Imperial College London

MEng Individual Project

Imperial College of Science, Technology and Medicine

Department of Computing

TensorBow: Supporting Small-Batch Training in TensorFlow

Author:
Ioan Budea

Supervisor:
Prof. Peter Pietzuch

Second marker:
Dr. Holger Pirk

June 17, 2019

Abstract

Deep neural networks are trained using mini-batch Stochastic Gradient Descent (SGD) on specialised hardware accelerators such as a GPU. Existing training systems support the configuration of the mini-batch size poorly: they couple its value, an accuracy-driven parameter, with the hardware configuration because of the training algorithms used to scale out to multiple GPU devices. Constantly increasing the mini-batch size to accommodate faster or more hardware resources hinders model convergence, and the tuning techniques used to compensate for this effect are problem- and model-dependent.

In this report, we explore how to design and build a deep learning system where the user can freely choose the batch size and still utilise all of the available hardware resources. We propose TensorBow, a system that adds support for small-batch training in TensorFlow. TensorBow integrates two deep learning systems, TensorFlow and Crossbow, combining the best features of the two into one cohesive solution. TensorFlow, for example, has many optimised operators and abstractions, but uses by default a single GPU stream per device for training a model. Crossbow is able to train multiple model replicas on one GPU concurrently and uses a novel synchronisation algorithm, *synchronous model averaging*, instead of the standard synchronous parallel SGD, to ensure convergence as the number of model replicas increases. However, it lacks automatic differentiation and support for advanced user primitives.

TensorBow adds support for the design principles of Crossbow in TensorFlow. It also proposes a transparent, automated parallel training interface. The main challenges in implementing TensorBow are related to the fact that many TensorFlow components and abstractions are designed under the assumption of training a single model replica per GPU, making them unsafe for concurrent use. We extend those components to safely train multiple model replicas per GPU.

Our experimental results show that TensorBow improves TensorFlow's hardware efficiency by at least 50%, irrespective of the neural network size used in our evaluation; and that the synchronisation overhead between models is very small.

Acknowledgements

Firstly, I would like to express my gratitude towards my parents, who have always supported and encouraged me unconditionally.

I would like to thank my supervisor, Prof. Peter Pietzuch for proposing interesting and challenging projects and for always keeping a high work standard. I am also thankful for the feedback given by Dr. Holger Pirk, who explained a lot about how I should view and tackle this project.

I am extremely grateful for the advice and guidance of Dr. Alexandros Koliousis. I believe our discussions were extremely productive and I always had an important idea or lesson learned from them. Special thanks to Dr. Luo Mai as well for all his support.

Finally, I would like to thank my close friends Alex, Adi and Andrei for all our technical discussions, moral support and for always bringing the best out of each other.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Context

The current decade is characterised by an unprecedented amount of data. For example, every day there are tens of petabytes of server logs generated by Microsoft data centres [42]; and tens of terabytes of user activity logs produced by Facebook social applications [44]. This kind of data powers a wide range of Artificial Intelligence (AI) applications [43]. AI applications leverage labelled or unlabelled data samples to train machine learning models in order to solve problems such as anomaly detection [41], product recommendation [40], user classification [55] etc. To ensure that data can be trained in a timely manner, practitioners often use hardware accelerators such as GPUs or TPUs [16]. These devices provide orders of magnitude speed-up [22] in training time compared to CPUs. This leads to a growing deployment of heterogeneous hardware, e.g. Facebook's Big Basin GPU servers, whose architecture was released as part of the Open Compute project [51], have 8 GPUs installed and they are used for many machine learning workloads and benchmarks.

Among all machine learning models, Deep Neural Networks (DNNs) have drawn substantial attention recently due to their unprecedented performance in challenging AI tasks, including computer vision [22], speech recognition [21] and natural language processing [20]. A DNN contains many inter-connected layers, such as fully-connected, pooling [10], convolutional [39] and recurrent [9]. Input data are fed into a DNN and propagate through layers parameterised by a set of variables (usually represented by an array of floating point numbers) to produce an output used for prediction. By stacking more and more layers, a DNN can extract increasingly more features, leading to the recent rise of Deep Learning (DL) models such as VGG [13], ResNet [12] and Inception [11]. DNNs can naturally leverage the recent rise of big data and heterogeneous hardware. Since they are computationally intensive and involve expensive floating-point operations on vectors and matrices, these workloads can be efficiently executed by specialised hardware accelerators like GPUs.

Supervised training of a deep neural model is often conducted with the *Stochastic Gradient Descent* (SGD) training algorithm. This approach uses a dataset with inputs and labels split into training and test data. The learning process involves predicting the label of each training data sample and computing an error – in mathematical terms, a loss function. Model variables are then adjusted using the gradient of the loss function with respect to each one of them. This process is iterative as it takes several steps to reach a local minimum of the loss function. The generalisation performance of the model is then judged on how well it predicts the previously unseen test data. In order to speed up the training process, *mini-batch SGD* is employed. Multiple input examples are grouped together in a batch and fed into the model at once. The training usually completes when a desired test accuracy is reached.

Mini-batch size plays a key role in determining the result of training. The value of a mini-batch size (in the following, we use batch size for simplicity) is an important parameter of the training algorithm that must be carefully configured. On one hand, a large batch size means aggregating multiple examples, thus producing an accurate approximation towards the true

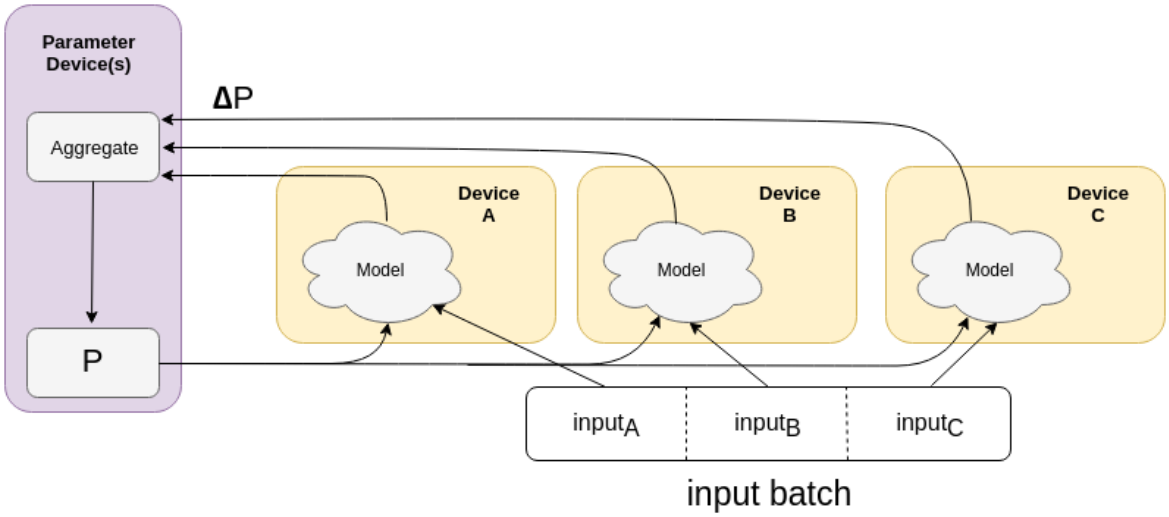


Figure 1.1: Introduction to synchronous SGD. The input data is split into three partitions one per GPU device. Each GPU uses a local copy of the model variables to compute a partial gradient, ΔP. A synchronisation step aggregates all partial gradients. The aggregated gradient, P, is communicated to each GPU to update its local model before training the next batch

gradient of the loss function; however, it reduces the number of gradient updates that correct the model parameters (as we have limited training samples). As a result, the algorithm explores less minima and this affects the generalisation capability of the model. Using small batch sizes, on the other hand, leads to a significant number of gradient updates to refine model parameters. However, as gradients can become less accurate due to the small number of samples used for one batch, very small batch sizes (i.e., 1 and 2) could adversely affect training. In practice, users often use small-batch training: typical effective batch sizes are usually between 16 and 512 [7, 4]. These batch sizes balance training speed and accuracy while being applicable to a wide range of DL models [7, 8, 6].

1.2 Problem statement and related work

Though important, the configuration of batch size is still poorly supported by deep learning systems today. These systems couple accuracy-oriented tuning for batch size and performance-oriented tuning for the system processing throughput. Any tuning of the batch size eventually affects system processing throughput, making it difficult to balance the accuracy and duration of a training job. Such a coupling is caused by two principle designs adopted by most, if not all, systems, including TensorFlow [2], PyTorch [56] and Caffe2 [45]:

- *Using a single GPU stream for training within a GPU.* Deep learning systems often adopt a dedicated GPU stream for training on a GPU. To fully utilise a contemporary GPU like a V100, a user has to increase the batch size up to 1024 or higher for certain models [32], which is orders of magnitude higher than the preferable small-batch range. Adopting a small-batch for training often results in severe device under-utilisation [4].

- *Using the Synchronous SGD algorithm to scale out training on multiple devices.* To scale out training, systems shard the training dataset on multiple GPU devices and adopt Synchronous SGD (S-SGD) to synchronise model replicas (see Figure 1.1 for more details). S-SGD, however, couples the mini-batch size with the number of devices, enforcing users to use large-batch training in a distributed setting. To use 256 GPUs for training, for example, practitioners have to use a batch size up to 65,536 [5].

To resolve the issues incurred by large batch sizes, users must tune a wide range of hyper-parameters such as the learning rate or the weight decay [5]. This had led to a large collection of

hyper-parameter tuning techniques, including auto-tuning [54], linear scaling [5] and warming-up [5, 25] for the learning rate, auto-tuning the batch size, or auto-tuning the momentum. Applying these techniques requires expertise in optimisation theory as well as experience in training models. In addition, these techniques are usually problem-specific and dataset-specific. For example, the tuning techniques proven useful for residual neural networks [12] are often hard to apply to generative neural networks [34, 33]. Finally, most, if not all, of these tuning techniques eventually fail when the batch size gets too big. Constantly increasing batch size would eventually hinder convergence [14].

More recently, we observe a new system design, namely Crossbow [4], that can help freely configure batch size when scaling training. Crossbow has two novel design principles. First, it advocates for a concurrent training strategy on one GPU. This can be achieved by spawning parallel GPU streams to train multiple replicas of a deep learning model inside a GPU even when they are configured to use a small batch size. Second, Crossbow argues for alternative synchronisation techniques, moving away from the de-facto S-SGD. By using a variant of model averaging techniques [38, 37], Crossbow is able to consolidate a large number of model replicas collaboratively trained using small batch sizes. As a proof-of-concept, Crossbow adopts a clean-slate design and implementation using a custom programming interface. This prevents Crossbow from being used by users who are mainly working with TensorFlow and other popular frameworks and benefit from a wealth of models already developed.

1.3 Objective and associated challenges

In this project, we explore a new system design that can enable small-batch training for a wide range of deep learning applications. For this, we realised two principle design choices advocated by Crossbow in the popular TensorFlow framework. The integration of the two systems is non-trivial due to limitations of TensorFlow:

- *Training one deep learning model per GPU.* TensorFlow assumes that it trains a single model per GPU. This makes it difficult to support small-batch training. Small-batch training requires to train multiple instances of the same model per GPU concurrently. However, many of the software design assumptions in TensorFlow do not support this. Consider, for example, TensorFlow's GPU memory allocator. TensorFlow instantiates only one allocator per GPU and it not safe to share it across two or more execution streams on the same device by default [49].

- *Device-oriented data parallel training.* TensorFlow assumes that data parallelism is only achieved by increasing the number of devices, not the number of model replicas trained. Small-batch training, however, entails training more model replicas than devices. This breaks many of TensorFlow's design assumptions about how the dataflow is replicated, executed, and combined at the end of an iteration of the training algorithm.

- *Cumbersome parallel training interface.* TensorFlow requires the developers to manually map operators to GPU devices in order to parallelise their computation. Although Keras [48], a layer built on top of TensorFlow, automates some of these placement decisions, neither can efficiently automate placement when training multiple model replicas per GPU.

1.4 Contributions

To address the above challenges, we propose a novel design that enables TensorFlow to efficiently support small-batch training. The new system, TensorBow, supports the following features:

- *A task scheduler that supports concurrent training of models on a single GPU.* We have added support for concurrency in TensorFlow (i.e. the ability to train many model replicas in a single GPU).

- *A model-oriented data parallel training architecture.* Instead of using the number of available devices to determine data parallelism, we use the number of model replicas. TensorBow handles the synchronisation among the model replicas.
- *An automated parallel training interface.* Instead of manually placing operators to devices, TensorBow's users write a program assuming a single device abstraction. TensorBow then automatically replicates the dataflow graph within and across devices.

Those modifications enable training TensorFlow models with a 30% reduction in time to reach a target test accuracy and up to a 50% improvement in a single GPU throughput, showcasing the benefits of the modified training technique and the improved convergence of training with a small batch size.

1.5 Academic and Industrial Relevance

This project takes a novel idea from an academic paper and applies it to a well-established industrial system, making it accessible to a vast target audience.

An improved training algorithm for DL models can lead to a much better resource utilisation, improving the productivity of cloud providers that offer training resources as a service (e.g. Google and Amazon), as well as the productivity of users that rent those resources. By contributing to two open source projects, one academic (`https://github.com/lsds/Crossbow`) and one industrial (`https://github.com/tensorflow/tensorflow`), the design and implementation contributions of this work can be further refined by the community.

The project also enables assessing alternate training algorithms for deep learning. The current approaches for scaling mini-batch SGD are non-transferable since they depend on tuning many different hyper-parameters that are model-specific. By porting Crossbow model averaging algorithm in TensorFlow, we aim to promote alternative ways to train and synchronise models.

Chapter 2

Background

Training deep learning models requires algorithms that take advantage of the parallelism of modern hardware. This chapter starts by introducing basic principles from optimisation theory applied to deep learning algorithms. These concepts are useful to understand what are the limitations of current state-of-the-art algorithms and why the batch size is such an important parameter. Then, this chapter presents an overview of two algorithms: the first one (S-SGD) needs to increase the batch size, while the second one (SMA) tries to keep it constant. The architecture of TensorFlow and Crossbow is presented in detail in the second part of the chapter to illustrate the pros and cons of each framework. By using distinct features from each system, we aim to improve convergence time and throughput of deep learning workloads.

2.1 Training algorithms

2.1.1 Mini-batch SGD

Deep learning is an optimisation problem. Optimisation refers to the task of either minimising or maximising an objective function $L(w)$ by changing model variables w. Optimisation algorithms for training deep learning models typically include some specialisation on the specific structure of the objective function, L, otherwise known as the loss function [17].

In supervised learning, the training data set contains multi-featured examples and an associated label for each example. The optimisation task is to find the model parameters w that minimise the number of labels predicted incorrectly.

The input dataset is split into training data and testing data, the latter used to evaluate the generalisation of the trained model. Training a model involves changing the model variables w to predict the labels of training data as accurately as possible. It usually takes many iterations over the training data to minimise the prediction error. The test data is then used to evaluate how well the model predicts previously unseen examples (the *test* or *validation accuracy*).

One iterative method for finding the parameters that give the minimum value of the function is gradient descent. If we wish to minimise our loss function $L(w)$ given some model parameters w, gradient descent proposes a new point:

$$w^{(k+1)} \leftarrow w^{(k)} - \lambda^{(k)} \nabla L(w^{(k)})$$

where λ is the learning rate at iteration k, a positive scalar determining the step size. The learning rate can be constant, but typically it will also be adjusted during training.

If w^* is the set of parameters that gives the global minimum of the loss function, in order to derive a loss value with error at most ϵ, i.e.

$$L(w^{(k)}) - L(w^*) \leq \epsilon$$

we would need $\mathcal{O}(\frac{1}{\epsilon})$ iterations [35].

However, the loss function is expressed as a sum over all N training examples. Each sample has a contribution expressed using a function l that, given w, measures the difference between the predicted label $y' = f_w(x)$ of a sample and the ground truth y.

$$L(w) = \frac{1}{N} \sum_{i=1}^{N} l(f_w(x_i), y)$$

Therefore, each iteration takes $\mathcal{O}(N)$ and the total computational complexity is $\mathcal{O}(N\frac{1}{\epsilon})$ [35]

Considering the size of the datasets used for deep learning training, it becomes clear that the cost of this method is prohibitive. To address the issue, the method of Stochastic Gradient Descent (SGD) was proposed. It uses the insight that there is one main difference between machine learning algorithms and general optimisation problems: the loss function can be decomposed as a sum over the training examples. Therefore, optimisation algorithms for machine learning typically compute each update of the parameters based on an expected value of the loss function estimated using only a subset of the terms of the full loss function (i.e. by looking at just a subset of the training data which will be called a mini-batch or batch for simplicity).

With the above, the model parameters are updated based on the following formula:

$$w^{(k+1)} \longleftarrow w^{(k)} - \lambda^{(k)} \nabla l_b(w^{(k)})$$

$$\nabla l_b(w^{(k)}) = \frac{1}{b} \sum_{i=1}^{b} \nabla l(f_w(x_i), y)$$

Since the batch size b is a constant, this brings down the cost of each iteration to $\mathcal{O}(1)$, so the overall complexity of SGD remains $\mathcal{O}(\frac{1}{\epsilon})$. This is a big advantage in the case of a large N.

In the case of deep learning, the model parameters w will be split across many layers. The technique for training arbitrary multi-layer neural networks is back-propagation [28]. It employs the chain rule in order to compute the necessary gradients for each update step. The process has two stages:

1. given a set of inputs, a forward pass through the network calculates the current output of the model; the error will be the difference between this output and the target result

2. the backward step passes the error to every layer in the network in reverse order and updates the weights according to the gradient descent method

The most important metric when training a deep learning model is time-to-accuracy, defined as the time taken in order to reach a desired level of test accuracy. Time-to-accuracy is influenced by 2 components: statistical efficiency (the theoretical bound on the number of iterations required in order to reach that accuracy - given without a proof in section 2.1) and hardware efficiency (the duration of each iteration, influenced by the hardware resources available and the way in which they are used).

2.1.2 The benefits of small-batch training

Batch size plays a key role in determining training accuracy. In the following, I will analyse the effect of batch sizes and the typical range of batch sizes when training real-world DL models.

The effect of the batch size. Batch size has a sophisticated effect towards the training process. From the learning point of view, its effect is mainly two-fold:

- *Noise of estimated gradient.* When running SGD, batching a large number of data samples for estimating true gradients can significantly reduce the noise of estimation [17]. This helps improve the robustness of the training process, and thus benefit convergence. However, noise, on the other hand, is key to enable SGD to thoroughly explore in a given loss space for minima, leading to stronger generalisation capability of a trained model [31].

- *Number of model updates.* A SGD method often works with datasets of limited sizes. As a result, training is often specified with a fixed number of data epochs and it is anticipated that the model can converge to a desired accuracy. Using a large batch size, however, can significantly reduce the number of updates to the model. In the worst case where the batch size is the same as the dataset size, we end up with updating the model only once throughout the training. Though the gradient estimation becomes accurate, the DL user has to specify a very accurate learning rate (as well as other hyper-parameters), which is prohibitively difficult and usually infeasible, so that SGD can reach the minima in one iteration of training.

The regime of effective batch sizes. Given this complex effect, batch size must be carefully configured in order to achieve high training accuracy. DL users often prefer setting up a batch size that can balance the noise and the number of updates. Recent empirical studies [7] show that effective batch size often settle within the small size regime (typically between 16 and 256) across a wide range of DL models. This observation is also supported by DL practitioners and researchers [6] who often find it difficult to achieve effective validation of a DL model in training when using batch size larger than 64, leading to recent rethinking of DL system design in key industry players such as Google and Facebook [18].

2.1.3 Synchronous Parallel SGD

In order to speed up the training process, the main approach is to divide gradient computation across multiple GPUs and exploit data parallelism. The idea is to partition the training examples from a batch and feed them to different GPUs, each computing a partial gradient. The problem then becomes how to synchronise those partial gradients and combine them to produce an update for the model variables.

The most common training algorithm is called Synchronous Parallel SGD (S-SGD) [25]. Every GPU is responsible for training a copy of the model on a subset of the batch data. We will call each of those separate instances of the model a *replica*. Because each GPU sees a different set of examples, each replica will have computed a different gradient at each iteration. The idea of the algorithm is to ensure that all GPUs have a consistent view of the model before completing an iteration and moving on to the next batch of examples.

The S-SGD algorithm is summarised in Algorithm 1 and it involves 5 key steps. Each GPU:

1. gets a partition of a training batch (line 4);

2. computes a partial gradient using the examples in Step 1 and the local model replica (line 5);

3. agrees with the other GPU devices on the aggregate gradient for this batch by taking the average over all the partial gradients (line 7);

4. updates the local replica with the average gradient (line 9); and

5. continues to the next iteration (if there is still a batch to process).

This method improves the execution time of an iteration of the algorithm, but it can be limited by the performance of the slowest node (a *straggler*), since the algorithm requires the gradients from all the GPUs before aggregation. One option to address the effect of stragglers is to use Asynchronous Parallel SGD (A-SGD) [27]. A-SGD modifies Step 3 of the above training method in that the fastest GPU does not have to wait for all partial gradients to be computed before updating its own model replica. Instead, it uses only the partial gradients that have been computed thus far. The downside of A-SGD is that incomplete gradient updates tend to have a weaker convergence rate [27]. Thus, S-SGD remains the main method used in practice [5].

