Technical Report

UCAM-CL-TR-956
ISSN 1476-2986

Number 956

UNIVERSITY OF CAMBRIDGE

Computer Laboratory

Optimisation of a modern numerical library: a bottom-up approach

Jianxin Zhao

April 2021

15 JJ Thomson Avenue
Cambridge CB3 0FD
United Kingdom
phone +44 1223 763500

https://www.cl.cam.ac.uk/

This technical report is based on a dissertation submitted September 2019 by the author for the degree of Doctor of Philosophy to the University of Cambridge, Corpus Christi College.

Technical reports published by the University of Cambridge Computer Laboratory are freely available via the Internet:

https://www.cl.cam.ac.uk/techreports/

ISSN 1476-2986

Abstract

Numerical libraries lie in the heart of modern applications, from machine learning, scientific computation and to Internet of Things (IoT). It has dominated many aspects of our daily lives. Numerical library used to lie in the low level of applications, and only need to focus on provide fast calculation. However, with social awareness of privacy and personal data arising, computation is gradually moved to devices in heterogeneous environment. Recently development of edge devices such as Edge TPU also promotes a trend of decentralised computation. Given this trend, a new understanding of the full stack of computation is required to optimise computation at various levels.

In this thesis, based on my experience participating in the development of a numerical library, I present a bottom-up approach that centres on numerical library to describe the optimisation of computation at various levels. I present the low-level design of numerical operations and show the related impact on performance optimisation. I create new algorithms for key operations, and build an automatic tuning module to further improve performance. At the graph level, which consists of multiple operations, I present the idea of using graph as common Intermediate Representation to enable interoperability on other computation frameworks and devices. To demonstrate this approach, I build the TFgraph system that provides a symbolic representation layer to exchange the computation between Owl and other frameworks. At a higher level, the computation graph can be seen as a unit, and at this level I identify the problems of computation composition and deployment. I build the Zoo system to address these two problems. It provides a small Domain-specific Language to enable composition of advanced data analytics services. Benefiting from OCaml's powerful type system, the Zoo provides type checking for the composition. Besides, the Zoo DSL supports fine-grained version control in composing. It also involves deploying composed services to multiple backends. Finally, the top level involves collaboration of multiple deployed computations. At this level, I focus on the barrier control methods, propose two quantitative metrics to evaluate existing barrier control methods, and bring new insights into their design. I have also built a simulation platform and real-world experiments to perform thorough evaluation of the PSP compared to existing barrier methods.

Acknowledgements

First, I would like to express my gratitude to my supervisor, Prof. Jon Crowcroft, for giving me the opportunity to undertake this PhD, allowing me to explore freely during my PhD, and showing the utmost generosity in helping me with thesis writeup and submission. My deepest thanks go to Dr. Richard Mortier, my advisor. His support and guide on my work is the best I can expect from an advisor, even though I didn't yield the expected output as return. This thesis is also dedicated to Prof. Chi Harold Liu. Without his continuous support to my career, I would never be able to start my PhD in Cambridge in the first place.

My special thanks go to Dr. Liang Wang. His extraordinary effort in guiding me through my whole PhD career is much more than I deserve. If he would use that effort somewhere else, I have no doubt that he can earn a second PhD degree; not just in computer science, but in any field of his choice. In collaborating with him on the Owl project, I have learnt a lot about both research and life. Without his unyielding help and guide, it would be impossible for me to graduate.

I would like to thank the OCaml Labs for being such a great group which allows me to join the weekly discussion, supports me with experiment device, and covers conference trip expense. A weirdo like me even begins to learn to socialise here. So thanks to all the lab members.

No doubt that my work benefits a lot from my Corpus Christi College. I have no idea how Tessa can tolerate my annoying emails, and yet she always help me as much as possible. A pity that I don't know half of the people there half as well as I should like. I'll never forget Leckhampton: the kitchen, the library, the sports field, and the beautiful garden full of sunshine.

