 Future work

In this work, we have shown that computation problems that are suitable for TensorFlow as a programming framework requires a few properties. The problem must be large enough to be decomposed into sufficiently many tasks to keep the computing pipeline busy. Due to the sensitiveness of pipeline disruption, synchronization should be kept at minimal and data movement should be balanced relative to computation workload. We have only explored a limited number of computation problems in this work. It will be interesting to explore more kinds of algorithms which can be reformulated in such a way that is suitable for execution on TensorFlow. In our implementations, we have used a client session as mean of providing tasks to the pipeline through dictionary feeding. As seen from the analysis this can potentially become a bottleneck. Therefore it would be interesting to further investigate on the possibilities of avoiding such interaction. Another problem with our implementation is that the problem must fit inside the memory of a computing node such that it can perform task decomposition. This is undesirable as problem sizes can potentially exceed the memory capacity of one node. One approach would be to disallow over decomposition and pin specific tasks to devices as in our optimized CG solver. Yet this still potentially requires data passing through the host memory system. Another approach would be to take advantage of the new Dataset API, where data is automatically feed into the graph from different sources, such as file. One important difference between Dataset API and Queue API is that dataset operations are
executed in C++ threads, while queue runners are executed as python threads, which are in principle not parallel. In addition, TensorFlow supports various means of I/O where data can be read. Currently, Amazon S3 object store is supported as a data source. Object store is a storage technology which can potentially improve I/O performance on HPC systems [14][38]. Apart from data flow, another area which can be explored is the use of low precision in algorithms. Since TensorFlow is data driven, one approach to reduce data flow latency is to reduce number precision, which effectively lower transfer cost. In terms of communication performance, studies and test suites are being made to quantify the performance of TensorFlow’s communication protocols [10]. Communication plays a vital role in the performance of distributed TensorFlow computation. Therefore being able to perform fine-grained network measurement and pin point bottlenecks is extremely important prerequisite for performance tuning. It would be beneficial to run these tests on Tegner, the cluster platform where tests in this study were conducted.

In terms of algorithms used in this work, it can be seen from profilings that the main bottleneck comes from data transfer. The GPU we used in this work has relatively small amount of memory. In the case of more powerful GPUs such as NVIDIA K80 with 12GB ram on each engine, it will be very difficult to keep the GPU full as transfer cost will become extremely expensive. This can be solved by redesigning the algorithm and introduce data locality, that is each GPU would be responsible only for one share of task without task over decomposition. We show that scalability can be achieved this way in one of our CG implementation. However techniques on efficient synchronization and broadcast operations have to be investigated.

In terms of TensorFlow operations, one interesting area which was not explored in this work is XLA [51]. For graphs with many small intermediate operations it is possible to reduce pipeline overhead by fusing those operations. Since TensorFlow theoretically does not restrict the type of accelerators being used as long as required computation kernels are provided, it would be interesting to explore if TensorFlow is capable of handling accelerators other than GPU. One such type of device would be FPGA [39]. It is possible to add support for different devices through XLA code compilation. Another interesting but not explored feature is fault tolerance. For HPC systems with large number of nodes, node failure is becoming a normal scenario which must
be routinely handled [43]. TensorFlow provides a snapshot feature which allows periodic snapshot of workers at a particular stage. With those snapshots it is possible to restart execution from a particular step given that all the computation are performed within the TensorFlow graph instead of client session. For highly parallelized and long running execution the ability to recover from faults would be extremely important.

To conclude, TensorFlow is a promising and yet to be explored framework for distributed computation.
Bibliography