Algorithm 1: Synchronous Parallel SGD (S-SGD)

input : w_0, an initial model;
B, a set of batches;
n, the batch size;
λ, a learning rate parameter;
output: w, the trained model.

while *target accuracy not reached* $\wedge$ $|B| \geq 0$ **do**
  // k-th iteration of the learning algorithm
  select batch $b \in B$;
  for $r = 1 \dots R$ **do**
    // replica r in the k-th algorithm iteration
    get a mini-batch b_r of size n/R from b ;
    compute $\Delta w_r = \nabla l_{n/R}(w^{(k)})$ using b_r ;
  end
  $\Delta w = \frac{1}{R} \sum_{r=1}^{R} \Delta w_r$;
  for $r = 1 \dots R$ **do**
    // each replica r updates the model with the average gradient
    $w^{(k+1)} \longleftarrow w^{(k)} - \lambda^{(k)} \Delta w$
  end
end

2.1.4 Synchronous Elastic Model Averaging

CrossBow [4] is a new machine learning system developed by the LSDS research group at Imperial College. It proposes a new algorithm for deep learning training called Synchronous Model Averaging (SMA), aimed at improving time-to-accuracy when compared with the previously discussed methods.

The idea behind SMA is based around the notion of a *learner* [4], an independent entity which takes some batch of samples and maintains a replica of the model being trained. Each learner computes the gradient based on the examples it encounters during an iteration. In order to prevent learners from diverging, a central average model guides the evolution of each learner by correcting the trajectory of each learner. Updating this central model is the part that makes this algorithm synchronous. There will be one designated learner chosen to manage the central model. However, unlike S-SGD, the model constructed by each learner will not be identical at the end of an iteration.

The intuition behind the concept of corrections is that SMA pulls diverging learners towards the central model that was computed at the end of the previous iteration. While each learner is independent, adding those corrections will eventually ensure that they will agree on the correctness of the central average model and will arrive at the same result. The correction is computed as the different between the gradients, scaled with a factor $\alpha = \frac{1}{\#learners}$.

The other novel idea proposed by CrossBow is adding a momentum term only to the update phase of the central model, in order to speed up the convergence of the central model compared to the learners.

The formalised algorithm is shown in Algorithm 2. Input hyper-parameters such as the learning rate can also be dynamically adapted during training as it is usually done in other parallel training methods, if necessary.

As explained in the previous section, training with a small batch size is a good option for achieving high statistical efficiency, but it may underutilise the GPU resources. The level of indirection introduced by the concept of a learner in SMA allows us to separate the batch from the number of replicas and the number of processing devices. Previously, each GPU was training one replica with a subset of the data within the batch. Now, since each learner gets a batch and has a model associated, we can co-locate as many of them on a single device as necessary. This

Algorithm 2: Synchronous model averaging (SMA) (Sourced from [4])

input : w_0, an initial model;
$w_1 \dots w_L$, L model replicas trained by L learners;
B, a set of batches;
λ, learning rate;
μ, a momentum parameter;
output: z, the trained model.

// initialise central average model and its previous version
$z \leftarrow w_0$;
$z_{prev} \leftarrow \emptyset$;
while *target accuracy not reached* $\wedge\ |B| \geq L$ **do**
  // iteration i of the learning algorithm
  $c_1, \dots, c_L \leftarrow \emptyset, \dots, \emptyset$;
  for $j \in \{1, \dots, L\}$ **do**
    // learner j in iteration i of the algorithm
    $B_j \leftarrow select(B)$; // Select batch for learner j
    $B \leftarrow B \setminus \{B_j\}$; // Remove the batch
    $\Delta w_j \leftarrow \lambda \nabla l(w_j)$; // Gradient for replica j using only examples from B_j
    $c_j \leftarrow \alpha(w_j - z)$; // Correction for replica j
    $w_j \leftarrow w_j - \Delta w_j - c_j$; // Update replica j
  end
  // Update central average model
  $z' \leftarrow z$;
  $z \leftarrow z + \sum_{j=1}^{L} c_j + \mu(z - z_{prev})$;
  $z_{prev} \leftarrow z'$;
end

promises to take advantage of the GPU resources at a much better rate when multiple learners run concurrently.

Given the observation that many learners may end up co-located onto the same GPU, we split the synchronisation phase into two stages: intra-GPU and inter-GPU aggregation. There will be a reference model per GPU, managed by a single learner, and the central model. The learners executing on a single GPU synchronise with respect to the local reference model, while the central model is aggregated from the reference models using the SMA algorithm. This technique accounts for the difference in communication latency within and across GPUs.

Finally, the number of learners per GPU is adjustable automatically at runtime by analysing the throughput of each GPU measured in batches processed per second. Initially, every GPU has a single learner, therefore a single replica to train. If the throughput of a processing unit has increased over the last period with a value above a constant threshold, a new learner is started on that GPU. On the other hand, if the throughput decreased, a learner is removed from the working set.

The SMA algorithm proposes a number of very intriguing advantages: an increased hardware efficiency with minimum statistical efficiency penalty, as well as the possibility of making the training faster while still being more robust because of the small batch sizes used. We would like to retain CrossBow's superior performance influenced by this new approach, but we don't necessarily want to change the API that other machine learning frameworks offer. Users are already familiar with these other building blocks that have reached a necessary level of maturity in their development.

Therefore, we want to interface with programs that can be otherwise run by other machine learning frameworks available, but apply these ideas in order to improve time-to-accuracy for training of deep learning models.

Framework	Stars	Watchers	Forks	Date created
TensorFlow	118,674	8,566	71,357	2015-11-07
PyTorch	23,881	1,175	5,667	2016-08-13
MXNet	16,052	1,185	5,811	2015-04-30
CNTK	15,676	1,393	4,198	2015-11-26
Caffe2	7,225	521	1,670	2015-06-25
Chainer	4,468	333	1,185	2015-06-05

Table 2.1: Github project statistics for various deep learning systems (as of 25-01-2019)

2.2 Training systems

2.2.1 Training with a GPU

Graphics Processing Units (GPUs) are throughput-optimised, multi-core computing devices. While CPUs have a hierarchy of memory caches designed to hide main memory access latency, GPUs rely on hardware multi-threading [30]. A GPU executes thousands of threads that have some processing to do while others are issuing requests for some data to be fetched from main memory. This is the main reason why GPU memory is focused on throughput rather than latency.

A GPU can be described as a Multiple Instructions Multiple Data (MIMD) processor constructed from a collection of multi-threaded Single Instruction Multiple Data (SIMD) processors (termed streaming multi-processors). GPUs can execute more floating-point instructions per second than a normal CPU. Although GPUs were initially designed for a narrow set of applications [30], programmers wanted to see if they could take advantage of the high hardware efficiency of GPUs by having a common abstraction to specify their applications. Programming languages were developed in order to enable interfacing with the GPUs. An example is NVIDIA CUDA (Compute Unified Device Architecture), which is a C-like programming language with a few restrictions. These programs written for GPUs are called kernels and map to a GPU thread, which controls a stream of instructions on the actual GPU.

Because many machine learning algorithms perform operations such as matrix multiplications, involving a large amount of memory, and execute many floating point instructions, GPUs became a natural choice for executing such workloads since they will achieve more model updates per second. It is common for modern servers to therefore include multiple inter-connected GPUs [5] to increase the throughput of various machine learning workloads. This is why we are going to focus on the GPU implementation details of the two frameworks to improve the existing setup.

2.2.2 TensorFlow

A new algorithm for training would be better supported and tested by an active community of users and developers of an existing machine learning framework. Table 2.1 shows the popularity of various frameworks in terms on the number of "stars", "watchers" and "forks" of their respective Github projects. TensorFlow is the most popular deep learning framework, which is why we chose it for our integration.

TensorFlow [2] is constructed based on the concept of modelling an arbitrary computation as a dataflow graph. This idea was previously present in many other distributed computing systems such as Spark [23] and DryadLINQ [15]. However, unlike traditional dataflow systems, where graph vertices were used to represent a functional transformation on immutable data, TensorFlow graph vertices encapsulate computations that own or update mutable data. Within TensorFlow, edges represent tensors (typed multi-dimensional arrays) that flow between vertices. Tensors follow a producer-consumer pattern, with the input vertex as the producer

```python
import numpy as np
import tensorflow as tf

# placeholder for input data
x = tf.placeholder(tf.float32, (100, 784), name="x")
# matrix of weights
W = tf.Variable(tf.random_uniform((784, 100), -1, 1), name="W")
# bias vector
b = tf.Variable(tf.zeros((100,)), name="b")
# result = ReLu(Wx + b)
result = tf.nn.relu(tf.matmul(x, W) + b, name="ReLu")

sess = tf.Session()
sess.run(tf.global_variables_initializer())
sess.run(result, feed_dict={x: np.random.randn(100, 784)})
```

Listing 2.1: Logistic regression implemented in TensorFlow

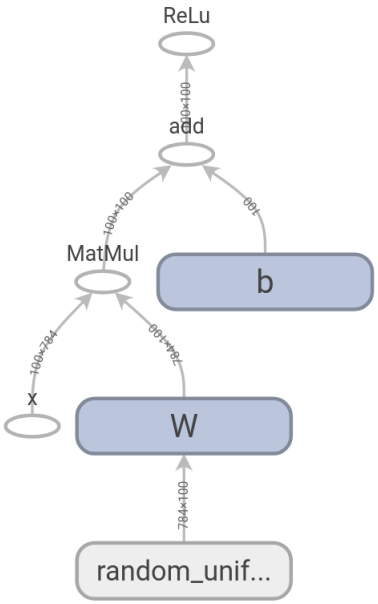


Figure 2.1: Corresponding computation graph for the logistic regression example of Listing 2.1

and the output vertex as its consumer. Listing 2.1 shows a user program (in Python) that implements logistic regression. The resulting dataflow graph is shown in Figure 2.1.

In a TensorFlow computational graph, each node can have zero or more inputs and zero or more outputs. A node is an instance of a particular operation. A kernel is the code which represents a particular implementation of an operation suitable for a particular device (e.g., CPU or GPU). TensorFlow exposes the available set of operations and kernels using a registration mechanism, and it allows users to extend these by registering additional operations and/or kernel definitions.

Apart from tensors, edges can have another special type, called *control dependencies*. No data is transmitted along control dependencies. They simply indicate that the source node for the control dependency edge must finish before the destination node starts executing. This is a valuable feature because our model may contain mutable state, so control dependencies can be used directly by clients to enforce an execution order between operations for memory considerations or other optimisations.

Because many machine learning algorithms require computing gradients for optimisation

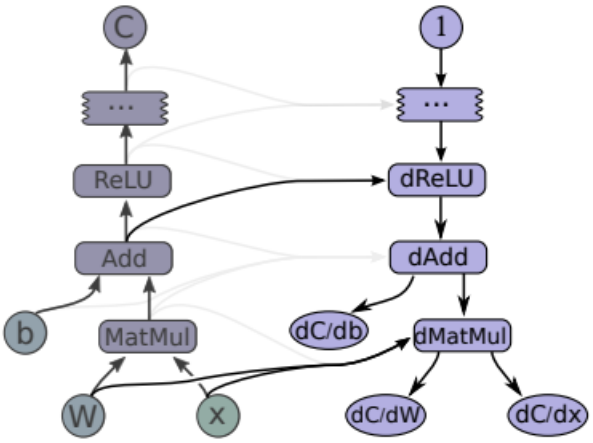


Figure 2.2: TensorFlow automatic differentiation using graph transformations. Sourced from [1].

(as explained in Section 2.1), TensorFlow has automatic differentiation as a feature within the framework. Automatic differentiation is a method that allows the user to compute the gradient of any tensor with respect to some set of other tensors it depends on. The method augments the original dataflow graph. Figure 2.2 shows an example of this graph transformation. The framework can automatically find the path in the graph linking an input tensor C to an output tensor W. Then, it walks along the inverse path from C to W and, for each node representing an operation in the original graph, it adds an extra node to the dataflow graph. This represents the practical implementation of the chain rule. The newly added nodes compute what is called the *gradient kernel* for the original operation in the forward path. Any operation can have a correspondent gradient kernel that uses as inputs the partial gradients computed already along the backward path and, optionally, the inputs and outputs of the forward operation.

Some of the other core design principles that guide the design of TensorFlow are:

- the ability to model any mathematical operation as a node in the computational graph
- deferred execution to achieve better performance and allow optimisations
- common abstractions for heterogeneous hardware and cross-platform support: it supports multiple Operating Systems, as well as desktops, servers and mobile devices
- multi-language development and support: TensorFlow implements its API in Python, C++, Go, Java etc.
- extensibility: mechanisms to add your own operations, kernels and devices
- support for local and distributed training

Because of all those benefits, many users have become accustomed to the programming interface that TensorFlow provides and have developed and deployed into production a variety of models tailored onto its API. There is also increasing interest to support many programming languages for developing machine learning models using TensorFlow. One such example is TensorFlow.js, an ecosystem of JavaScript tools aimed at expanding the creation and deployment of TensorFlow models into web browsers or in applications developed under Node.js.

TensorFlow was widely adopted because it was one of the first projects that offered fast prototyping and a rich set of predefined bindings for developers. The core of TensorFlow has reached a desirable level of maturity, so any notable improvements may generate a tremendous impact.

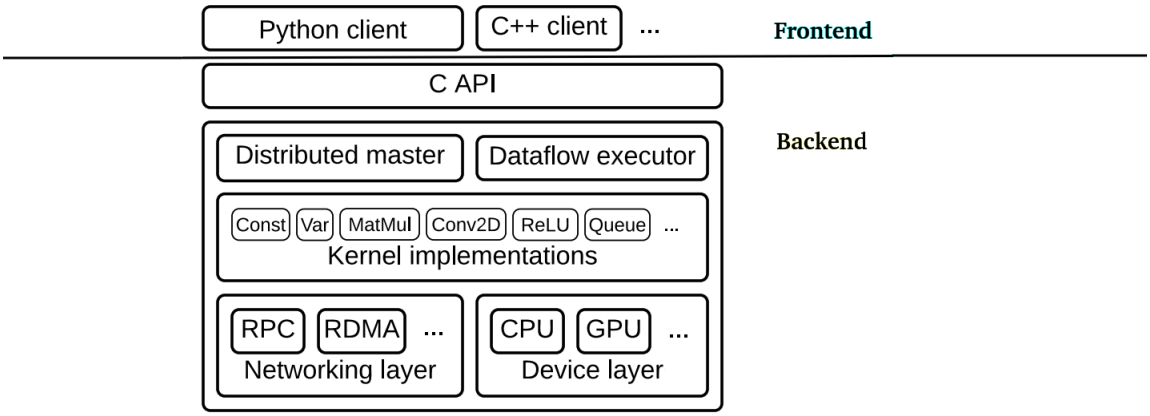


Figure 2.3: TensorFlow system architecture. Adapted from [2].

2.2.2.1 Architecture

The TensorFlow architecture is divided by a C API into two parts: the frontend and the backend subsystems (Figure 2.3). The frontend supports various programming languages, Python being the one most commonly used by developers to specify dataflow graphs. The frontend is responsible for assembling the computational graph and calling the C API to request various services from the TensorFlow backend. The backend is the program that actually does the execution of the user-specified graph. It is implemented in C++ and contains the runtime (represented in Figure 2.3 by the Distributed master and Dataflow executor components), the kernels implementing the supported operations, a networking layer for communications and a device layer which abstracts different types of devices (e.g. CPUs, GPUs, or TPUs).

TensorFlow has both a local and a distributed implementation of its interface. Apart from the 2 main components described in Figure 2.3, TensorFlow identifies special entities for the execution of a program: the client, the master and one or more worker processes. The client corresponds to the frontend and communicates with the master process to start the training, while each worker process is responsible for facilitating access to one or more computational devices (such as CPU cores or GPU cards) and for executing graph nodes on those devices as instructed by the master. In the local implementation, the master, the client and the worker(s) are located within a single operating system process running on a single machine. The distributed implementation uses similar mechanisms and interfaces, but supports running the client, the master and the workers in different processes on different machines.

In order to understand what information is available to the TensorFlow runtime from the frontend, we describe the principles behind the construction of the C API and how it acts as a bridge between the frontend and the backend.

Multi-Language Interoperability TensorFlow uses Bazel as the build tool to generate a Python package. Before the C++ part of system is compiled, Bazel invokes SWIG. SWIG [52] is the tool that connects the frontend and the backend of TensorFlow. It is an interface compiler able to connect programs written in C/C++ with high-level languages such as Python, Ruby, or Perl. It takes existing function declarations found in C/C++ header files and uses them to generate wrapper code that another language needs to call to use the underlying C/C++ code. The methods to be exported are specified in special interface files ending with the `.i` extension. Apart from a few SWIG-specific directives, `.i` files contain ANSI C function and variable declarations.

Figure 2.4 shows the outline of the build process. After the necessary files have been automatically generated by SWIG (one containing Python code and the other one C++ code), the backend is compiled into a shared object that will be dynamically linked to provide access to the methods present in `c_api.h`. Then, the Python generated file knows how to load the shared

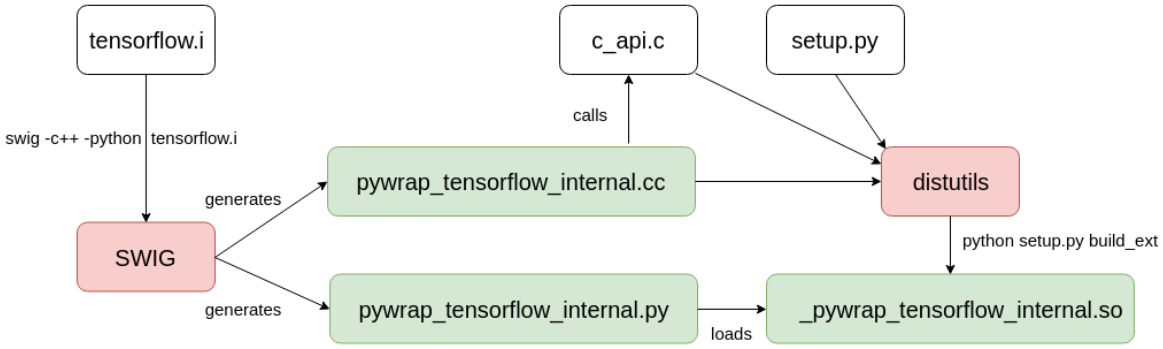


Figure 2.4: Outline of the automatic wrapper code generation process. Adapted from [3].

```
message NodeDef {
  // The name given to this operator
  string name = 1;
  // The operation name
  string op = 2;
  // Each input is "node:src_output" with "node" being a string name and
  // "src_output" indicating which output tensor to use from "node"
  repeated string input = 3;
  string device = 4;
  // Operation-specific graph-construction-time configuration.
  map<string, AttrValue> attr = 5;
  [...]
};

// Represents the graph of operations
message GraphDef {
  repeated NodeDef node = 1;
  [...]
};
```

Listing 2.2: Protocol buffer used for the nodes and graph representing the dataflow

library and forward any requests to the appropriate function.

Protocol buffers Another important mechanism for the interaction between the frontend and the backend are protocol buffers, used for passing serialised data from one language to another. Protocol buffers [50] are structured messages specified using a platform-independent, language-neutral syntax. They are composed of an arbitrary number of fields, each one with a given name and type. The files containing their definition are compiled to generate classes responsible for reading and writing the data in the target language during the build phase.

Listing 2.2 shows one important protocol buffer containing the definition of a node and a dataflow graph. The graph, along with its metadata, is constructed in the client API language (e.g. Python), and then it is serialised and passed to the backend.

2.2.2.2 User sessions

Now that the mechanisms used by the frontend client to request services from the backend are explained, we can highlight in more detail the process of constructing a Session, as well as its responsibilities within the framework, as illustrated in Figure 2.5. Our focus is on the `DirectSession` class implementation, which encapsulates both the master and the worker functionality because this class corresponds to the local implementation of TensorFlow.