To the Gcores pals: Ryoma, 42, Nadya, Simon, Xizongbu, Old White, NJBK, Director, Moby, Headless, Bus Driver, SunSky... this list can go on and on. I just feel so glad and fortunate to know you guys. When I said that Gcores has saved my life, I was only half joking. You have indeed lifted me up from the abyss. To Giorno, Zoro and Sakuragi: your golden spirits have shined upon the path for me, and I'm still trying my best to follow it. To the Fire Keeper: even when the fire is fading, I can always find peace at thy side. Thank you all so much for being with me through the valley of the shadow of death.

I cannot express my gratitude enough to my family. Their love and unwavering support is the biggest reason I can survive through the darkest hour.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Nowadays computation is ubiquitous. Machine Learning (ML), especially Deep Learning, has dominated many aspects of our daily lives, such as speech recognition [1], image processing [2], medical decision making [3], and autonomous driving [4], etc. Scientific computations such as Algorithmic Differentiation [5] and N-Body simulations [6] usually require a lot of computation resource. Moreover, by bringing devices and services together, the Internet of Things (IoT) is widely used in the industry [7], such as product quality monitoring, prediction, and maintenance, as well as in smart home automation. To extract insights from the rich set of data acquired, computation is a key component in the IoT system [8]. Therefore, numerical libraries lie in the heart of modern applications.

Such complex real-world computation used to be mainly conducted on resource-rich cloud computing infrastructures. This architecture has issues such as high service response latency, communication cost, single point failure, and most importantly, data privacy concerns. The privacy issue has been a recurring topic in the Internet Trends Report in recent years [9]. With social awareness of privacy and personal data arising, centralised data storage and analysis has attracted criticism [10]. Recently edge devices are developing fast; even Edge TPU [11] is not difficult to access. These factors promote a trend of decentralised computation deployment on edge devices. This requires a thorough understanding of numerical libraries at various levels, from how each operation works up to the applications.

1.1 Performance of Computation

It has always been the focus of numerical libraries to improve the performance of computation by exploiting parallelism of both hardware and software [12]. It involves multiple levels computation.

There exist several forms of parallel execution in modern processors. A super-scalar processor exploits instruction-level parallelism within instruction stream itself and executes different instruction in parallel. Hardware also supports multi-threading by providing multiple execution contexts in the L1 cache of one core, so as to hide latency in memory access. These features are solely decided by hardware. Adding ALUs in a processor core can also increase compute capability. One implementation is the "single instruction multiple data" (SIMD) approach. That means multiple elements are executed in parallel on all ALUs using the same instructions. Besides, multiple cores can process instruction streams in parallel, with the help of libraries such as pthread or OpenMP.

Besides utilising hardware, another factor in improving the performance of operations is the specific algorithms implemented, especially for core operations such as the convolution. For example, there have been on-going research work on algorithms of the

convolution operation to keep optimising its execution speed and memory usage: the `im2col` [13], the FFT-based algorithm [14], and the Winograd algorithms [15]. The choice of implementation algorithm is also one important factor that a numerical library should consider.

Given different application scenarios, such as different hardware or input specification, a library often needs to fine-tune the implementation used, so as to better benefit from the hardware/software parallelism features. Automatically Tuned Linear Algebra Software (ATLAS) [16] is a BLAS implementation, featuring automatically-tuned routines on specific hardware. MKL-DNN [17], a highly optimised implementation on Intel processors, provides sophisticated implementation routines for different input and kernel shapes of the convolution operations.

Finally, the parallelism is extended from multiple cores on one machine to multiple machines that are connected by a network. There are three parallel programming models: shared address space, message passing such as MPI [18], and data parallel. The crucial factor at this level is the communication instead of computation itself.

1.2 Computation Graph

The Computation Graph (CGraph) was first introduced by Karp and Miller in [19]. The idea is to present a sequence of computation with a directed graph. Computation steps correspond to nodes of a graph, and dependency between steps is denoted by edges in the graph. This simple idea is the key to many numerical applications such as Automatic Differentiation [20]. CGraph is the core data structure in popular deep learning libraries, such as TensorFlow [21] and PyTorch [22]. Recently there have been efforts to introduce standards for the neural networks so as to exchange neural networks among various frameworks. Such standards include Open Neural Network Exchange (ONNX) [23] and Neural Network Exchange Format (NNEF) [24]. These standards all rely on the computation graph to provide unified interfaces for neural networks.