Users interact with the TensorFlow backend by creating a `Session` object. The object's methods are exposed via the C API. Next, we describe the two main operations of a session:

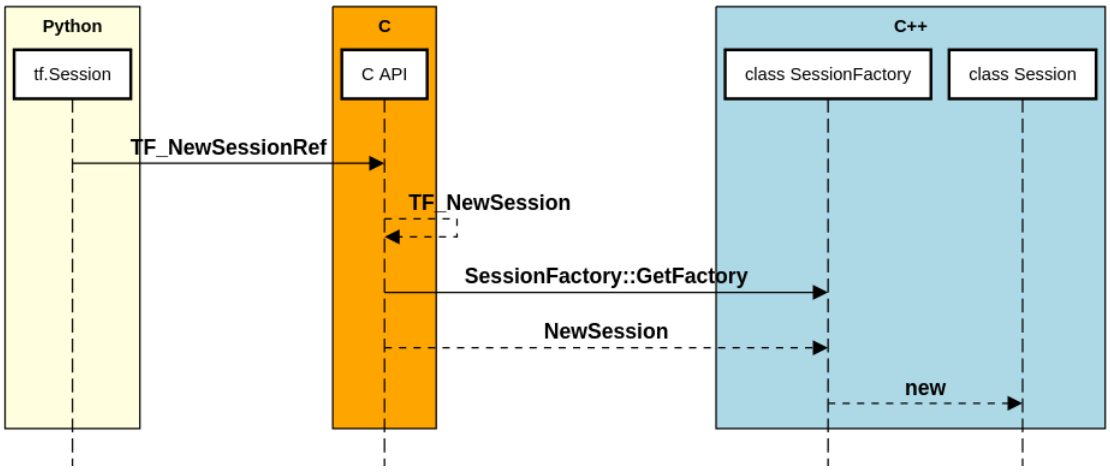


Figure 2.5: TensorFlow session creation workflow

creation of a session and running a session.

Creating a session. The `ExtendGraph` method, for example, is used to create or augment a dataflow graph with additional nodes and edges, as indicated by the protocol buffer passed as an argument to this method When the client program calls `tf.Session()`, the constructor calls directly into the shared library in order to get a new session handle via `TF_NewSessionRef`. Then, in the backend, the `SessionFactory` is called to instantiate the `Session` taking into account the supplied options. `NewSession` is actually a virtual method of the `SessionFactory` class that will be overriden by the subclasses.

The reason for making `NewSession` as a virtual method of the class `SessionFactory` is to enable the instantiation of different types of `tensorflow::Session` objects using `SessionFactory` polymorphism based on the options passed from the frontend. This abstract factory method inspects the `SessionOptions` object constructed beforehand. If the target is an empty string (the default), it indicates a `DirectSession`, the default option used for running local training on a single machine.

The constructor of `DirectSession` is mainly responsible for the creation of a thread pool to support concurrent execution and keeping track of the available computational devices. Additionally, `DirectSession` will also hold a map of the CPU scheduling threads (executors) responsible for running computations and returning the results.

Running a session. The `Run` method of `DirectSession` is the main path to the TensorFlow runtime and will handle one iteration of the training algorithm. This is the other primary method exposed by the C API to Python clients via the function `TF_SessionRun_wrapper`. Because the dataflow graph is lazily evaluated, the client needs to explicitly start the computation and pass a set of tensor output names that are requested to be computed, as well as an optional set of tensors as inputs. Using these arguments, the TensorFlow implementation will traverse the graph to find all the nodes that the result depends on, isolating them in a subgraph that will represent the scope of that run.

When the `Run` member function gets called, the executors for the given arguments (feeds and fetches, i.e. inputs and outputs) are either retrieved from a stored map which acts as a cache or constructed. When new executors are created, the `CreateGraphs` method gets called. `CreateGraphs` contains the logic which will determine the subgraphs, decide the placement of nodes on devices and finally partition the input graph across those devices based on the identified constraints. Then, the code path for creating executors implements optimisations on each graph partition such as common subexpression elimination, constant folding etc. Finally, it will instantiate new local executors which take ownership of the designated graph partition by calling `NewExecutor` and adding them to the cache.

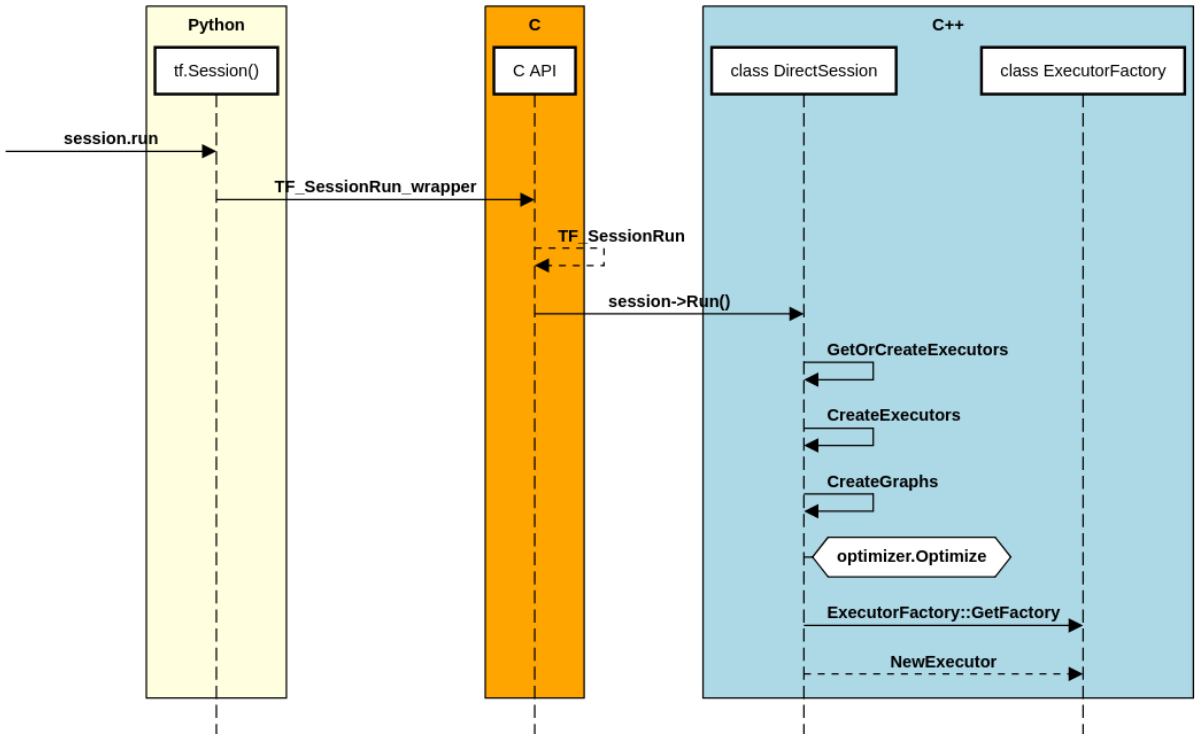


Figure 2.6: TensorFlow session run workflow when creating new executors

After the appropriate executors are obtained by the `Run` method, those executors are started in parallel and synchronised using a barrier to ensure all the independent computations have been completed before processing the results (i.e. outputs) back in the main thread.

2.2.2.3 Executors

The `Executor` class contains the logic for ordering nodes and operations, preparing inputs to kernels, processing outputs and propagating results after each kernel execution completes. As mentioned previously, there is one executor instantiated per graph partition and they will all run concurrently to speed up the computation. The three main functionalities that need to be provided by an executor are:

1. input and output processing for each operation implementation (i.e. each graph node)
2. inter-device data exchange (with the help of the additional send/receive nodes and rendezvous points for synchronisation)
3. complete execution of a graph iteration

Each executor will start traversing its subgraph internally. It executes the nodes in an order which yields a valid topological sort (without doing the sorting explicitly). The execution starts at the nodes which have an indegree of 0. These nodes are added into a queue of ready nodes, from where they are processed and dispatched via the `ScheduleReady` function. This function uses the thread pool previously created within the session to complete the execution of each node.

2.2.3 Crossbow

Crossbow [4] is a deep learning system designed to support resource-efficient and scalable small-batch training. What sets apart this system is the way to exploit unused GPU resources when the batch size is too small. It co-locates models on the same GPU, abstracting them by introducing the notion of *learners* whose number is adjustable during training.

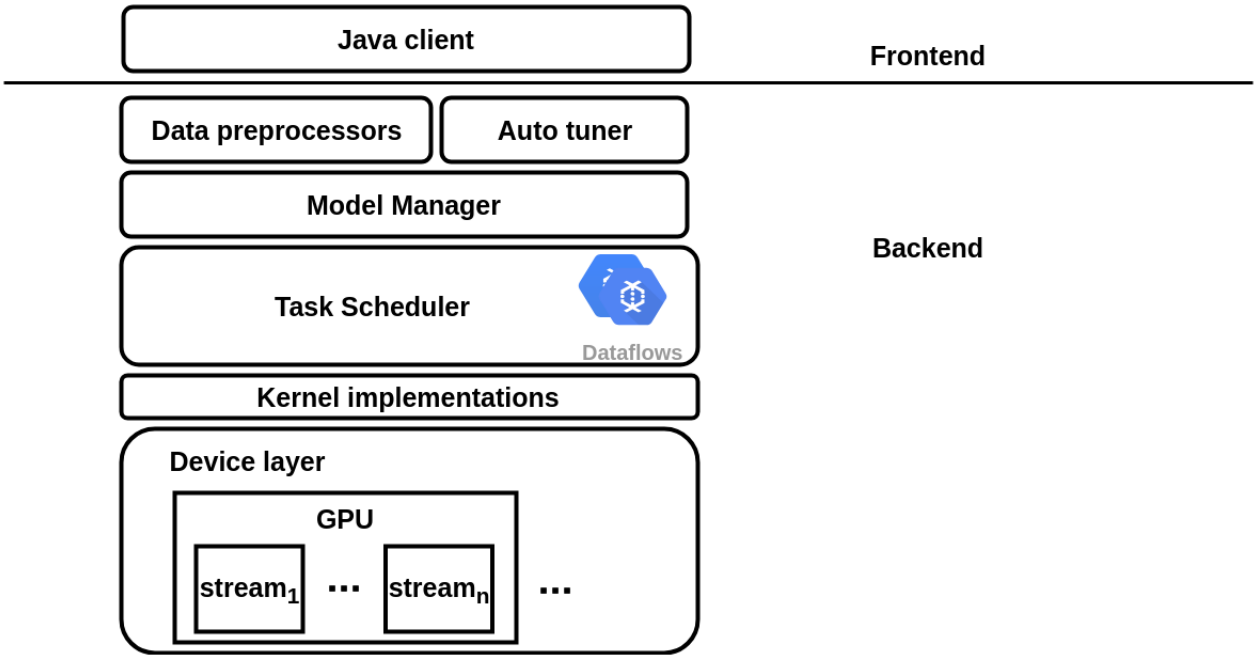


Figure 2.7: Crossbow system architecture

Model replication is handled transparently by the system. Crossbow supports a variety of training algorithms, most notably *Synchronous Model Averaging* (SMA) (Algorithm 2). The general contribution of Crossbow is that training with SMA achieves better hardware and statistical efficiency without the need for additional hyper-parameter tuning. If training can be reasonably accelerated irrespective of the parameters used, we achieve a decoupling between the system performance considerations and the theoretical convergence properties of the model.

2.2.3.1 Architecture

The architecture of Crossbow has unique design choices which make it suitable for training deep learning models with the SMA algorithm. The concept of a GPU stream is a first class citizen in the framework, since this is a very important aspect to share GPU resources effectively. All the model variables are replicated for each learner automatically, without the need to specify additional operators or separate dataflow graphs. It also has efficient mechanisms for synchronisation that enable learners on the GPU to access a shared average model.

Figure 2.7 shows a high-level diagram of the most important components of the system. Crossbow has a Java front-end that enables users specify their models. It is responsible for constructing the computational graph, generating a valid order of execution for operators and registering it with the backend, which is implemented in C. We refer to the backend as Crossbow's multi-GPU runtime library.

When comparing this client and runtime with TensorFlow's one, the frontend plays a bigger role in the overall system design. It is not only responsible for assembling the computational graph. It also supports the following:

- *Graph transformations.* It transforms a user-supplied graph into a sequence of operators (via a topological sort). Based on this ordering, it devises an offline memory plan to reuse if possible memory allocated for storing the inputs and outputs of operators.

- *Variable initialisation.* When registering the variable size and shape to the back-end, variables are already initialised in CPU memory (ready to be copied to GPU memory).

- *Task handling.* It has a task dispatcher that will assemble a task for a specific graph and dispatch it to the runtime; and a result handler to receive task results asynchronously.

The main components of the Crossbow runtime library are:

- *The model manager.* It handles a pool of model replicas and a pool of device streams to decide which resources should be used for the next iteration of the training algorithm.
- *The task scheduler.* It assigns computation or synchronisation tasks to GPU devices. After assigning tasks to resources, it handles task completion events coming from devices asynchronously (via callbacks).
- *The auto tuner.* It modifies the number of learners per GPU based on training throughput.

2.2.3.2 Tasks

The execution of a dataflow graph is comprised of two types of tasks that Crossbow issues and manages: *learning tasks* and *synchronisation tasks*. Those types of tasks runs on different GPU streams (learner streams and synchronisation streams, respectively).

Learning tasks. The learning task encapsulates that sequence of operators that results with the gradient values for a model replica. It takes an input batch of training examples and a model replica (picked by the model manager) and will produce a gradient. A learning task can execute on any of the available learner streams on the GPU where the model replica resides.

Synchronisation tasks. After a sequence of learning tasks, synchronisation starts. A per-replica synchronisation task computes the difference between the model replica and the central average model. This difference, together with the gradient for that replica, is used to correct the trajectory of the corresponding learner in the loss space. These local tasks only require read access to the central average model, which is why many local synchronisation tasks can run concurrently on different GPU streams.

Then, the central average model needs to be updated with the aggregate value of all differences. Crossbow allocates one copy of the central average model per GPU that is consistent across devices. Placing a barrier while learners synchronise their differences would prevent them from making any further progress (and, as a result, Crossbow would use GPU resources inefficiently). To overcome this problem, Crossbow overlaps these device-wide synchronisation tasks with subsequent learning tasks using separate synchronisation streams per device.

As highlighted in section 2.1.4, synchronisation tasks have two stages: inter-GPU and intra-GPU aggregation. The intra-GPU operators will wait all differences to be computed for all replicas in a particular device to aggregate them *locally*. The inter-GPU operators then perform an *AllReduce* aggregation [60] to combine the per-device differences; and then update the average model on each device consistently.

2.2.3.3 Task schedulers

In an iteration of the SMA training algorithm, the task scheduler would start a learning task for each replica in the system, assign it to one of its multiple worker threads and then issue subsequent synchronisation tasks. The worker thread that gets a task issues calls to the GPU to execute a sequence of kernels. An important requirement is that all those kernel calls to the GPU are asynchronous, such that the worker thread can return immediately to schedule the next task.

When a task finishes and returns the corresponding model replica to the pool of available one, the task scheduler will take the next input batch and repeat the process in a first-come, first-served manner. This mechanism improves hardware utilisation because the task scheduler does not have to wait for a particular replica to become available.

To deal with the data dependencies introduced by overlapping the learning and synchronisation tasks, the task scheduler uses GPU events. To ensure task τ finishes before another task τ', the task scheduler places an event generator after τ completes and an corresponding

event handler before τ' begins. GPU event handlers block any subsequent execution on the stream they were scheduled until notified. The task scheduler also uses the same event-based mechanism to process task completion events on the CPU and return the model replicas to the pool.

Overall, this task scheduler design makes the system extremely good for small learning tasks because it reduces the dispatching latency as much as possible.

2.3 Summary

The first take-away point of the background is that the choice of optimisation algorithm matters both in terms of accuracy and hyper-parameter constraints. SMA uses a variant of model averaging techniques to construct a central model from many independent replicas. By introducing the notion of a *learner*, it adds a level of indirection that enables us to separate the value of the batch size from the number of model replicas and the number of computational devices. This is an improvement over the inherent coupling of S-SGD.

The second main part of the chapter emphasises an important aspect, which is that the choice of system is as important as the choice of algorithm when considering scaling out the training process. No framework offers all the advantages, While Crossbow automatically replicates the dataflow graph internally to enable data parallelism and uses concurrency features present in modern GPUs, it does not support automatic differentiation. On the other hand, TensorFlow does not replicate the graph (i.e. the user must explicitly program this using a concept known as towers), but it performs automatic differentiation and offers a wide variety of operators. This integration aims to enhance both frameworks with features from their counterpart.

Chapter 3

TensorBow System Design

The key idea of our design is two-fold. First, we make Crossbow operators (e.g. a convolutional layer) pointers to TensorFlow ones. This paves the way for Crossbow to access a wealth of operators and compiler optimisations available in TensorFlow. Second, we make TensorFlow tensors for model variables (e.g. the weights of a convolutional layer) pointers to Crossbow ones. This allows TensorFlow to automatically access multiple model replicas per GPU without having to manage them explicitly. We then devise the TensorBow API, an interface which handles execution of a single kernel within the TensorFlow runtime. This helps make the exiting Crossbow parallel training abstractions compatible with TensorFlow.

3.1 Design Considerations

At a high level, both Crossbow and TensorFlow manage four main components: a set of *model variables*, a set of *operators* that update the model variables, and one or more *executors* that dispatch operators to run on a set of GPU *devices*.

We combine the two systems using TensorFlow as the base, because we want to design a system that supports all of TensorFlow's operators (and, thus, supports a wide range of applications and models). As a result, in our design the computational graph will be managed at least partially by TensorFlow since we must be able to retain its client API that allows users to write their applications and models. As explained in the previous chapter, the entry point for the user-defined dataflow graph of operators will be a TensorFlow `Session`, which in turn will interact with the other three components: model variables, operators and devices. We considered three alternative designs, summarised in Table 3.1.

Option 1. The first option we considered was to reimplement Crossbow's SMA algorithm (including auto-tuning the number of learners per GPU) using the interfaces and mechanisms provided by TensorFlow. Since a considerable amount of time had already been invested in understanding the TensorFlow framework, this option would result in no interactions, and thus no inter-operability issues, between the two frameworks. However, many functions of the TensorFlow executors would need to be changed to reflect performance optimisations available in Crossbow (e.g. overlapping kernel computations). It would be difficult to isolate gradient computation as it would require us to ensure that each executor gets access not only to different data, but also to different resources such as memory even on the same device. Furthermore, the operation of applying gradients would need to modified in a non-trivial manner for synchronisation with SMA.

Option 2. The second option was to simply take the graph passed to the TensorFlow and pass it to Crossbow. Crossbow would then take ownership of the computational graph, and use its own components to complete its execution, including managing the level of device setup and execution. This solution would require only an adapter layer to map the names of the kernels and operations from TensorFlow to the ones known by Crossbow, as well as to identify and

	Option 1	Option 2	**Option 3**
Operators	TensorFlow	TensorFlow	TensorFlow
Model variables	TensorFlow	CROSSBOW	CROSSBOW
Executors	TensorFlow	CROSSBOW	CROSSBOW
GPU Device Management	TensorFlow	CROSSBOW	TensorFlow

Table 3.1: Design options (i.e., which high-level components are managed by which framework). Option 3 is the one selected

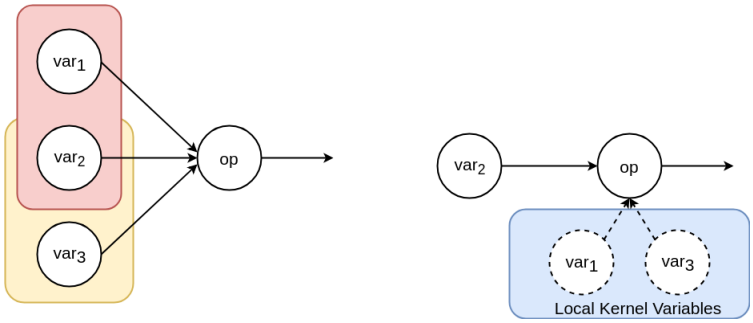


Figure 3.1: Operator op requires access to 3 variables, var_1, var_2 and var_3. Of the three, var_1 and var_3 are local variables and var_2 is a model variable. In order to translate the TensorFlow dataflow graph on the left to the CROSSBOW one on the right, there are many possible groupings of variables that could result in a dataflow misconfiguration

register the model variables. Despite being the simplest approach, it is limited by the operators CROSSBOW supports by default: although considerable, it remains a small subset of the the operators exposed to users by the TensforFlow client API.

Certain operators, for example convolution and batch normalisation, require access to hidden local variables as well as model variables. Both frameworks handle access to these variables differently. On one hand, TensorFlow automatically augments the dataflow graph to represent local variables as nodes that feed their output to these operators. On the other hand, CROSSBOW embeds them as pointers to memory within the operator. Translating the dataflow graph from TensorFlow to CROSSBOW requires to identify local variables in the graph and register them to the corresponding kernel in CROSSBOW. This is difficult because the metadata that identifies local variables in TensorFlow is not available up-front. It would add a lot of complexity to the translation as it is be very hard to distinguish between all the possible groupings, as depicted in Figure 3.1.

Option 3. The third design option follows the same pattern as the second one, but instead of giving away the control of the computation completely to CROSSBOW, TensorFlow retains the responsibility of managing the execution of an operator scheduled by CROSSBOW. Thus, we can still aim to support all TensorFlow operators by using a mutual dependency between the two frameworks.

For this idea to be possible, the principle behind operator execution must be similar across frameworks. Additionally, CROSSBOW must retain all the necessary information for the graph execution and wrap all the data structures and arguments using the classes that TensorFlow uses for graph execution. CROSSBOW would be responsible for managing the model variables and scheduling tasks, while requesting support from TensorFlow in terms of the actual computations being performed when it decides to execute an operator and for synchronisation among replicas.

After considering all design options, the best alternative was this last one for two reasons. First, we can take advantage of the richer and heavily optimised set of operations for different platforms already implemented in TensorFlow. Second, using the same implementations for

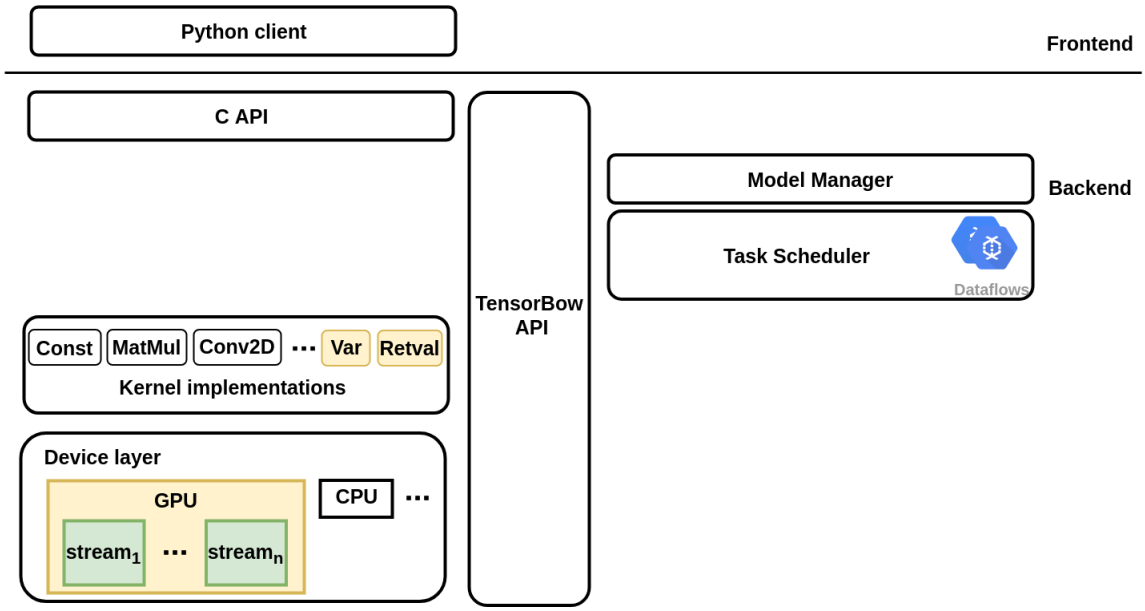


Figure 3.2: TensorBow unifies components from two systems

operators would offer a common ground for evaluating the efficiency of the training algorithms and obtaining a better estimation of the deep learning models and use-cases that benefit from using SMA compared to S-SGD.

In order for TensorBow to be fully compatible with the existing TensorFlow client API, certain TensorFlow components must remain unchanged when realising the workflow described in the previous argument; and some are replaced by components from the Crossbow runtime library. Figure 3.2 highlights the relevant components of each framework.

TensorFlow's Python client and its C API remain unmodified. So are TensorFlow's kernel implementations, with the exception of a few classes such as the ones for variable operators, argument passing etc. These TensorFlow components interact with Crossbow's model manager, the latter handling model replication. The new TensorBow API will replace TensorFlow's dataflow executors with Crossbow's multi-threaded task scheduler. Finally, TensorBow keeps the current device layer from TensorFlow, but it extends it to support multiple execution streams per GPU. Crossbow's task scheduler will enable dynamic selection of the target GPU stream per learning task (as opposed to static assignment, the default strategy in TensorFlow). Specific implementation details on changes to those components are provided in Chapter 4.

3.2 Workflow

Our proposed solution starts by extending TensorFlow's `Session` class. The newly created component, `TensorBowSession`, has a disjoint set of options (added to a custom protocol buffer), containing the desired logic for all the necessary steps and deciding where the separation of concerns would occur between TensorFlow and Crossbow. An outline of the new class hierarchy in TensorFlow is presented in Figure 3.3. The new class replicates some of the `DirectSession` functionality for session creation, dealing with the client's requests and processing the results. The main difference is how task execution is encapsulated within the `Session.Run()` function call.

TensorBow runs unmodified TensorFlow applications. A `Session` instance lets users drive a TensorFlow graph computation by calling the `Session`'s `Run` method multiple times. By providing users with an alternative session implementation, we keep the changes to the existing TensorFlow code-base minimal. We simply augment the existing functionality rather than introducing a lot of decision points to separate the code paths. Of course, we still need to introduce an additional adapter layer (namely, the TensorBow API, described in Section 3.3)

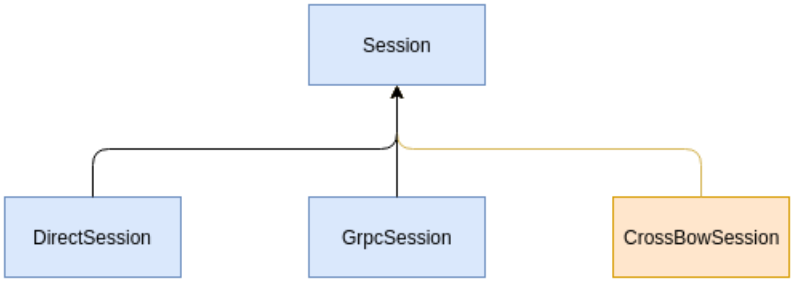


Figure 3.3: Updated TensorFlow `Session` class hierarchy

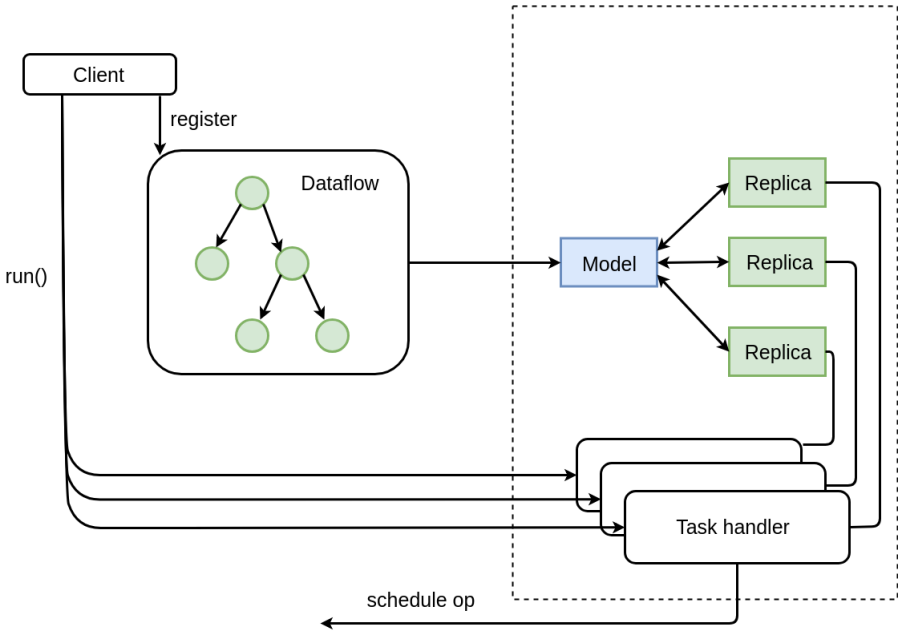


Figure 3.4: TensorBow's workflow for replicating the entire computation

and design ways to incorporate data structures from the coupled frameworks. But we can expose the whole integration as a cohesive solution, while having the freedom to decide our own task placement, operation execution order and resource management.

At the point where we create a dataflow graph in TensorBow, we can influence the number of partitions and the placement of nodes for the input graph. We make sure that no partitioning is taking place within a graph by default. We simply replicate the entire graph (containing all the operations that are to be performed on a GPU), since we want to parallelise the entire graph computation. Another important reason for this decision is that when partitioning takes place, TensorFlow introduces special `Rendezvous` points (realised by adding `Send` and `Receive` nodes in the graph to push to and fetch data from a different device, respectively). It is very challenging to distinguish between disconnected graphs that are only logically connected via these special nodes and other graphs that simply reside on a different device. Replicating and synchronising all of them would introduce unnecessary overheads that we avoid by letting users specify only a single graph.

Figure 3.4 shows the workflow of TensorBow. After the graph is optimised and returned, we register all the required information to Crossbow to initialise the library. This includes:

- Registering all the unique kernel objects from TensorFlow to Crossbow.
- Topologically sorting the graph, and registering it to Crossbow. This determines the execution order of the operators of a learning task.
- Getting the shape of each model variable and the buffer size that it requires in memory

(depending on the data type) and registering it to CROSSBOW. This information will be used by CROSSBOW to create multiple model replicas per GPU.

When the CROSSBOW runtime is initialised, a TensorFlow client can start submitting learning tasks for execution to CROSSBOW the same way it would normally invokes the default TensorFlow executors. CROSSBOW's task schedulers have total freedom to decide when to schedule each operator: on which stream and on which device. The TENSORBOW API permits them to perform all those necessary operations using TensorFlow's operator implementations.

The input data for each learning task can be provided by users in two different ways:

1. *A feed dictionary.* A feed dictionary is a Python dictionary of tensors, passed as input to the `session.run()` method. The dictionary is translated to a vector of CPU allocated tensors by the C API and passed to the `TensorBowSession`.

2. *tf.data.* This is a recent package that exposes different operators for data management (e.g. image decoding and cropping). These operators become nodes in the dataflow graph. Some operations can be can be executed on the GPU, but not all of them. Because we don't partition the graph, we require a GPU implementation for all the nodes that we need to execute. Thus, we can support only the GPU operators from the `tf.data` package.

In our prototype we focus on the feed dictionaries because it is easier to control without changing any operator logic. Using the user-supplied dictionary of inputs makes it easier to ensure that different model replicas get different data to work on and thus can be trained independently.

3.3 The TensorBow API

The next design consideration is the API that allows TensorFlow runtime to interact with the CROSSBOW runtime. For example, the TENSORBOW session must call CROSSBOW's functions for registering the dataflow graph; and TensorFlow variable operators require to access the model replicas managed by CROSSBOW. Similarly, CROSSBOW needs to call a function which schedules a particular kernel on a target device. These interactions are managed via the TENSORBOW API.

Figure 3.5 shows an example sequence of function calls that either framework makes for executing a dataflow graph. We show system interactions for one worker only, but in reality this is replicated via multiple execution threads to parallelise the computation within and across GPUs.

After the computational graph is registered in CROSSBOW, TensorFlow instructs CROSSBOW to execute a specific dataflow, identified by a unique id. In practice, TensorFlow will submit as many execution requests as model replicas which are available. CROSSBOW will assign a different thread to handle each request and execute them in parallel. TensorFlow will then block at a barrier waiting for the parallel executions to complete.

For each kernel in the graph, CROSSBOW will call the kernel method in TensorFlow via the API. TENSORBOW handles the arguments required to call each kernel. TENSORBOW will also select the right device and a compute stream therein, as instructed by CROSSBOW, and invoke the `Compute` method on the kernel. TensorFlow will transparently manage storing the inputs and outputs of each kernel in the graph based on the graph topology. If some operator requires access to the model variables, TensorFlow makes appropriate calls to CROSSBOW for fetching pointers to a model replica buffer, encapsulating the buffer in a `Tensor` object.

When CROSSBOW has executed the last operator of a task, it sends a notification to TensorFlow. TensorFlow can then return the result back to the Python client and proceed to the next wave of parallel execution requests. An alternative would be for TensorFlow's `Run` method to return immediately after scheduling all tasks, without waiting for tasks to complete. Returning results asynchronously to the client is supported by CROSSBOW, but not TensorFlow. This would cause problems with the logic of many TensorFlow applications.

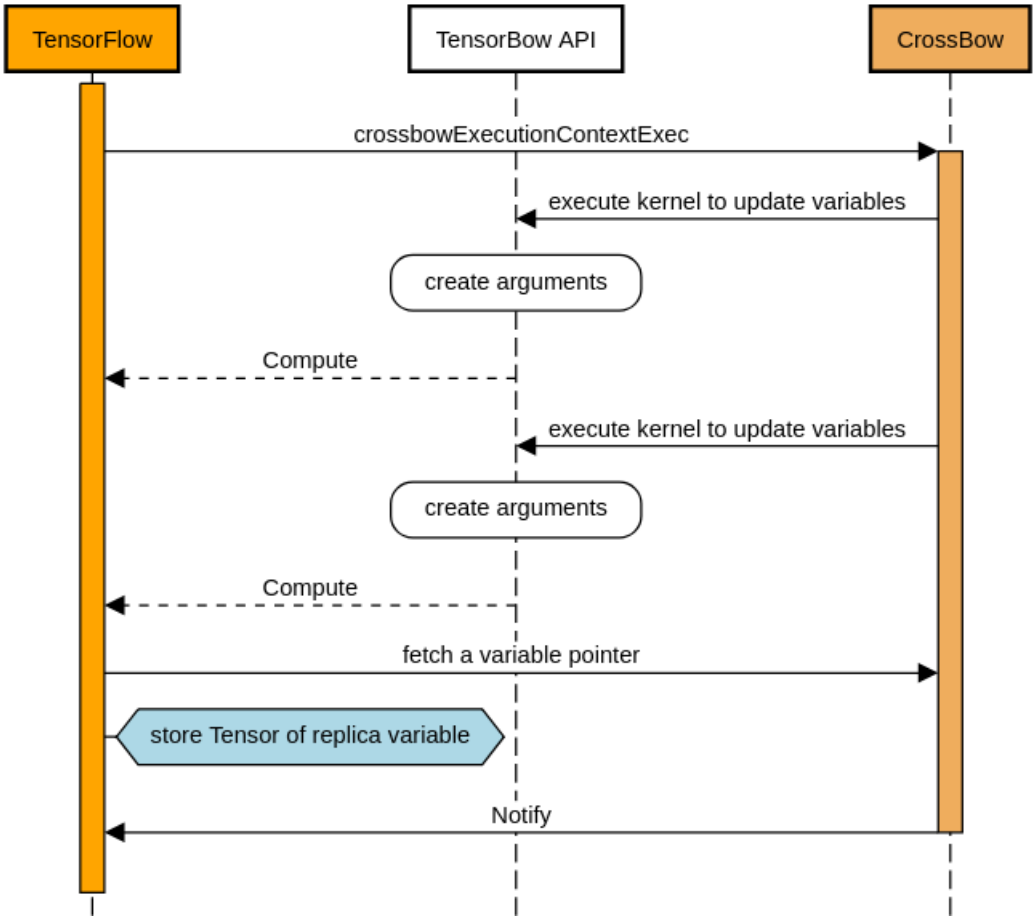


Figure 3.5: The TensorBow API enables interoperability between the two frameworks during the execution of a single learning task.

3.4 Summary

At a high level, what we are really interested in is using TensorFlow as the underlying implementation of both the learning task and the synchronisation task operators and let Crossbow be responsible for the scheduling decisions and handling model variable replicas. To enable interoperability between the 2 systems, we introduce 2 main components: `TensorBowSession`, an extension of the standard TensorFlow `Session` class, and the TensorBow API for managing any interactions initiated by Crossbow. Those additional components are expressive enough to expose an automated parallel training interface for unmodified TensorFlow programs.

Chapter 4

System Implementation

In this chapter, we provide details of the challenges faced when implementing TensorBow on top of TensorFlow. We begin by explaining how we added support for concurrent training of two or more replicas on the same GPU: Section 4.1 describes how we enabled TensorFlow to support multiple CUDA streams per GPU; and Section 4.2 explains how we enabled TensorFlow to support multiple model variable replicas per GPU. In Section 4.3 we describe how we overcame concurrency issues when using the aforementioned newly added features at run-time. Finally, in Section 4.4 we describe how we efficiently synchronise model replicas *within* and *across* devices to reduce the data movement overhead.

4.1 Device Management

4.1.1 Initialisation

The first fundamental limitation we encountered when investigating the open-source implementation of TensorFlow is its inability to instantiate and use more than one GPU stream for computation. The default implementation sets the maximum number of streams per GPU used for kernel computations at 1. This value is hard-coded in the constructor of the `GPUDevice` class (the object responsible for stream initialisation and device management logic), as shown in Listing 4.1.

In practice, the `GPUDevice` constructor creates one *stream group* per GPU device (see Listing 4.2). Within a stream group, TensorFlow creates a dedicated stream for computation, one for host-to-device data transfers, one for device-to-host data transfers and one for device-to-device data transfers. So, by default, it is possible to have up to four concurrent operations per GPU device: one kernel computation and three data transfers.

We changed the class to create multiple stream groups so that we can train multiple model replicas in parallel, one per group. There are two reasons for creating new groups rather than multiple computation streams per group. First, there is very low overhead in calling the CUDA interface at initialisation time to instantiate a new stream. So, even if the extra data transfer streams are not used, the impact is negligible. Second, most of the stream synchronisation logic in TensorFlow expects a stream group as a parameter, so changing the abstraction of a stream group would not keep the device layer code compatible with the existing implementation. We enable mutiple multiple stream groups per device by adding a new field to the `GPUOptions` protocol buffer message that is specifying their number. The value of the private member `max_streams_` stored on the `BaseGPUDevice` class, previously hard-coded to 1, is now read from the `GPUOptions` and set inside the constructor of the `BaseGPUDevice` parent class.

The default strategy of TensorFlow with multiple stream groups is to partition the graph and assign different branches to different stream groups. The assignment is done in round-robin order based on the following heuristic:

- Nodes with 0 inputs (e.g. constants) are always executed on a different stream.

```
class GPUDevice : public BaseGPUDevice {
 public:
  GPUDevice(const SessionOptions& options, const string& name,
            Bytes memory_limit, const DeviceLocality& locality,
            TfGpuId tf_gpu_id, const string& physical_device_desc,
            Allocator* gpu_allocator, Allocator* cpu_allocator)
      : BaseGPUDevice(options, name, memory_limit, locality, tf_gpu_id,
                      physical_device_desc, gpu_allocator, cpu_allocator,
                      false /* sync every op */, 1 /* max_streams */) {
    if (options.config.has_gpu_options()) {
      force_gpu_compatible_ =
          options.config.gpu_options().force_gpu_compatible();

    }
  }
};
```

Listing 4.1: Restrictive stream initialisation in `GPUDevice` class. By default `max_streams` is one.

```
class BaseGPUDevice : public LocalDevice {
    [...]
    private:
      struct StreamGroup {
        se::Stream* compute = nullptr;
        se::Stream* host_to_device = nullptr;
        se::Stream* device_to_host = nullptr;
        gtl::InlinedVector<se::Stream*, 4> device_to_device;
      };
    [...]
}
```

Listing 4.2: TensorFlow's stream group data structure

- If possible, a node will be placed on the same stream as one of its inputs to avoid inter-stream dependencies.
- A node with lots of neighbours is placed on a new stream to make the consumers of its output run in parallel.

This heuristic does not yield many benefits. Most modern deep neural networks have a low branching factor (e.g., 2 for the ResNet family of networks [12]) and there are not many operators within a branch to execute in parallel before an inter-stream dependency presents itself. Frequent stream synchronisation is also very costly. Our approach on the other hand, is more beneficial because we introduce inter-stream dependencies only at the end of the dataflow graph execution.

4.1.2 Dynamic device context selection for GPUs

Since we are interested in reusing the device layer that TensorFlow provides, another implementation choice is to delegate the responsibility of initialising the devices to TensorFlow.

The Crossbow components that would normally provide this logic must be adapted such that they will only register the device information in their local data structures. These operations must happen in order to enable Crossbow to perform non-device management operations (i.e. allocating memory or launching kernels are examples of non-device management operations). Also, it avoids the problem of having each framework operate on a different working set of device streams. Informing the Crossbow model manager about the number of available

```
class Device : public DeviceBase {
 public:
  // Performs the actual compute function. Subclasses may override this function
  // if they wish to perform some initialisation before each compute.
  virtual void Compute(OpKernel* op_kernel, OpKernelContext* context) {
    op_kernel->Compute(context);
  }

  // Provides access to a target device context
  virtual Status GetDeviceContext(int id, DeviceContext** device_context) {
    return Status(error::UNIMPLEMENTED, "Device did not override this method");
  }

  // Provides access to an independent memory allocator for this device
  virtual Status GetDeviceAllocatorForStream(int id, Allocator** allocator) {
    return Status(error::UNIMPLEMENTED, "Device does not support this method");
  }

  // Returns the resource manager associated w/ this device.
  virtual ResourceMgr* resource_manager() { return rmgr_; }
  [...]
}
```

Listing 4.3: TensorFlow's `Device` abstract class with its new methods, `GetDeviceContext` and `GetDeviceAllocatorForStream`

compute streams per device is the last stage of the CROSSBOW execution context initialisation since we must wait for this information to become available in TensorFlow.

Listing 4.3 shows the common TensorFlow interface for all devices. It includes a `Compute` method, which by default just forwards each call to the `OpKernel` class.

However, the `BaseGPUDevice` class overrides the default logic to enable the right device context before the computation takes place. A GPU device context is just an abstraction over the GPU streams that uses them without taking ownership of the pointers or interacting with the `StreamGroup`. After enabling the right device context, the `Compute` method of the `BaseGPUDevice` class waits on any dependent streams to finish the enqueued work. This step is necessary only if the current operation has a different device context than some parent node in the computational graph. Then, GPU operations are scheduled by making a request to the `StreamExecutor`, an abstract wrapper around the CUDA driver.