1.3 Computation Composition and Deployment

Recently, computation on edge and mobile devices has grown rapidly. Many challenges arise when moving computation from cloud to edge devices, such as limited computation power, and personalising analytics models. However, one problem is not yet well defined or investigated: the deployment of computation such as machine learning services. Most machine learning libraries such as TensorFlow and Caffe focus mainly on the training of analytical models. On the other hand, end users mainly use trained models to perform inference. This gap between the current numerical systems and users' requirements is growing.

Another challenge in conducting computation on edge devices is model composition. Training a model often requires large datasets and rich computing resources, which are often not available to normal users. The composition of computation as a service is still not well investigated. The basic idea is that many common building-block computations, such as image detection, can be reused by composition instead of being created again and again in different applications.

Two related topics are model serving [25] and serverless computing [26]. Containerisation as a lightweight virtualisation technology has gained enormous traction. It is used in deployment systems such as Kubernetes. Another backend is JavaScript. Using JavaScript to do analytics aside from front end development have begun to attract inter-

est from academia and industry, such as Tensorflow.js and Facebook's Reason language. Unikernels such as MirageOS [27] is another option.

1.4 Collaboration of Computation

Once the computation is distributed across multiple nodes, collaboration is often required for effective data processing, for example in collaborative sensing [28] or edge coordination in IoT [29]. One notable example is the distributed training of machine learning models, and it is still a hot research topic on which various techniques are developed [12]. One recent application field is Federated Learning [30] where the participating workers tend to be unreliable.

A key aspect in collaborative parallel computation is the barrier control method, which decides the synchronisation of nodes. Currently three schemes are widely used: Bulk Synchronise Parallel (BSP) [31], Stale Synchronise Parallel (SSP) [32], and Asynchronous Parallel (ASP) [33]. The Probabilistic Synchronous Parallel (PSP) [34] barrier control method is recently proposed to accommodate a heterogeneous computational environment. It does not require a centralised node to hold the global state, and introduces a new dimension in the tuning space of barrier design. A thorough evaluation of barrier control methods in a large tuning space remains a challenge.

1.5 Motivation and Contribution

This thesis is motivated by my development work on Owl [35]. It is an OCaml-based numerical library that aims to provide full-stack support for numerical methods, scientific computing, and advanced data analytics. I have been participating in its developing and maintaining for more than three years in a team of 2-3 members, and have contributed more than 30,000 LoC to the code base. Currently I am leading writing a book about OCaml scientific computing based on the Owl library[1]. I keep using applications such as image classification to drive the development, including testing the coverage of functionality, validating design of code architecture, etc. During the development, I find that optimising a numerical application often needs to be considered from multiple layers in the library.

According to current practice, optimising an application often requires the work of different parties. Numerical libraries take care of the performance of computation, and a computation can be executed on various platforms. Applications should be re-configured when deployed to different scenarios. When the deployed computations need to collaborate, distributed computation engines are used. This practice divides the whole lifecycle of computation to different parties, and thus leads to non-ideal performance. To this end, I propose the hypothesis that, a numerical library can optimise its applications at multiple levels by taking a bottom-up approach.

This approach of holistic coverage of application in a numerical library has already shown certain signs. Previously, a numerical library mainly focused on a limited number of core functionalities, such as the ndarray provided by NumPy. While that is still true today, one growing trend is that a numerical library incorporates more and more functionality centred on the core functionalities. The automatic differentiation and XLA graph support are integrated into NumPy by JAX [36]. Julia also expands to multiple disciplines such

[1]OCaml Scientific Computing, Liang Wang and Jianxin Zhao, Online Draft URL:`https://ocaml.xyz/book/`

as parallel computation [37] and mathematical optimisation [38]. However, the coverage of these libraries is still in a fragmented way.

Extending this trend, this thesis proposes four different levels of optimisation for numerical applications: basic operations, computation graph, computation composition, and barrier control in computation synchronisation. Particularly, my research contributions at these levels are listed below:

- I present the low-level design of numerical operations and show the related impact on performance optimisation. I also demonstrate the relationship between numerical operations and high level applications such as Deep Neural Network (DNN) inference and training. I create new algorithms for key operations, and present an automatic tuning module to further improve performance of the basic operations in the Owl library.

- Multiple operations can be constructed into a computation graph. At the graph level, I present the idea of using graphs as an intermediate representation to enable interoperability with other computation frameworks and devices. To demonstrate this approach, I build the TFgraph system that provides a symbolic representation layer to exchange the computation between Owl and other frameworks.

- At a higher level, the computation graph can be seen as a unit, and at this level I identify the problems of computation composition and deployment. I build the Zoo system to address these two problems. It provides a small Domain-specific Language to enable composition of advanced data analytics services. Benefiting from OCaml's powerful type system, the Zoo provides type checking for the composition. Besides, the Zoo DSL supports fine-grained version control in composing. It also involves deploying composed services to multiple backends.

- Finally, at the level of collaboration of multiple deployed computations, I focus on the barrier control methods, propose two quantitative metrics to evaluate existing barriers, especially the PSP, and bring new insights into the barrier control method design. I have also built a simulation platform and real-world experiments to perform thorough evaluation of the PSP compared to existing barrier methods.

1.6 Thesis Outline

In this chapter I have briefly introduced the field of numerical computation, from the basic parallelism in the implementation of single operation, to computation as a graph, the composition and deployment of computation in different scenarios, then to the collaboration of deployed computation on multiple devices. I have stated the motivation and research topic of this thesis, inspired by my experience developing the Owl numerical library. In Chapter 2 I give a background introduction to the material involved in the original work in this thesis.

In Chapter 3 I describe the low-level architecture design of the Owl library. I first describe some advanced DNN applications constructed in Owl, and how they can be reduced to basic operations provided by the library. I then categorise these operations, and pick representative ones to demonstrate the low-level design of these basic operations and optimisation opportunities by comparing with existing NumPy libraries. Finally, I present a performance tuning module in Owl, to provide tuned performance of vector

operations on different hardware platforms. This chapter is mainly based on my code contribution to the master branch of Owl source code.

In Chapter 4 I present the idea of using Computation Graph as an intermediate representation to promote computation interoperability. Based on the previous chapter, to further improve the performance of computation, it is imperative to provide mechanism in Owl to execute computation on various hardware accelerators. I introduce the computation graph in TensorFlow, the targeted platform to export computation graph to. Next I present the design of system TFgraph, to convert the computation graph from Owl to TensorFlow. I then present several real-world applications of this system.

The previous chapter is about a single computation in the form of computation graph, and then Chapter 5 centres on the topic of computation composition and deployment. I identify two challenges that are not yet well explored in the literature about data analytics on edge devices: service composition and deployment. I then present the Zoo system to address the previous two challenges. I design concise Domain-specific Language (DSL) to enable composition of different data analytics services, and also deploy them to multiple backends. Finally, I present a use case to demonstrate the expressiveness of the DSL, and thoroughly evaluate different deployment backends for analytics services. The work in this chapter is based on a paper on which I collaborated with Tudor, Richard and Liang [39].

In Chapter 6 the topic is collaboration of deployed computational services , especially in the barrier control methods in distributed training. I propose two metrics to evaluate the barrier control methods, and conduct a thorough evaluation of and provide insight into Probabilistic Synchronous Parallel (PSP) using these metrics. My evaluation work is based on the work of PSP that is proposed by Liang, Ben, and Richard in [34]. I also participate in its subsequent work that was submitted to the SysML2020, which this chapter is based upon.

In Chapter 7 I summarise the contributions made in this thesis. I then list various paths for extending my work which could be carried out in future.