The problem with this workflow is that in TensorFlow, when multiple stream groups are present, an execution thread requests a map which specifies which operator runs on which device context. This happens at construction time and remains statically allocated for the duration of the entire program. In TENSORBOW we want to do this assignment at computation time because we don't want to impose an execution order across streams. To do this, we need to change the device layer to get a target device context on demand, for every kernel execution.

Then, we can specify a target device context instead of the default one based on the arguments that accompany each `OpKernel`. To control this placement, we augment the existing interface from the `BaseGPUDevice` to expose a target device context. Then, each task handler thread from CROSSBOW executing a kernel on the GPU gets to control which device context it needs to enable via the arguments provided for the computation. The `GetDeviceContext` method was added on the `Device` class for this purpose.

Listing 4.4 shows how this new function is implemented by the `BaseGPUDevice` class (it includes performing bound checking). It is important to note that each device context is reference counted, so we increase the reference count before returning the object. The caller thread will then be responsible for releasing the resource.

Finally, when the GPU executes the computation, it knows which request has come from a

```
class BaseGPUDevice : public LocalDevice {
 public:
  [...]
  Status GetDeviceContext(int stream_id, DeviceContext** device_context) override {
    CHECK_LE(0, stream_id);
    CHECK_LT(stream_id, device_contexts_.size());
    device_contexts_[stream_id]->Ref();
    *device_context = device_contexts_[stream_id];
    return Status::OK();
  }
  [...]
}
```

Listing 4.4: `BaseGPUDevice` implements the new `GetDeviceContext` method

Crossbow task handler thread and it uses this information to bypass the synchronisation check with other streams. This is now safe as we will isolate the whole graph computation inside a stream, so any previous nodes have consumed their inputs and produced the outputs using the same underlying stream via the device context, so the data is available.

4.2 Model variables

The next major implementation component focuses on model variables. While the GPU device memory associated with multiple model replicas is allocated and managed by Crossbow, we need a way to enable TensorFlow to differentiate between model replicas, fetch the correct underlying memory buffer for each replica and update them accordingly with the gradients resulting from the computation of a learning task.

4.2.1 Variable names

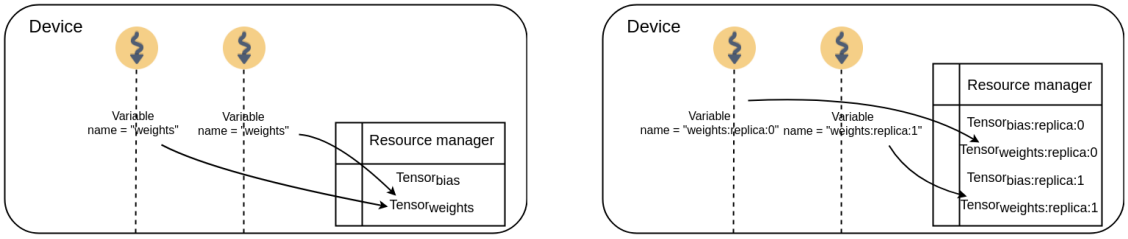
TensorFlow manages resources, including model variables, with the help of a *resource manager* (the `ResourceMgr` class). The resource manager stores resources according to their unique name (the resource name, plus the device name where they reside). Each device has an associated resource manager. Different threads can share state if they define a resource with the same name on the same device.

In the case of model variables, we want to avoid having different parallel dataflow computations access and operate with the same underlying model replica buffers. Model variables appear in the computational graph as a special node whose `Compute` method is a stateful kernel that accesses memory resources (basically, tensors). Each variable is stored in the device resource manager based on the name given by the client Python program (and communicated via a protocol buffer). The kernel simply invokes a `LookupOrCreate` method on the resource manager to create a new entry (a new tensor) for this variable name or retrieve its contents (some previously allocated tensor).

Knowing how the resource manager indexes elements, we can associate a different name for each model replica variable to distinguish between resources. Since Crossbow allocates a unique id for each replica, this identifier is a good candidate to include in the updated name. This additional logic is added to the `VariableOp` kernel class. Figure 4.1 shows how this enhancement achieves the desired result.

4.2.2 Variable buffers

A `Tensor` class in TensorFlow represents a tensor, associating a data buffer (a `void *` pointer) with a shape (i.e. the dimensions of a multi-dimensional array). For data buffers, TensorFlow uses a `TensorBuffer` class abstraction. In TensorBow, the tensor buffer should point to the



(a) Problem: Multiple execution threads iterating over the same graph and using the same variable name end up accessing the same tensor from the resource manager.

(b) Solution: Suffix every variable name with a unique replica id inside the `VarOp` operator and register each one with the resource manager.

Figure 4.1: TensorBow replicates model variables in the resource manager for each replica to prevent aliasing.

corresponding model replica variable managed by Crossbow. This integration required to extend the `Tensor` and `TensorBuffer` classes with setter methods.

The `TensorBuffer` class was initially an abstract class with virtual methods to access the data buffer pointer and its size (the `data()` and `size()` methods, respectively). This was changed in TensorBow to reduce the overhead of virtual function calls, which is important for operators that call many times the function for getting the data (this optimisation was also included in TensorFlow v1.13). The `Tensor` setter could then delegate the calls to the `TensorBuffer` and change its private data pointer. Two more modifications were made:

- TensorFlow's `TensorBuffer`s are reference-counted. So, we introduce a `modified` boolean field to distinguish between buffers allocated by TensorFlow (and, thus, can be safely deallocated) and buffers managed by Crossbow.
- We introduce an `index` field to associate a `TensorBuffer` with a particular Crossbow model replica variable. This index is used to fetch the model variable buffer associated with a specific node in the dataflow graph (identified by that index).

Listing 4.5 shows the modified `TensorBuffer` class after all the modifications, which now has the responsibility of storing the contents directly. Changing the underlying `Tensor` buffers has the only downside of losing the ability to track allocations when running TensorFlow in debugging mode, since TensorFlow expects this invariant all buffers are allocated using a specific allocator. But model variables are the only buffers managed by the Crossbow library, allocated at initialisation and freed at the end of training. So we do not introduce any memory leaks.

4.2.3 Variable initialisation

Crossbow has internal mechanisms to deal with model variable initialisation in an elegant way. Given a CPU buffer allocated on the CPU, populated with initial values, Crossbow handles model replication and copying to GPU memory transparently.

We let TensorFlow initialise model variables on the CPU, and pass them to Crossbow. To achieve this, we influence the placement of the TensorFlow operators that run model variable intialisation and place them on the CPU. Since TensorFlow is able to run the initialisation on any target device directly, we can run the subgraph using the standard TensorFlow execution management. We use a heuristic to identify the TensorFlow model variable initialisation subgraph based on the presence of model variables and optimiser nodes.

After the initialised operations are executed on the CPU, the resource manager for that device will hold tensors that are initialised approriately. We then do a pass over the CPU registry manager, access those tensor buffers and pass them to Crossbow model manager to use them as the starting point for all model replicas.

```
// Interface to access the raw ref-counted data buffer.
class TensorBuffer : public core::RefCounted {
 public:
  explicit TensorBuffer(void* data_ptr)
      : data_(data_ptr), modified_(false), index_(0) {}
  ~TensorBuffer() override {}

  // data() points to a memory region of size() bytes.
  void* data() const { return data_; }

  void set_modified(bool modified) { modified_ = modified; }
  bool modified() const { return modified_; }

  void set_index(int index) { index_ = index; }
  int index() const { return index_; }

  virtual size_t size() const = 0;
  [...]
 private:
  void* data_;
  bool modified_;
  int index_;
};
```

Listing 4.5: Updated `TensorBuffer` class with the `data_`, `index_`, and `modified_` fields. The class now stores the data pointer directly.

4.3 Running learning tasks in parallel

4.3.1 TensorBow Session and input management

In order to build a system that is able to execute learning tasks in parallel, a new component which controls the options and parameters for dispatching tasks must be introduced. In chapter 3, we motivated the need for introducing a new `TensorBowSession` class. Its design follows the same principles as the `DirectSession` implementation, except it does not support any form of partial runs. Partial runs allow the user to continue the execution of a graph with a different set of feeds and fetches, which is not a major concern when training deep learning models for long durations. Additionally, the API for partial runs is still experimental and subject to changes, so we decided to focus on the core functionality.

The constructor of the `TensorBowSession` must initialise the CrossBow execution context, required in all the subsequent calls to the CrossBow API. It contains logic for controlling the state of the Crossbow engine in addition to its own session state.

Apart from handling user requests and dispatching tasks to Crossbow, another important responsibility of the `TensorBowSession` class is managing model inputs and outputs. The `TensorBowSession` has access to a vector of CPU allocated tensors which are given as inputs to the `Run` function via the Python argument `feed_dict`. Because it is aware of the number of model replicas managed by Crossbow and the number of task handler threads available, it will ensure that the data gets sharded so that each worker gets a different partition to work on.

After sharding, the data is transmitted using the TensorFlow call frame structure. The corresponding kernel functions for passing arguments and returning results (namely `ArgOp` and `RetvalOp`) are modified to understand if the `Compute` request comes from the TensorBow API and take care of data movement to and from the GPU.

```
/* TensorFlow */
class OpKernel {
 public:
  explicit OpKernel(OpKernelConstruction* context,
                    std::unique_ptr<const NodeDef> node_def);

  virtual void Compute(OpKernelContext* context) = 0;
  [...]
};

/* CrossBow */
typedef void (*crossbowKernelFunctionP)(void *);
```

Listing 4.6: Abstract class for kernel implementations in TensorFlow (above), compared to CROSSBOW's one (below).

```
typedef void *crossbowTensorflowKernelP;

typedef struct crossbow_kernel {
  int id;
  char *name;
#ifdef TF_CROSSBOW_INTEGRATION
  crossbowTensorflowKernelP tfkernel;
#else
  crossbowKernelFunctionP func;
#endif
} crossbow_kernel_t;
```

Listing 4.7: A CROSSBOW kernel stores a pointer to a TensorFlow kernel object

4.3.2 Kernel execution

In CROSSBOW, a kernel is registered as a function pointer. In TensorFlow, every kernel inherits from the `OpKernel` abstract class and overrides the virtual `Compute` method to provide an implementation. Getting a pointer to the `Compute` method directly is not a straightforward task, so having access to the actual object instantiated by TensorFlow would yield a more robust solution. Listing 4.6 shows the representation for kernels employed by each framework. We need to find a suitable way for storing the TensorFlow kernel implementations and constructing this additional `OpKernelContext` class instance which acts as a wrapper for all the arguments to the `Compute` method.

TensorFlow instantiates the kernels when new executors are added into the session cache. Those executors have a very efficient way of storing an immutable version of the computational graph in a `GraphView`, coupling together graph nodes and the newly created kernels into a `NodeItem`.

In order to get access to the kernels for each operator, we let the session create an executor for each graph corresponding to a GPU device. Then, we inspect each `NodeItem` from the corresponding executor using the graph topological order. This provides access to the kernel instantiation which we can then register in the CROSSBOW runtime. A separate field had to be introduced in place of the C function pointers. The `tfkernel` member has the type `crossbowTensorflowKernelP` (i.e. `void*`).

At this point, CROSSBOW has the complete information about the model variables and the dataflow operators that need to be executed. The task scheduler threads can start execution as soon as the session schedules any computations. Each thread will iterate over the dataflow and call the TENSORBOW interface function when they need to schedule a particular operator.

```
class OpKernelContext {
 public:
  struct Params {
    // The op kernel being computed.
    OpKernel* op_kernel = nullptr;
    // The device on which the kernel is running.
    DeviceBase* device = nullptr;
    DeviceContext* op_device_context = nullptr;
    // CrossBow kernel arguments
    crossbowStreamP stream_info = nullptr;
    // Make the OpKernel use a special allocator
    Allocator* allocator = nullptr;
    [...]
  };
  [...]
}
```

Listing 4.8: New members of the `OpKernelContext` class, `stream_info` and `allocator`, used to isolate kernel computation to specific GPU streams

Crossbow creates a unique stream data structure for encapsulating the information required by any kernel. Our TensorBow interface keeps this structure, but it needs to make this data structure available to the TensorFlow kernel objects. To achieve this, we augment the parameters passed to each kernel execution with the `stream_info` pointer to the Crossbow data structure. Listing 4.8 shows the modified `OpKernelContext` class for passing parameters to TensorFlow kernels. This method ensures that any information from the Crossbow runtime is immediately accesible within TensorFlow operators and is used to decide when additional operator logic should execute (e.g. replicating model variables).

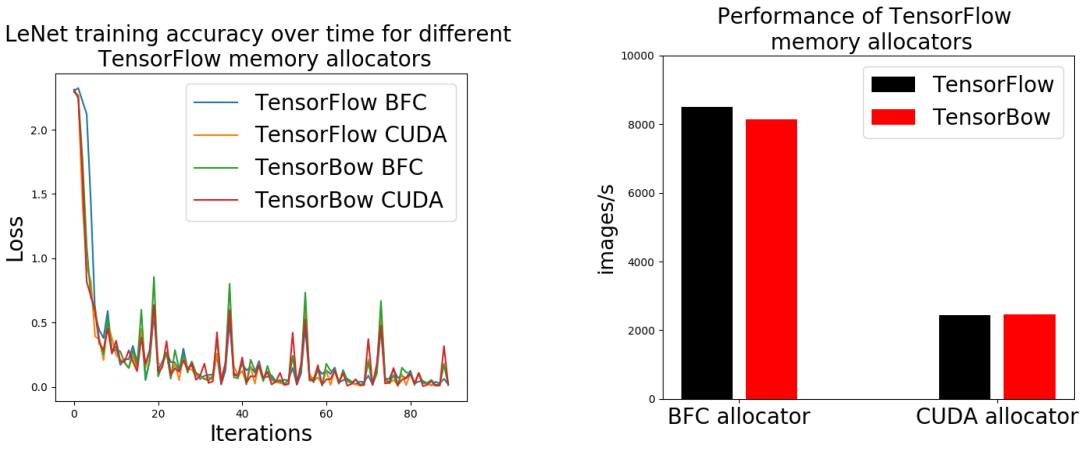
One last consideration in terms of kernel execution is handling both synchronous and asynchronous kernels. Subclasses of `AsyncOpKernel` expose both a `ComputeAsync` and a `Compute` method which awaits the termination of the operation. Therefore, the TensorBow API can call only the device `Compute` method to launch the kernel on a target stream. This enables us to handle send and receive nodes. Although we avoided inter-device data transfers by skipping graph partitioning, TensorFlow has no kernel for casting host memory to device memory. So, at graph construction time, it inserts send and receive nodes to perform this type casting. There is no risk in executing the receive node synchronously because the consumer will not block waiting for the tensor. The data has already been produced by a previous operation.

4.3.3 Stream-safe memory allocation

A limitation of TensorFlow is that its GPU memory allocator cannot be used across streams on the same GPU [49]. The default allocator is the *Best-Fit with Coalescing* (BFC) allocator. TensorFlow creates one such allocator per GPU. The BFC allocator is responsible for managing the entire device memory performing caching, memory defragmentation and buffer reuse to improve performance.

When an operator requests from the BFC allocator to reserve memory to store its output tensor, the allocator can decide whether to reuse an existing memory buffer, previously allocated for a different operator but currently not used by any other; or, if none of the previous buffers fit, allocate a new one. New allocation requests can also be directed to `GPUcudaMallocAllocator`, a wrapper for CUDA driver memory allocation calls. CUDA driver calls are thread-safe. Memory reuse decisions, however, by the BFC allocator are not. This is a problem when multiple threads try to schedule dataflow operators concurrently on different streams on the same GPU, since their output buffers can get mangled (e.g. overlap).

We had two options to overcome this problem. The first one was to avoid any memory



(a) Effect on statistical efficiency

(b) Effect on hardware efficiency

Figure 4.2: Comparison between memory allocators in TensorFlow

reuse and use directly the `GPUcudaMallocAllocator` as the default allocator. The second one was to create one allocator per GPU stream, each responsible for managing a separate memory partition. We performed a comparison between the two types of allocators. Figure 4.2 shows the results. We observe that both allocators do not affect model convergence (Figure 4.2a), but their performance is very different: using the `GPUcudaMallocAllocator` results in 4× decrease in throughput compared to the `BFCAllocator`. This happens because *all* calls to the driver are synchronous and *every* operator makes one. Allocation time also has a high variance [58].

We implemented the second option: replicate the `BFCAllocator`. We split the device memory into as many partitions as the number of stream groups, and instantiate one for each partition. The `GPUDevice` class manages an array of allocators (accessible by a getter method). There is a one-to-one mapping between an allocator and a stream group. The `OpKernelContext` class, from where all memory allocation requests originate when a kernel runs, is also extended to include a pointer to the allocator associated with the stream the kernel runs on (Listing 4.8).

4.3.4 cuDNN handlers

NVIDIA cuDNN [59] is a library which provides GPU implementations of many primitives for deep neural networks such as convolutions, pooling layers, and activation functions. To use this API, an application has to initialise a *handler* to the cuDNN library that directs subsequent library function calls to a particular stream on a device.

During device initialisation, TensorFlow creates one cuDNN handle per device, stored in a corresponding `CudnnAccess` class. By default, threads access the handler via the `GetHandle` method. This method will return a `CudnnHandler`, which is a RAII (Resource Acquisition Is Initialisation) wrapper for the handler, protected by a mutex. This hinders performance when running training multiple replicas per GPU because all cuDNN library calls made by different threads will be eventually serialised.

To overcome this limitation, we preallocate multiple cuDNN handlers per GPU and store them in the `CudnnAccess` class. The class now also caches requests for getting a cuDNN handler coming from a particular thread (at the time associated with a particular GPU stream), keeping a one-to-one mapping between handlers and GPU streams. By associating GPU streams with handlers, cuDNN calls from different training tasks can run concurrently.

4.4 Parallel model synchronisation

We have enabled concurrent training of multiple model replicas on the same GPU. But all the replicas need to be synchronised in an efficient with the SMA algorithm.

For each model variable, TensorFlow inserts an *optimiser* operator in the computational graph. The optimiser takes the model variable tensor and the gradient tensor as inputs and applies the gradient to the model variable. After using the gradient, TensorFlow will mark this node as complete and may reuse the memory allocated for its inputs. This is problematic because it requires us to develop a solution that either stores a copy of the gradient in memory or does the synchronisation among replicas on a per-variable basis. When an execution thread reaches the point of executing an optimiser, we must perform all the necessary computations for the SMA algorithm for this model variable replica since the gradient buffer may be reused for one of the next operators.

For SMA (see Chapter 2, Algorithm 2), the optimiser's `Compute` method performs the following local computations in order:

1) Calculates the difference between the current model variable and the average model variable

2) Updates the model variable with the gradient; and

3) "Corrects" the model variable (i.e., it scales the difference computed in Step 1) and updates the model variable again).

SMA then requires to aggregate the differences from all replicas. With more than one replica per GPU, the cost of inter-GPU synchronisation scales with the number of model replicas. So, we take advantage of locality and perform first an aggregation within a device (*intra-GPU* synchronisation) and then choose only one execution thread per GPU to aggregate values across devices (*inter-GPU* synchronisation). Threads that operate on the same device implement a leader election mechanism to determine which one performs the intra- and inter-GPU synchronisation. The leader is the last thread that computed the difference for a variable (thus, making sure that results from other threads are in place).

Figure 4.3 shows this process for two model replicas trained on the same device. The second thread that updates its weights variable is the last, so it becomes the leader and performs the local aggregation: it accesses all replicas via Crossbow's model manager and computes the sum of the differences for all the models located on the same device. It stores the result in a newly created tensor which is used as input for an NCCL *AllReduce* operator [60] that aggregates the differences across GPU devices. This operation is executed on a separate synchronisation stream to overlap the *AllReduce* with the remaining computation. The process is identical for all variables, only the leader is different. For the bias variable, for example, it is the first thread that performs intra- and inter-GPU synchronisation steps.

4.5 Summary

The main challenges in implementing TensorBow are related to the fact that many crucial TensorFlow components for GPU training (e.g. memory allocators, cuDNN handlers) were designed under the assumption of training one replica per GPU. So, it was unsafe to use these components concurrently by two or more GPU streams (and as a result, by two or more CPU threads, scheduling operators on the same GPU).

We started from a TensorFlow feature that is currently considered "experimental" by the community, that is, the ability to create multiple stream groups per GPU, and we extended it to use those streams efficiently to train multiple replicas per GPU. With our changes it is now safe to train multiple model replicas on the same GPU in TensorFlow and synchronise them efficiently.