Chapter 2

Background

This thesis aims to present an approach to optimise numerical applications at different levels, from performance of basic operations to barrier control between computation nodes. In this chapter, I review the background literature about optimisation at these different levels. In Sec. 2.1, I present the classification of existing numerical libraries, the optimisation techniques that can be used. I then introduce the Owl library. In Sec. 2.2, I briefly explain the idea of a computation graph, its application in existing numerical libraries, and a recent trend of neural network compilers that are based on computation graph. At a higher level, a whole computation can be served for easy access to end users. In Sec. 2.3, I introduce the current research work and products on model serving systems. Finally, to collaborate among computations, such as in distributed model training, a key factor is the barrier control method. I introduce the related work in Sec. 2.4.

2.1 Numerical Libraries

There are two widely used specifications of low level linear algebra routines. Basic Linear Algebra Subprograms (BLAS) [40] consists of three levels of routines, from vector to matrix-vector and then to matrix-matrix operations. The other one, Linear Algebra Package (LAPACK) [41], specifies routines for advanced numerical linear algebra, including solving systems of linear equations, linear least squares, eigenvalue problems, SVD, etc.

The implementations of these specifications vary in different libraries, e.g. OpenBLAS [42] and Math Kernel Library (MKL) [17]. OpenBLAS is a popular open source optimised BLAS library. MKL is a proprietary library, and provides highly optimised mathematical functions on Intel processors. It implements not only BLAS and LAPACK but also FFT and other computationally intensive mathematical functions. Another implementation is Eigen [43], a C++ template linear algebra library. The CPU implementation of many kernels in TensorFlow uses the Eigen Tensor class[1]. The Automatically Tuned Linear Algebra Software (ATLAS) [16] is another BLAS implementation, featuring automatically-tuned routines on specific hardware.

These basic libraries focus on optimising the performance of operations in different hardware and software environments, but they do not provide APIs that are easy to use for end users. That requires libraries such as NumPy, Julia, MATLAB, and Owl. NumPy is the fundamental package for scientific computing with Python. It contains a powerful N-dimensional array abstraction. Julia is a high-level, high-performance dynamic programming language for numerical computing. Both are widely used and considered state

[1] TensorFlow Architecture, `https://www.tensorflow.org/guide/extend/architecture`

of the art in numerical computing [44, 45]. Both NumPy[2] and Julia[3] rely on OpenBLAS or MKL for linear algebra backends. MATLAB, the numerical computing library that has millions of uses worldwide, also belongs to this category[4].

Deep learning libraries such as TensorFlow [21], PyTorch [22], and MxNet [46] are popular. Keras [47] is a user-friendly neural networks API that can run on top of TensorFlow. Instead of the wide range of numerical functionalities that NumPy etc. provide, these libraries focus on building machine learning applications for both research and production. The Owl library provides its own neural network module.

Besides machine learning, the uses of numerical library are wide spread in many fields. Just to name a few of them here, Fast Fourier Transform (FFT) is a core technique for signal processing, and thus used in our daily lives: mobile phones, image and audio compression, communication networks, large scale numerical physics and engineering, etc. Solving differential equations is studied ever since the invention of calculus, driven by the applications in mechanics, astronomy, and geometry, etc. Regression is one key topic for predictive analytics and decision making in business and finance etc., as well as the traditional mathematical subjects such as probability, statistics, and linear algebra. They all require efficient numerical implementation to be applied in real world applications.

Of course, I cannot cover all of these applications in this thesis. In the next chapter, I will use a deep neural network as motivative application to demonstrate how such an advanced application can be constructed with basic components in the Owl library and how the library can be optimised layer by layer. In the next section, I introduce the background of numerical computation optimisation.

2.1.1 Optimisation of Numerical Computation

To achieve good performance with regard to execution speed and memory usage has always been the target of numerical libraries. With the end of the Moore's law, it is not straightforward anymore to free-ride the performance boost gained from hardware upgrade. Faced with applications of ever-growing complexity, a numerical library needs to utilise various technologies to improve its performance. Below, I list some of the techniques that I use to optimise operations in Owl.

SIMD One method to utilise the parallelism of a computation platform is to use the Single Instruction Multiple Data (SIMD) instruction sets. They exploit data level parallelism by executing the same instruction on a set of data simultaneously, instead of repeating it multiple times on a single scalar value. One way to use SIMD is to rely on the automatic vectorisation capabilities of modern compilers, but in many cases developers have to manually vectorise their code with SIMD intrinsic functions. The Intel Advanced Vector Extensions (AVX) instruction set [48] is offered on Intel and AMD processors, and the ARM processors provide the NEON [49] extension.