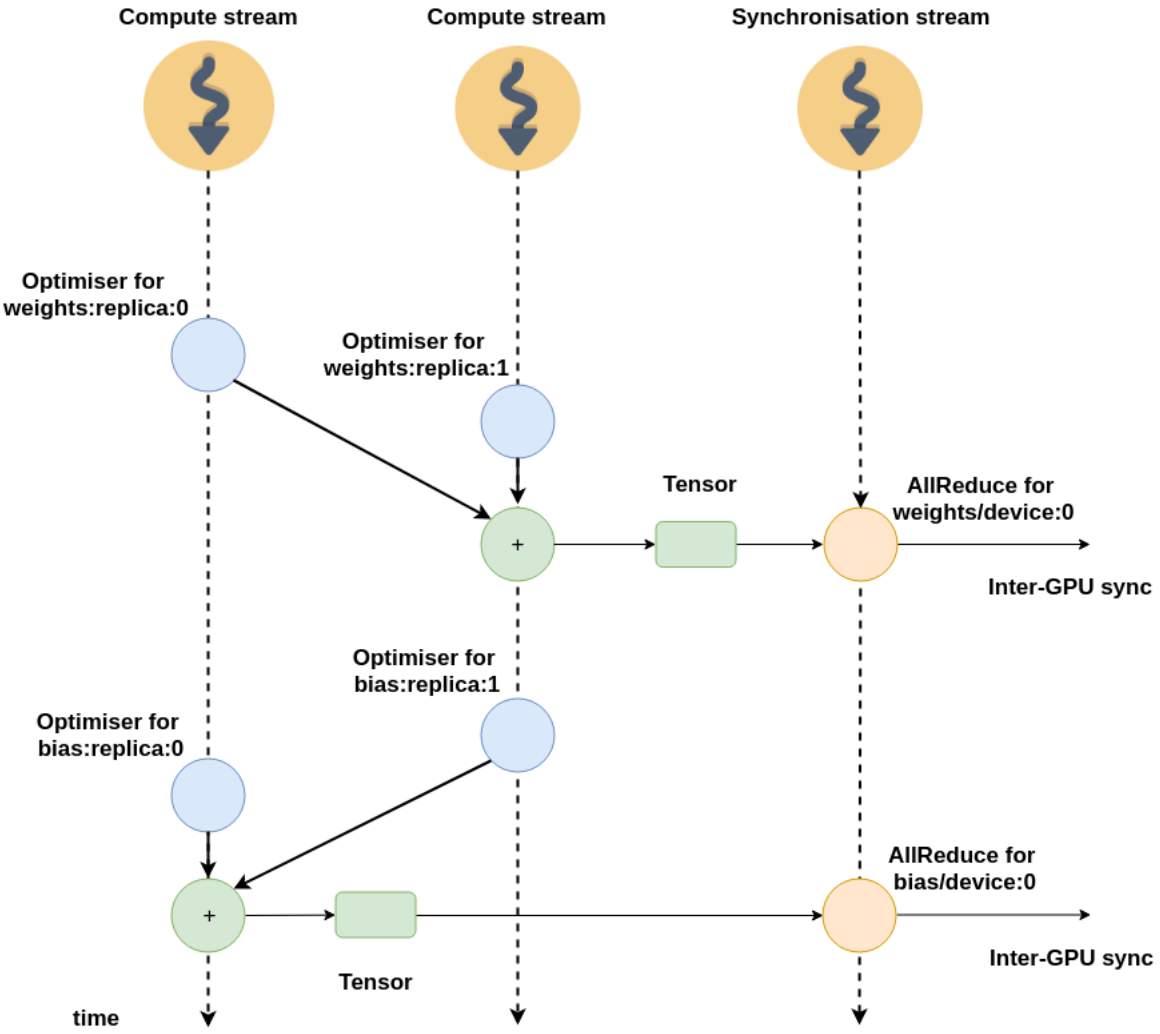


Figure 4.3: Intra-GPU and inter-GPU variable synchronisation in TensorBow

Chapter 5

Evaluation

The results in this chapter aim to support our design and implementation decisions presented in the previous chapters. We first argue that training with a large batch size hinders statistical efficiency. On the other hand, a small batch size takes a long time to reach the target accuracy. We show that training multiple model replicas on a small batch size improves the time-to-accuracy. TensorBow has better performance because of the SMA algorithm and the improved hardware efficiency (Section 5.2). We show how the system scales in terms of throughput when the number of GPUs increases (Section 5.4), when the number of model replicas increases (Section 5.3) and when the batch size increases (Section 5.5). Finally, we analyse the effectiveness of our incremental synchronisation approach (Section 5.6).

5.1 Experimental Setup

Experiments are deployed on a server with 2 Intel Xeon E5-2640 v4 2.4 GHz CPUs (20 CPU cores in total) and 64 GB of RAM. The server has 4 NVIDIA GeForce GTX Titan X (Pascal), each with 3,072 cores and 12 GB of RAM, connected via PCIe 3.0 (×16). It runs Linux kernel 4.15 with the NVIDIA driver 418.39. Experiments run in a Docker container with CUDA 10.0 and cuDNN 7.4.2. Our implementation of TensorBow extends TensorFlow version 1.12.

Workloads. We use different neural networks in our evaluation with varying depths and model variable sizes. They are summarised in Table 5.1. Logistic regression is the smallest of the examples, but represents a type of workload that enables fast initial tests. The rest of the models chosen are convolutional neural networks. LeNet is a conventional neural network with a small memory footprint. The ResNet family of neural networks is one the most prominent deep neural network architecture and is chosen because it is much more compute intensive than the others.

The models used for every experiment to compare the two systems are identical: we use the same hyper-parameters and variable initialisation. In all the experiments, we refer to a given batch size as the batch size used to compute the gradients for each instance of the model.

5.2 The effect of small-batch training on convergence

The first set of experiments shows the effect of the batch size on model convergence. We show that the best batch size is not always the one that yields the highest throughput. We train ResNet-32 and LeNet on TensorFlow and perform a batch size exploration. The graphs show the median test accuracy over the last 5 epochs, as in [5], to reduce the variation. Figure 5.1 shows that the best batch size for ResNet-32 to reach a target test accuracy of 80% is 256. In Figure 5.2 shows that by further increasing the batch size value to 1024, we observe a benefit in throughput (10% more, from around 3,000 images per second to 3,300 images per second), but the latter batch size never reaches an accuracy of 80% for the model in the first 80 epochs.

Model	Dataset	Dataset size (MB)	Model size (MB)	# operators
Log. Regression	MNIST	179.45	0.0314	65
LeNet	MNIST	179.45	4.24	169
ResNet-32	CIFAR-10	703.12	1.79	2748

Table 5.1: Model and dataset characteristics used for evaluation.

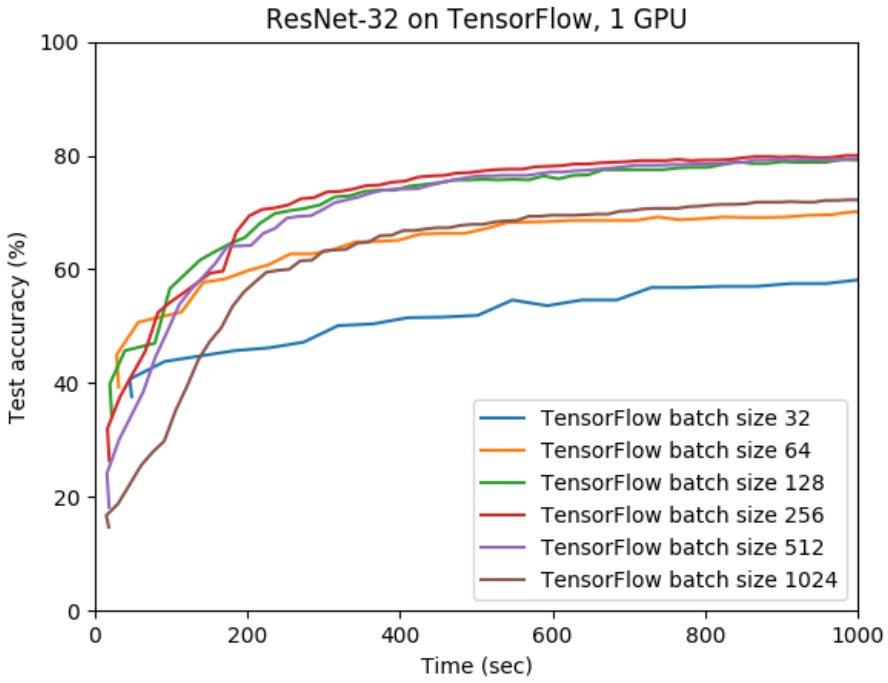


Figure 5.1: Batch size exploration when training ResNet-32 on 1 GPU with TensorFlow. Batch sizes between 128 and 512 take approximately the same time to reach test accuracy 80% despite the fact that training with bigger batch sizes results in higher throughput. By increasing the batch size to 1,024, the model does not even converge to 80%.

Similarly, for LeNet, the best batch size for TensorFlow is 16 to reach a target test accuracy of 99%. For a batch size of 8, training reaches 99% accuracy slower: in 934 seconds compared to 536 seconds for a batch size of 16. For larger batch sizes, the model does not reach the target accuracy after 100 epochs. Note that a smaller batch size has better statistical efficiency (i.e. epochs to accuracy) but worse hardware efficiency (i.e. throughput). Our aim is to improve both numbers.

We compare these findings with the best results obtained with TensorBow running SMA. Figure 5.4 shows the time to accuracy for TensorBow compared to the best result for TensorFlow. With 1 replica, for the same batch size 16, the time to accuracy is almost identical. When training multiple replicas with a batch size of 4, SMA converges in just 382 seconds, decreasing the time to accuracy by approximately 30%. The line for TensorBow with 1 replica and a batch size of 4 shows how slow a single model converges for this batch size.

Figure 5.5 displays the statistical efficiency of the training process. *We observe that a smaller batch size gives a much better statistical efficiency in terms of epochs to target accuracy, at the expense of a longer training time.* Training with multiple replicas further improves statistical efficiency. This is because multiple model replicas can explore a larger portion of the space in parallel while the average model can reduce the variance among them, thus requiring fewer epochs to find good minima.

With this example, we show that by training multiple model replicas we get a benefit in hardware efficiency, increasing the performance of a very small batch size even further, without

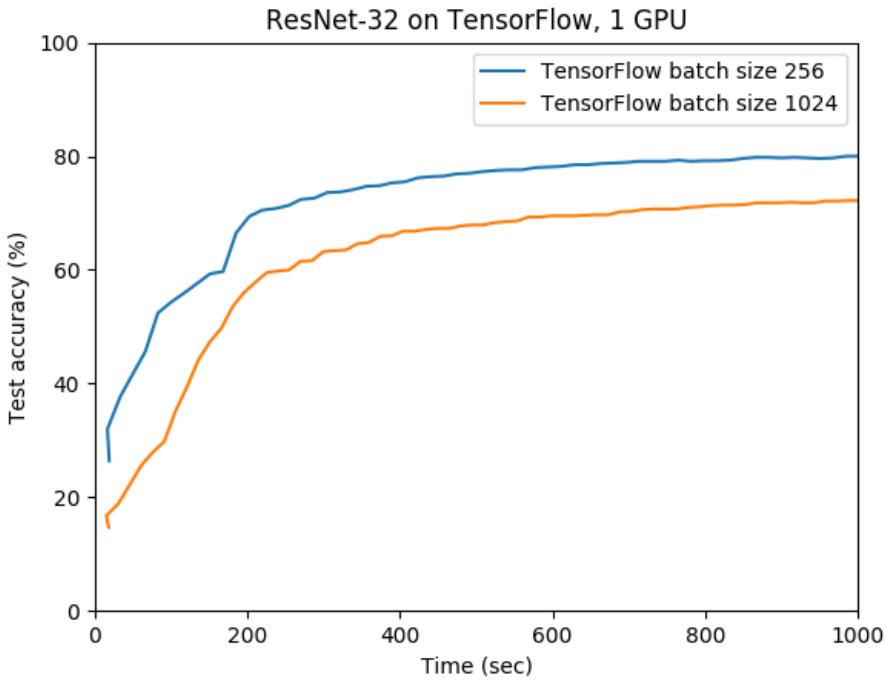


Figure 5.2: Test accuracy over time when training ResNet-32 on 1 GPU with TensorFlow. Up to a batch size of 256, the statistical efficiency improves. Any higher value (e.g. 1,024) hinders the convergence of the model.

hindering its statistical efficiency. This experiment shows that TensorBow can improve both the statistical efficiency and the hardware efficiency for LeNet, resulting in better overall time-to-accuracy.

5.3 The effect of training multiple replicas per GPU

In this section, we evaluate the scalability of the system as the number of replicas trained per GPU increases. We show the hardware efficiency results for logistic regression, LeNet and ResNet-32 when training multiple models in parallel. For each model, we execution the dataflow graph (i.e. computes the gradients and updates the model variables) but we do not perform any synchronisation.

Figure 5.6 compares TensorFlow and TensorBow training a logistic regression model with the MNIST dataset on 1 GPU with batch size 16. We measure the throughput as the number of stream groups (and, consequently, the number of available computational GPU streams) increases. TensorFlow trains a single model on the device irrespective of the number of streams available. Even for such a small model, the throughput decreases slightly with the number of streams due to its stream placement strategy, which induces additional dependencies between datafloe nodes. For example, constant operators such as the learning rate are placed on a new stream, even though they are required on every iteration. Overall, this default placement strategy is not scalable and does not fully take advantage of the available GPU streams.

On the other hand, TensorBow will train a model replica on each of the available stream groups. Despite having to do some extra computations (by executing each operator multiple times, once per replica) this strategy yields much better hardware efficiency. This is the result of using the Crossbow task scheduler which is very efficient for small tasks.

Next, we show the results for the convolutional models, ResNet-32 and LeNet. In Figures 5.7 and 5.8, we use box plots to summarise results from 5 different runs. In the plots, the edges represent the lower and upper quartile values of the throughput, the red line shows the median

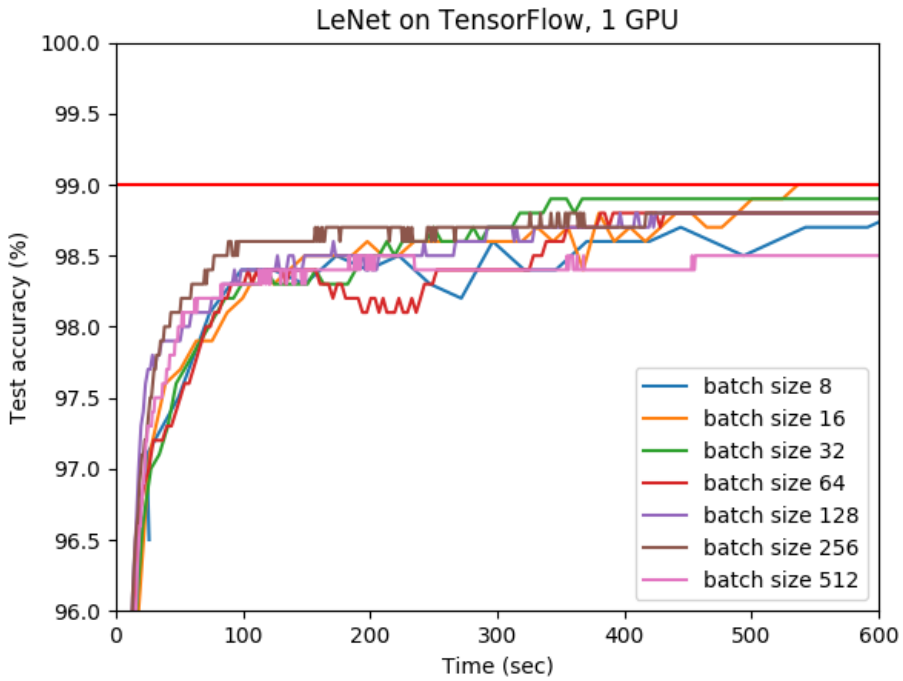


Figure 5.3: Batch size exploration when training LeNet on 1 GPU with TensorFlow. Only batch sizes 8 and 16 converge to the 99% test accuracy target within 100 epochs of training. Batch size 16 converges the fastest.

and the whiskers extend to $\pm 1.5\times$ the interquartile range.

Figure 5.7 shows the scalability of training ResNet-32 on TensorBow. In this experiment we vary the number of replicas trained on 1 GPU and we monitor the throughput. Using 2 replicas on 1 GPU gives an improvement of more than 43% in the average throughput. Going from 2 to 3 replicas further improves throughput by 10%, but the throughput drops slightly in the case of 4 replicas. Being a compute-intensive model, ResNet-32 training causes *stragglers* to appear. The throughput starts to saturate because having many replicas introduces stragglers since we need to wait for all parallel tasks to complete before returning the result of the iteration to the user. Nonetheless, this experiment shows that TensorBow can improve the training process on 1 GPU for a compute-intensive model. To achieve the best performance for this model and any larger networks, it is important to identify the right number of model replicas.

A similar behaviour can be observed in Figure 5.8 when training LeNet. We focus only on the 1 GPU case because the size of the model is small and the iteration time is short even for 1 replica (less than 2ms per task). For 1 model replica, the throughput is already higher than TensorFlow (13,000 images per second compared to 8,700 for TensorFlow). This is caused by the faster scheduling of Crossbow, which becomes critical for small-sized tasks like the ones created for LeNet. We can again observe that adding more model replicas improves the training throughput. Going from 1 to 2 replicas gives an increase of more than 51% in the average throughput. Increasing the number of replicas further, we can observe that the benefit is always around 10% when looking at the mean. For this model, we can scale up to 8 model replicas without affecting the performance.

In summary, this set of experiments shows that we can significantly improve the throughput for all the tested networks on 1 GPU.

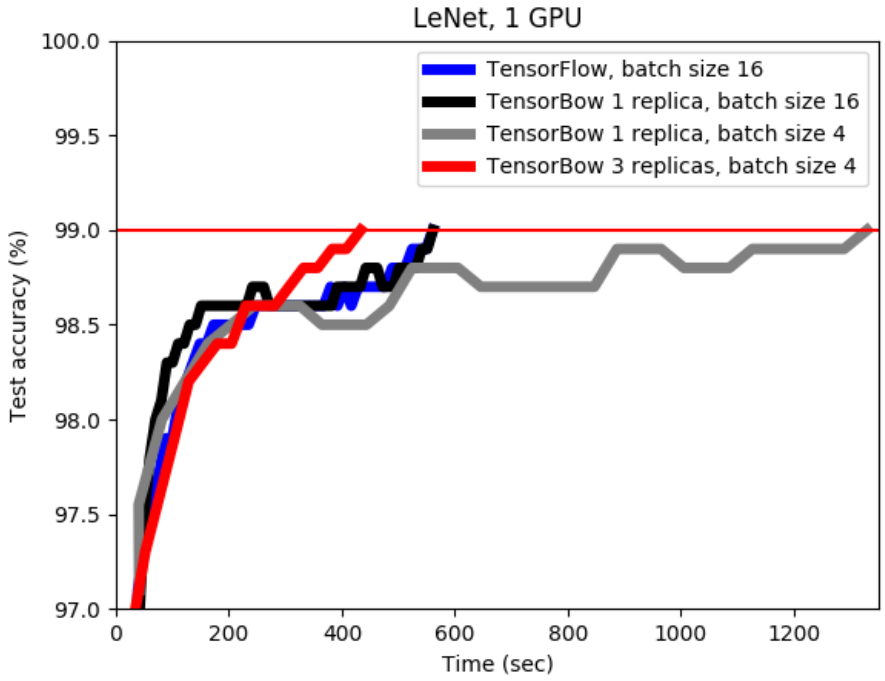


Figure 5.4: Time to reach test accuracy 99% when training LeNet on 1 GPU. TensorBow outperforms the best result for TensorFlow when training 3 replicas with a batch size 4 because it improves both hardware efficiency and statistical efficiency.

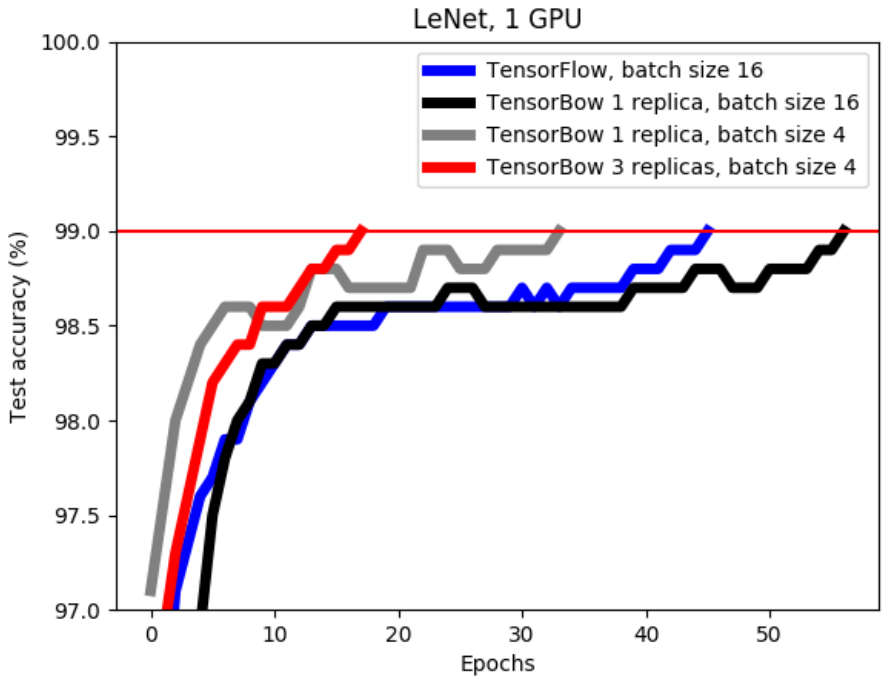


Figure 5.5: Statistical efficiency, measured as the number of epochs to reach test accuracy 99%, when training LeNet on 1 GPU. A batch size of 4 has better statistical efficiency than bigger batch sizes. TensorBow further improves it by training multiple model replicas and synchronising them with SMA.

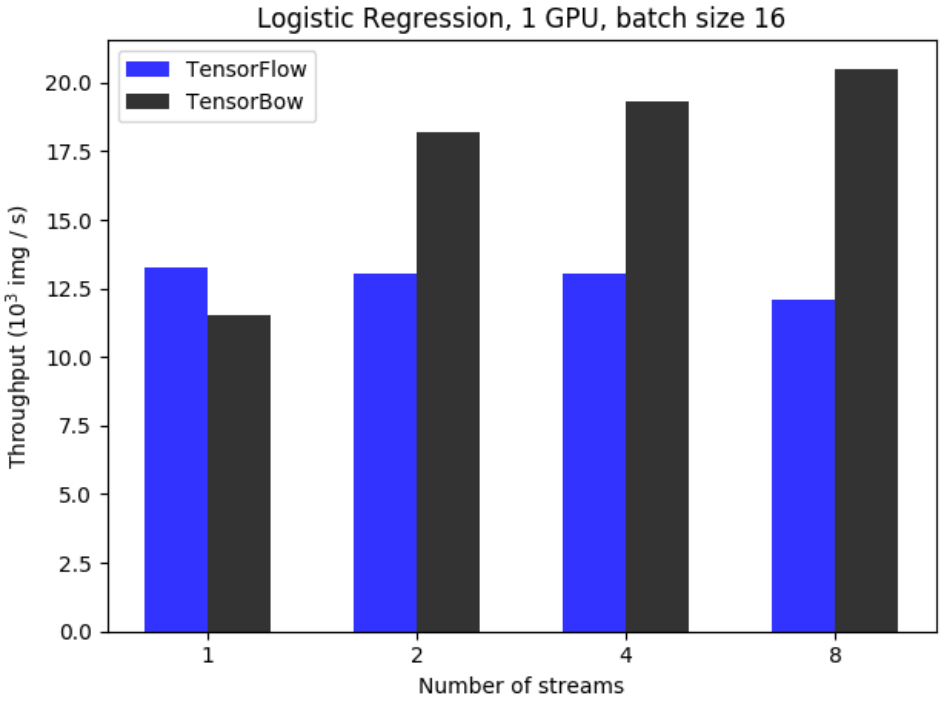


Figure 5.6: Comparing TensorFlow and TensorBow hardware efficiency when training logistic regression on 1 GPU with batch size 16 as we vary the number of stream groups per GPU. TensorFlow cannot take advantage of the multiple streams, while TensorBow can.

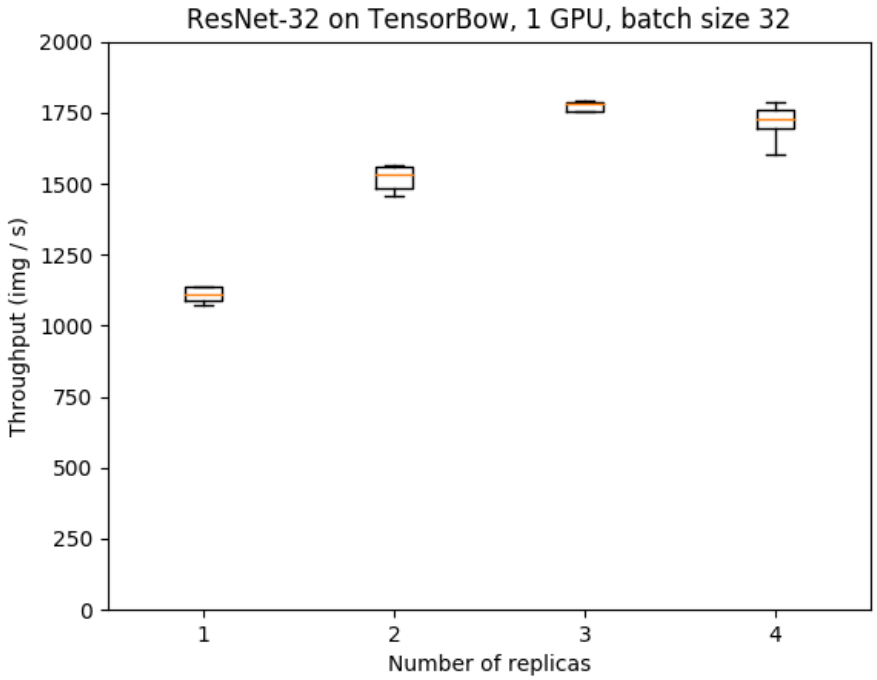


Figure 5.7: TensorBow's hardware efficiency when training ResNet-32 on 1 GPU with batch size 32. Beyond 3 model replicas, there is no further improvement in training throughput.

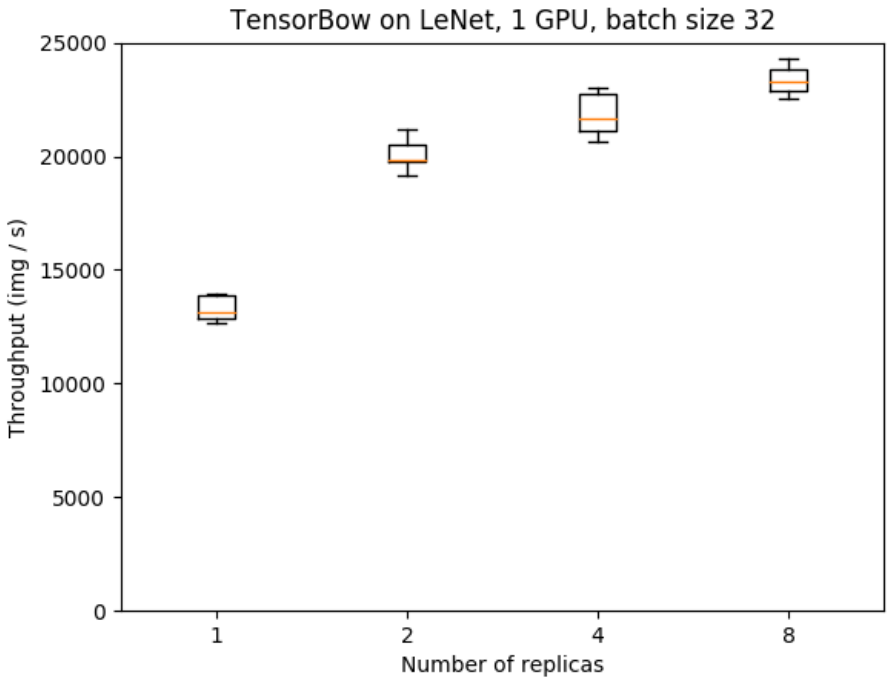


Figure 5.8: TensorBow's hardware efficiency when training LeNet on 1 GPU with batch size 32. The system manages to increase the throughput with multiple model replicas because of the small network size and very fast scheduling.

5.4 Scaling to multiple GPUs

We established how TensorBow scales the training throughput as the number of streams increases for 1 GPU device. In the next experiment, we evaluate how its scales as the number of GPUs increases. We use ResNet-32 for this experiment, since the other models are too small to scale to multiple GPUs. Figure 5.9 shows the results. We fix the batch size to 32. The blue line represents the throughput obtained by training the model on TensorFlow with 1 GPU at 1,062 images per second. Looking at TensorBow's throughput for 1 model replica (black bars), running on 2 GPUs results in almost a linear increase: from 1,109 to 1,949 images per second. On 2 GPUs with 2 model replicas each, we observe an improvement of 32% in hardware efficiency, just a few percentages below the improvement we observe for the 1 GPU case (39%).

On 4 GPUs and a single model replica on each the training throughput is identical to having 2 GPUs with 2 model replicas. The benefit is sublinear when comparing with the 1 GPU case because of the scheduling overhead caused by working with small tasks. We can further improve these results by setting the thread affinity, splitting execution threads across CPU sockets. Training 2 model replicas on each of the 4 GPUs improves the performance by 35%. Therefore, we can conclude that the relative increase in throughput remains constant as we scale out from 1 to 4 GPUs.

5.5 The effect of batch size

In the previous experiments, we analysed hardware efficiency when training with a small, fixed batch size. We now study the effect of increasing the batch size. Figure 5.10 shows how TensorBow and TensorFlow perform when training ResNet-32 on 1 GPU for various batch sizes. We consider TensorBow with 1 and 2 model replicas.

For a small batch size, below 128, the performance of TensorBow with 1 replica is on par with the numbers for TensorFlow. This shows that the overall system overhead (e.g. the one introduced by the additional operators due to model averaging) is negligible compared to the

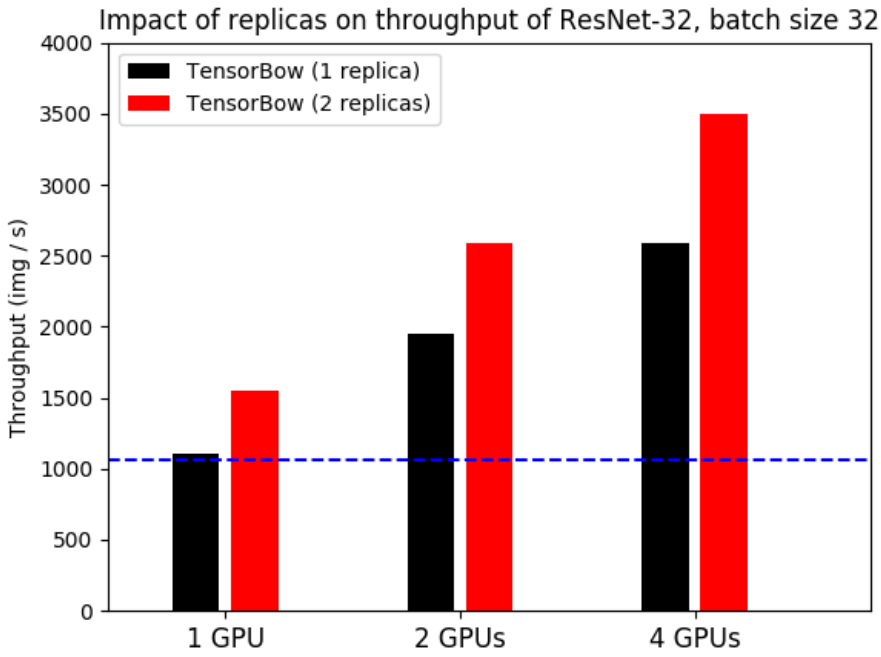


Figure 5.9: Scaling TensorBow to multiple GPU devices. The blue line represents the performance of TensorFlow on 1 GPU. Irrespective of the number of devices, we observe an increase in throughput when training a second model replica per GPU.

task size. In all those cases, by adding a second model replica on the same device we increase the throughput by more than 40%.

Moving to a batch size of 128 or higher, we still see an improvement in hardware efficiency by adding a second model replica, but the improvement is just over 22% for a batch size of 128 and 256. The reason for this behaviour is that a bigger batch size translates to more compute-intensive kernels. There is less room for concurrency as fewer GPU resources are left unused by one operation.

5.6 Synchronisation Overhead

In this section we evaluate the intra- and inter-GPU synchronisation overhead of the SMA algorithm to understand the effectiveness of our incremental synchronisation implementation and how much we deviate from the ideal performance.

Intra-GPU synchronisation. In Figure 5.11, we show the throughput when training LeNet as we vary the number of model replicas on 1 GPU. We compare the case where no synchronisation between models occurs with the case of intra-GPU aggregation. The throughput drops only slightly. The biggest decrease is for the case of 2 model replicas, from 19,175 images per second to 15,405 images per second, which represents a decrease of less than 20%. All the other cases have show a better result, so overall our implementation is reasonably efficient.

The same experiment for ResNet-32 gives a better result because of the increased depth of the model. In Figure 5.12, the blue line shows the baseline for running the model on TensorFlow. Having much more operations reduces the impact of intra-GPU synchronisation. For 2 model replicas, the throughput drops from 1,458 images per second to 1,435 images per second, a decrease of less than 1.6%. For 3 model replicas, the throughput is 1,585 images per second compared to 1,753 images per second with no synchronisation, which represents a less than 10% decrease. Overall, the results for ResNet-32 are much better than the ones for LeNet and both

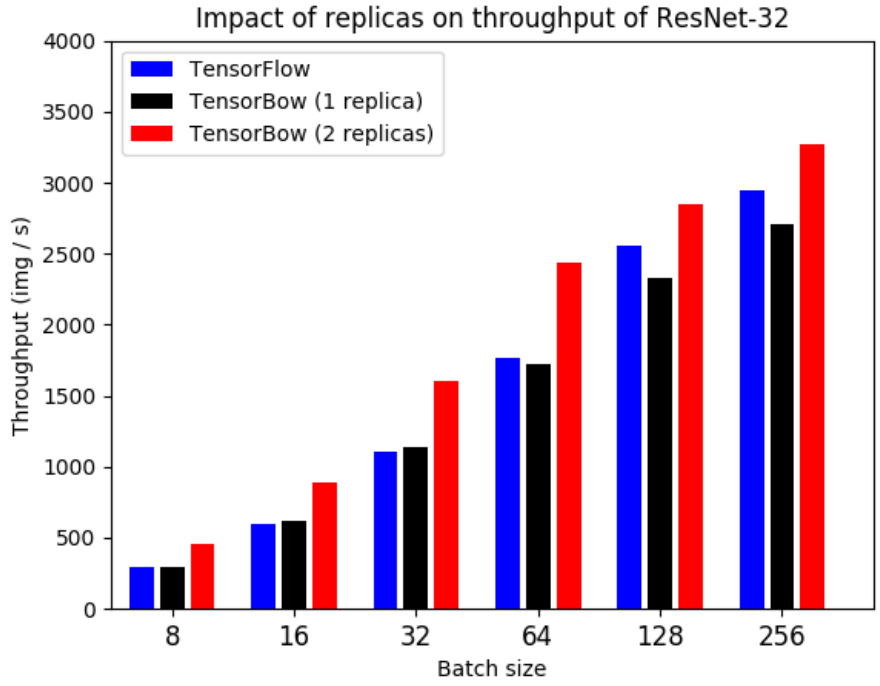


Figure 5.10: Training throughput when training ResNet-32 on 1 GPU with varying batch sizes. Small-batch training takes advantage of the resources available, and increases the throughput by training a second model replica on the GPU. For larger batch sizes, there is less room for improvement because each operator works on more data, therefore becomes more compute-intensive.

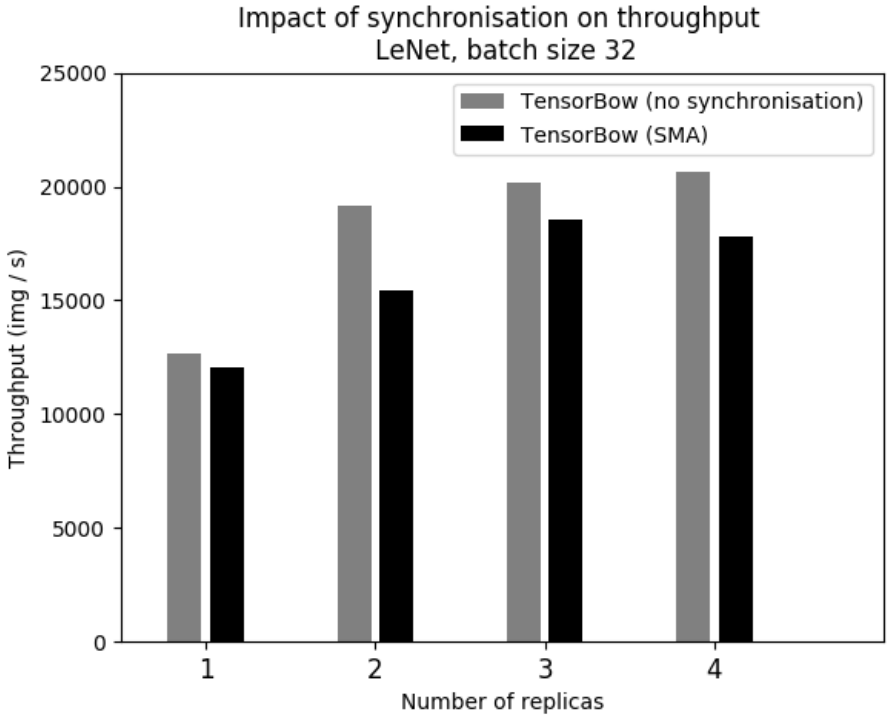


Figure 5.11: Intra-GPU synchronisation overhead when training LeNet on 1 GPU. The overall cost is at most 20% because the model is small and the task duration is short.

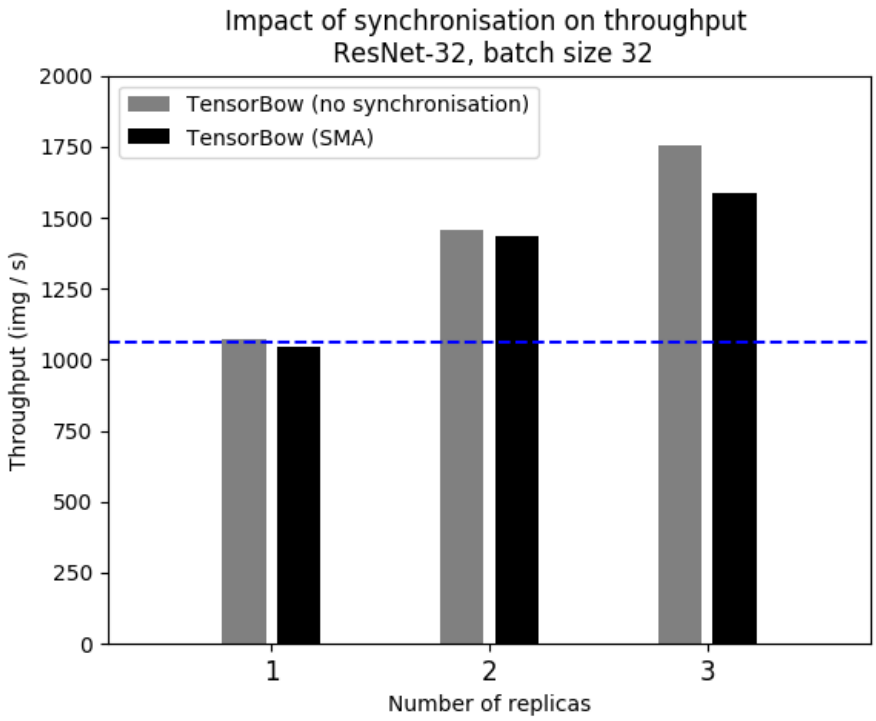


Figure 5.12: Intra-GPU synchronisation overhead when training ResNet-32 on 1 GPU. The overall performance penalty is small because the model is very deep.

experiments show a manageable overhead for intra-GPU synchronisation.

Inter-GPU synchronisation. To measure the inter-GPU synchronisation, we compare the performance of TensorBow on 2 GPUs with 1 and 2 model replicas with the ideal baseline for training independent models (Figure 5.13). We observe that the overall performance decreases by about 50%, but the relative increase in throughput by adding a second replica on each device remains constant. The inter-GPU synchronisation overhead is unavoidable because of the NCCL operators. We measured the performance of TensorFlow when training ResNet-32 with the same batch size on multiple devices as well. We observed a decrease in throughput of more than 30% with NCCL.

Overall, given that the relative performance increase stays almost constant in all scenarios, the synchronisation implementation does not induce unnecessary bottlenecks. We can further improve the numbers of our TensorBow implementation by grouping variables together to minimise the number of NCCL AllReduce calls issued or by moving the entire synchronisation step at the end of a learning task.

5.7 Summary

The experimental results presented in this chapter validate our design decisions and implementation choices and show that GPU resources can be better utilised if more than one model replica is trained concurrently. TensorBow performs on par with TensorFlow when training 1 model replica per GPU for large neural network models and better for smaller models. Irrespective of the network size, we show an improvement in the hardware efficiency by at least 43% when collocating a second model replica on the same device. The intra-GPU synchronisation overhead is also negligible for compute-intensive models. The inter-GPU synchronisation induces an additional overhead which is not a bottleneck of the algorithm used. We can attempt additional techniques such as variable grouping to close the performance gap when communicating across devices.

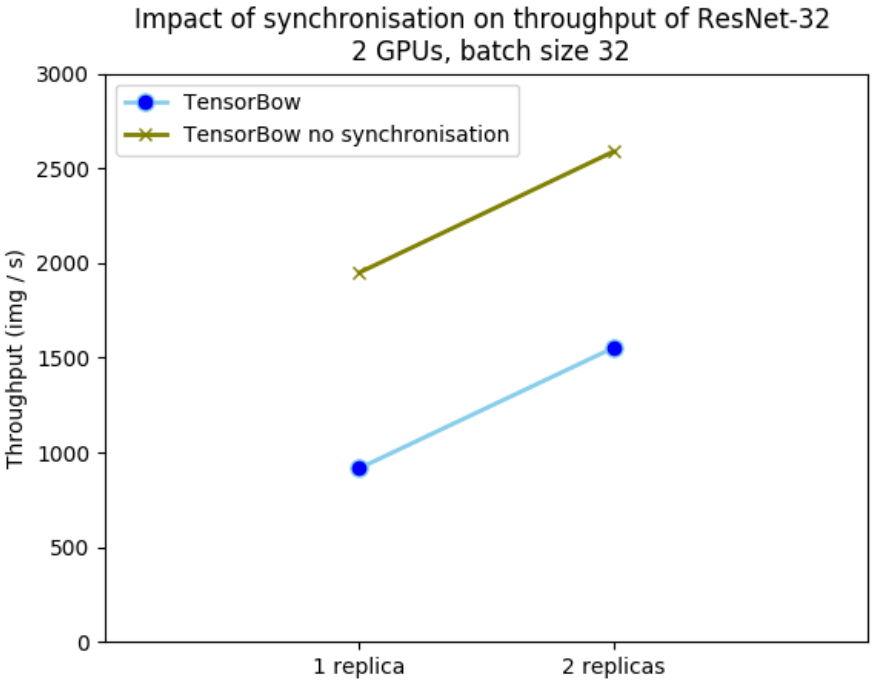


Figure 5.13: Inter-GPU synchronisation overhead in TensorBow. The relative gain by training a second model replica per GPU remains constant.

While these results illustrate that increasing the number of model replicas per GPU can be beneficial to reduce the training time, they also show that identifying the correct number of replicas becomes crucial to reach the best performance.

Chapter 6

Conclusions & Future Work

This project presented some of the current problems faced by deep learning systems. Researchers are still building systems where they choose a batch size that helps improve resource utilisation and scale out in terms of throughput, without considering the effect it has on convergence. Any loss in precision must be compensated by the algorithms used.

With TensorBow, we add better support for small batch training in TensorFlow. We make the necessary changes to support this safely within the TensorFlow infrastructure. Overall, we show a possible design for how to build a system where, irrespective of the batch size, we aim for a constant hardware utilisation that is as high as possible. By training multiple model replicas per device, we take advantage of the free resources and we can aim to improve both hardware efficiency and statistical efficiency.

We advocate that any future machine learning system should follow a model-oriented data-parallel design since it helps implementing and testing new training and synchronisation algorithms and does not restrict any of the existing ones. We now describe how we can extend our prototype to support other components of the TensorFlow ecosystem.

XLA. Accelerated Linear Algebra (XLA) [57] is a compiler capable of optimising computations specified in TensorFlow. It takes as input a dataflow graph, specified using the TensorFlow API, and replaces all the operations with a sequence of auto-generated operators. The new computation graph is described using a protocol buffer and passed to a TensorFlow `Session` for execution. This protocol buffer does not differ much from the standard one. Any actual modifications happen within the `Session` class during the optimisation phase. We execute this phase during our `TensorBowSession` initialisation as well. But we need to study if all of the XLA operations are capable of running on TensorBow. It is very likely that the XLA target-independent optimisations can be incorporated. However, the XLA can perform specific optimisations to partition a sub-graph across streams. Such steps would require special considerations in order to retain the ability to train multiple replicas per GPU.

Distributed TensorBow. We will further study if improving single-node performance of the training system benefits distributed training. This is an interesting question because the overall scalability of distributed training may be bounded by local resources and single machine capabilities in certain scenarios [46]. To address this, the synchronisation mechanism is the main component which needs to be revisited. It remains to be seen how this component can be adapted. The intra-GPU aggregation will continue to happen transparently. For inter-GPU aggregation, the model replicas or gradients need to be exchanged over the network. NCCL already provides support for doing inter-GPU aggregation between different machines, so TensorBow can be further extended to support this.

Support for large models. Many modern neural network architectures have memory requirements which exceed the capabilities of a single GPU. TensorFlow now has support for

performing memory swaps during training via special operators [47]. Those operators can be adapted and tested for multiple model replicas to support training large neural networks.

Appendix A

Project Setup

Crossbow had to be integrated with TensorFlow as a library so that TensorFlow can use Crossbow's model manager and task manager. We achieved this by exposing Crossbow's C API directly to TensorFlow using the `extern "C" {}` directive and bypassing the JNI header files used

Bazel [61], the tool used to compile TensorFlow, is very strict with the location of the files included in the build. We created special BUILD files, so that TensorFlow was aware of the newly introduced dependency, Crossbow. The dependency were introduced in the `third-party/` folder, which contains various other extensions and tool-kits (e.g. the JPEG image library). Additionally, we added an `ld.config` entry to the linker locate the Crossbow shared dynamic library at run-time.

Finally, the easiest way to enable GPU support for TensorFlow is to use Docker, since it reduces the number of steps required for setup. The only requirement is having the NVIDIA GPU drivers installed on the host. Then, any Docker image using `nvidia-docker` as the base image has appropriate GPU support. We developed a new Docker file to create an image that contains all the necessary dependencies for both TensorFlow and Crossbow.

Bibliography

[1] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dan Mane, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viegas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: Large-scale machine learning on heterogeneous distributed systems. Mar 14, 2016. URL `https://arxiv.org/abs/1603.04467`. Accessed 16th June 2019.

[2] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, Manjunath Kudlur, Josh Levenberg, Rajat Monga, Sherry Moore, Derek G. Murray, Benoit Steiner, Paul Tucker, Vijay Vasudevan, Pete Warden, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. Tensorflow: A system for large-scale machine learning. pages 265–283, Berkley, CA, USA, 2016. USENIX Association.

[3] Liu Guangcong. *TensorFlow kernel analysis*. Github. URL `https://github.com/sergey-serebryakov/tensorflow-internals`. Translated by Sergey Serebryakov; Copyright 2017 Liu Guangcong, Accessed 16th June 2019.

[4] Alexandros Koliousis, Pijika Watcharapichat, Matthias Weidlich, Luo Mai, Paolo Costa, and Peter Pietzuch. Crossbow: Scaling deep learning with small batch sizes on multi-gpu servers. Jan 8, 2019. URL `https://arxiv.org/abs/1901.02244`. Accessed 16th June 2019.

[5] Priya Goyal, Piotr Dollár, Ross Girshick, Pieter Noordhuis, Lukasz Wesolowski, Aapo Kyrola, Andrew Tulloch, Yangqing Jia, and Kaiming He. Accurate, large minibatch sgd: Training imagenet in 1 hour. Jun 8, 2017. URL `https://arxiv.org/abs/1706.02677`. Accessed 16th June 2019.

[6] Nitish Shirish Keskar, Dheevatsa Mudigere, Jorge Nocedal, Mikhail Smelyanskiy, and Ping Tak Peter Tang. On large-batch training for deep learning: Generalization gap and sharp minima. Sep 15, 2016. URL `https://arxiv.org/abs/1609.04836`. Accessed 16th June 2019.

[7] Dominic Masters and Carlo Luschi. Revisiting small batch training for deep neural networks. Apr 20, 2018. URL `https://arxiv.org/abs/1804.07612`. Accessed 16th June 2019.

[8] Sam McCandlish, Jared Kaplan, Dario Amodei, and OpenAI Dota Team. An empirical model of large-batch training. Dec 14, 2018. URL `https://arxiv.org/abs/1812.06162`. Accessed 16th June 2019.

[9] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, Nov 1, 1997. doi: 10.1162/neco.1997.9.8.1735. URL `http:`

//www.mitpressjournals.org/doi/abs/10.1162/neco.1997.9.8.1735. Accessed 16th June 2019.

[10] Y-Lan Boureau, Jean Ponce, and Yann LeCun. A theoretical analysis of feature pooling in visual recognition. In *ICML'10*, pages 111–118. Omnipress, 2010.

[11] Christian Szegedy, Wei Liu, Yangqing Jia, Pierre Sermanet, Scott Reed, Dragomir Anguelov, Dumitru Erhan, Vincent Vanhoucke, and Andrew Rabinovich. Going deeper with convolutions. pages 1–9. IEEE, Jun 2015. ISBN 1063-6919. doi: 10.1109/CVPR.2015.7298594. URL https://ieeexplore.ieee.org/document/7298594. Accessed 16th June 2019.

[12] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. pages 770–778. IEEE, Jun 2016. doi: 10.1109/CVPR.2016.90. URL https://ieeexplore.ieee.org/document/7780459. Accessed 16th June 2019.

[13] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. Sep 4, 2014. URL https://arxiv.org/abs/1409.1556. Accessed 16th June 2019.

[14] Elad Hoffer, Itay Hubara, and Daniel Soudry. Train longer, generalize better: closing the generalization gap in large batch training of neural networks. May 24, 2017. URL https://arxiv.org/abs/1705.08741. Accessed 16th June 2019.

[15] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. EuroSys, pages 59–72, New York, NY, USA, 2007 . ACM. doi: 10.1145/1272996.1273005. URL http://dl.acm.org/citation.cfm?id=1273005. Accessed 16th June 2019.

[16] Ion Stoica, Dawn Song, Raluca Ada Popa, David Patterson, Michael W. Mahoney, Randy Katz, Anthony D. Joseph, Michael Jordan, Joseph M. Hellerstein, Joseph E. Gonzalez, Ken Goldberg, Ali Ghodsi, David Culler, and Pieter Abbeel. A berkeley view of systems challenges for ai. Dec 15, 2017. URL https://arxiv.org/abs/1712.05855. Accessed 16th June 2019.

[17] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep learning*. The MIT Press, London, England, 2016. ISBN 9780262035613. URL http://www.deeplearningbook.org. Accessed 16th June 2019.

[18] Jeff Dean, David Patterson, and Cliff Young. A new golden age in computer architecture: Empowering the machine-learning revolution. *IEEE Micro*, 38(2):21–29, Mar 2018. doi: 10.1109/MM.2018.112130030. URL https://ieeexplore.ieee.org/document/8259424. Accessed 16th June 2019.

[19] Tianqi Chen, Mu Li, Yutian Li, Min Lin, Naiyan Wang, Minjie Wang, Tianjun Xiao, Bing Xu, Chiyuan Zhang, and Zheng Zhang. Mxnet: A flexible and efficient machine learning library for heterogeneous distributed systems. Dec 3, 2015. URL https://arxiv.org/abs/1512.01274. Accessed 16th June 2019.

[20] Melvin Johnson, Mike Schuster, Quoc V. Le, Maxim Krikun, Yonghui Wu, Zhifeng Chen, Nikhil Thorat, Fernanda Viégas, Martin Wattenberg, Greg Corrado, Macduff Hughes, and Jeffrey Dean. Google's multilingual neural machine translation system: Enabling zero-shot translation. *Transactions of the Association for Computational Linguistics*, 5:339–351, 2017.

[21] G. Hinton, Li Deng, Dong Yu, G. E. Dahl, A. Mohamed, N. Jaitly, Andrew Senior, V. Vanhoucke, P. Nguyen, T. N. Sainath, and B. Kingsbury. Deep neural networks for acoustic

modeling in speech recognition: The shared views of four research groups. *IEEE Signal Processing Magazine*, 29(6):82–97, Nov 2012. doi: 10.1109/MSP.2012.2205597. URL https://ieeexplore.ieee.org/document/6296526. Accessed 16th June 2019.

[22] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. Imagenet classification with deep convolutional neural networks. pages 1097–1105, USA, 2012 . Curran Associates Inc.

[23] Matei Zaharia, Mosharaf Chowdhury, Tathagata Das, Ankur Dave, Justin Ma, Murphy McCauley, Michael J. Franklin, Scott Shenker, and Ion Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *NSDI*, 2012 .

[24] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. Nov 8, 2018. URL https://arxiv.org/abs/1811.03600. Accessed 16th June 2019.

[25] Alex Krizhevsky. One weird trick for parallelizing convolutional neural networks. Apr 23, 2014. URL https://arxiv.org/abs/1404.5997. Accessed 16th June 2019.

[26] Jian Zhang and Ioannis Mitliagkas. Yellowfin and the art of momentum tuning. Jun 12, 2017. URL https://arxiv.org/abs/1706.03471. Accessed 16th June 2019.

[27] Sorathan Chaturapruek, John C. Duchi, and Christopher Ré. Asynchronous stochastic convex optimization: the noise is in the noise and sgd don't care. In *NIPS*, page 1531–1539. Curran Associates, Inc., 2015 .

[28] D. E. Rumelhart, G. E. Hinton, and R. J. Williams. *Learning internal representations by error propagation*, pages 318–362. Parallel distributed processing: explorations in the microstructure of cognition. MIT Press, Cambridge, MA, USA, 1986.

[29] Christopher J. Shallue, Jaehoon Lee, Joseph Antognini, Jascha Sohl-Dickstein, Roy Frostig, and George E. Dahl. Measuring the effects of data parallelism on neural network training. Nov 8, 2018. URL https://arxiv.org/abs/1811.03600. Accessed 16th June 2019.

[30] David A. Patterson and John L. Hennessy. *Computer organization and design*. Morgan Kaufman, Amsterdam, 5th edition, 2014. ISBN 9780124077263.

[31] Lam M. Nguyen, Nam H. Nguyen, Dzung T. Phan, Jayant R. Kalagnanam, and Katya Scheinberg. When does stochastic gradient algorithm work well? Jan 18, 2018. URL https://arxiv.org/abs/1801.06159. Accessed 16th June 2019.

[32] Hongyu Zhu, Mohamed Akrout, Bojian Zheng, Andrew Pelegris, Amar Phanishayee, Bianca Schroeder, and Gennady Pekhimenko. Tbd: Benchmarking and analyzing deep neural network training. Mar 16, 2018. URL https://arxiv.org/abs/1803.06905. Accessed 16th June 2019.

[33] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. 10 June 2014. URL https://arxiv.org/pdf/1406.2661.pdf. Accessed 16th June 2019.

[34] Lars Mescheder, Andreas Geiger, and Sebastian Nowozin. Which training methods for gans do actually converge? Jan 13, 2018. URL https://arxiv.org/abs/1801.04406. Accessed 16th June 2019.

[35] Léon Bottou, Frank E. Curtis, and Jorge Nocedal. Optimization methods for large-scale machine learning. *SIAM Review*, 60(2):223–311, Jan 2018. doi: 10.1137/16M1080173. URL https://arxiv.org/pdf/1606.04838.pdf. Accessed 16th June 2019.

[36] D. Randall Wilson and Tony R. Martinez. The general inefficiency of batch training for gradient descent learning. *Neural Networks*, 16(10):1429–1451, 2003. doi: 10.1016/S0893-6080(03)00138-2. URL https://www.sciencedirect.com/science/article/pii/S0893608003001382. Accessed 16th June 2019.

[37] Sixin Zhang, Anna Choromanska, and Yann LeCun. Deep learning with elastic averaging sgd. Dec 20, 2014. URL https://arxiv.org/abs/1412.6651. Accessed 16th June 2019.

[38] B. T. Polyak and A. B. Juditsky. Acceleration of stochastic approximation by averaging. *SIAM Journal on Control and Optimization*, 30(4):838–855, Jul 1, 1992. doi: 10.1137/0330046. URL https://search.proquest.com/docview/925919659. Accessed 16th June 2019.

[39] Y. Lecun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, Nov 1998. doi: 10.1109/5.726791. URL https://ieeexplore.ieee.org/document/726791. Accessed 16th June 2019.

[40] Bhargav Kanagal and Sandeep Tata. Recommendations for all: Solving thousands of recommendation problems daily. pages 1404–1413. IEEE, Apr 2018. doi: 10.1109/ICDE.2018.00159. URL https://ieeexplore.ieee.org/document/8509380. Accessed 16th June 2019.

[41] Dominique T. Shipmon, Jason M. Gurevitch, Paolo M. Piselli, and Stephen T. Edwards. Time series anomaly detection; detection of anomalous drops with limited features and sparse examples in noisy highly periodic data. Aug 11, 2017. URL https://arxiv.org/abs/1708.03665. Accessed 16th June 2019.

[42] Luo Mai, Vamsi Kuppa, Sudheer Dhulipalla, Sriram Rao, Kai Zeng, Rahul Potharaju, Le Xu, Steve Suh, Shivaram Venkataraman, Paolo Costa, Terry Kim, and Saravanan Muthukrishnan. Chi: A scalable and programmable control plane for distributed stream processing systems. *Proceedings of the VLDB Endowment*, 11(10):1303–1316, Jun 1, 2018. doi: 10.14778/3231751.3231765. URL https://www.microsoft.com/en-us/research/uploads/prod/2018/11/mai18chi.pdf. Accessed 16th June 2019.

[43] A. Halevy, P. Norvig, and F. Pereira. The unreasonable effectiveness of data. *IEEE Intelligent Systems*, 24(2):8–12, Mar 2009. doi: 10.1109/MIS.2009.36. URL https://ieeexplore.ieee.org/document/4804817. Accessed 16th June 2019.

[44] Lior Abraham, Subbu Subramanian, Janet L. Wiener, Okay Zed, John Allen, Oleksandr Barykin, Vinayak Borkar, Bhuwan Chopra, Ciprian Gerea, Daniel Merl, Josh Metzler, and David Reiss. Scuba. *Proceedings of the VLDB Endowment*, 6(11):1057–1067, Aug 27, 2013. doi: 10.14778/2536222.2536231. URL https://research.fb.com/wp-content/uploads/2016/11/scuba-diving-into-data-at-facebook.pdf. Accessed 16th June 2019.

[45] Yangqing Jia, Evan Shelhamer, Jeff Donahue, Sergey Karayev, Jonathan Long, Ross Girshick, Sergio Guadarrama, and Trevor Darrell. Caffe. MM '14, pages 675–678. ACM, Nov 3, 2014. URL http://dl.acm.org/citation.cfm?id=2654889. Accessed 16th June 2019.

[46] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. Making sense of performance in data analytics frameworks. In *Proceedings of the 12th USENIX Conference on Networked Systems Design and Implementation*, NSDI'15, pages 293–307, Berkeley, CA, USA, 2015. USENIX Association. ISBN 978-1-931971-218. URL http://dl.acm.org/citation.cfm?id=2789770.2789791. Accessed 16th June 2019.

[47] Tung D. Le, Haruki Imai, Yasushi Negishi, and Kiyokuni Kawachiya. Tflms: Large model support in tensorflow by graph rewriting. Jul 5, 2018. URL https://arxiv.org/abs/1807.02037. Accessed 16th June 2019.

[48] François Chollet. Keras, 2015. URL https://github.com/fchollet/keras. Accessed 16th June 2019.

[49] Memory management for tensorflow. URL https://github.com/miglopst/cs263_spring2018/wiki/Memory-management-for-tensorflow. Accessed 16th June 2019.

[50] Google. Protocol buffers, 2019. URL https://developers.google.com/protocol-buffers/. Accessed 16th June 2019.

[51] Open compute project, 2011. URL https://www.opencompute.org/about. Accessed 16th June 2019.

[52] David M. Beazley. Swig : An easy to use tool for integrating scripting languages with c and c++. Monterey, CA, 1996.

[53] Seiya Tokui, Kenta Oono, Shohei Hido, and Justin Clayton. Chainer: a next-generation open source framework for deep learning. *NIPS Workshop on Machine Learning Systems (LearningSys)*, 2015.

[54] Xianyan Jia, Shutao Song, Wei He, Yangzihao Wang, Haidong Rong, Feihu Zhou, Liqiang Xie, Zhenyu Guo, Yuanzhou Yang, Liwei Yu, Tiegang Chen, Guangxiao Hu, Shaohuai Shi, and Xiaowen Chu. Highly scalable deep learning training system with mixed-precision: Training imagenet in four minutes. Jul 30, 2018. URL https://arxiv.org/abs/1807.11205. Accessed 16th June 2019.

[55] Marco Pennacchiotti and Ana-Maria Popescu. A machine learning approach to twitter user classification. 2011.

[56] PyTorch. Pytorch. URL https://pytorch.org/. Accessed 16th June 2019.

[57] Google. Xla overview. URL https://www.tensorflow.org/xla/overview. Accessed 16th June 2019.

[58] Stephen Jones. Memory management tips, tricks & techniques, 2015. URL http://on-demand.gputechconf.com/gtc/2015/presentation/S5530-Stephen-Jones.pdf. SpaceX, GTC, Accessed 16th June 2019.

[59] NVIDIA Corporation. cuDNN Developer Guide, 2019. URL https://docs.nvidia.com/deeplearning/sdk/cudnn-developer-guide/index.html. Accessed 16th June 2019.

[60] NVIDIA Corporation. NVIDIA Collective Communications Library (NCCL), 2019. URL https://docs.nvidia.com/deeplearning/sdk/nccl-developer-guide/docs/index.html. Accessed 16th June 2019.

[61] Bazel. Bazel. URL https://bazel.build/. Accessed 16th June 2019.