Multi-processing Another form of parallelism is to execute instructions on multiple cores. OpenMP [50] is a C/C++/FORTRAN compiler extension that allows shared memory multiprocessing programming. It is widely supported on compilers such as GCC and Clang, on different hardware platforms. It is important for a numerical software to port

[2]Building from source, NumPy Documentation, `https://numpy.org/devdocs/user/building.html`

[3]Intel MKL linear algebra backend for Julia, `https://github.com/JuliaComputing/MKL.jl`

[4]MATLAB Incorporates LAPACK, `https://uk.mathworks.com/company/newsletters/articles/matlab-incorporates-lapack.html`

existing code to the OpenMP standard, especially when it needs to utilise multiple cores efficiently to perform computation-heavy tasks [51].

Parameter tuning To achieve good performance often requires choosing suitable system parameters on different machines or for different inputs. Aiming at providing fast matrix multiplication routines, the ATLAS library [16] runs a set of micro-benchmarks to decide hardware specifications, and then search for the most suitable parameters such as block size in a wide tuning space. In [52] the authors propose to tune the performance of machine learning libraries such as TensorFlow by exploiting multiple levels of parallelism.

Multiple dispatch One general algorithm cannot always achieve optimal performance. In [53], the authors explain how the Julia library achieves high-performance computation. One of the most important techniques they use is "multiple dispatch", which means that the library provides different specific implementations according to the type of inputs.

Besides these techniques, the practical experience from others always worth learning during development. The authors in [54] share their years of development experience on linear algebra libraries. The tutorials in [55] provide a set of general techniques to improve the performance of numerical operations, such as cache optimisation and loop unrolling. These principles still hold true in the development of modern numerical libraries. It has shown that an optimised routine can perform orders of magnitude faster than a naive implementation.

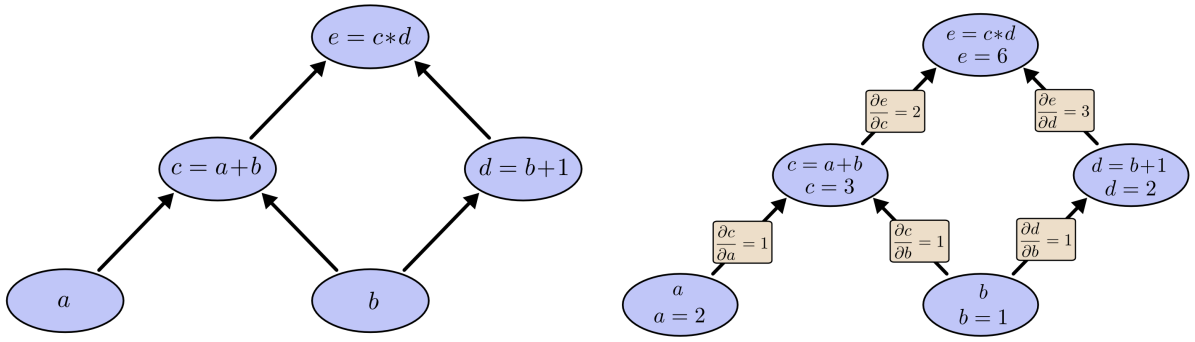
2.1.2 Owl Numerical Library

Owl [35] is a library for scientific and engineering computing in the functional programming language OCaml. Instead of being just a collection of mathematical functions, Owl provides full stack support for numerical computing and analysis. At the core of Owl are the basic numerical operations on N-dimensional arrays. The computation is supported by C language implementation and low-level libraries including OpenBLAS and LAPACKE, or the pure OCaml implementation that is contained in the `owl-base` library in Owl. Owl also provides a computational graph module that can make the existing N-dimensional array computation into lazy evaluation mode. Based on these core operations are advanced functionalities, including statistics, linear algebra, regression, algorithmic differentiation, optimisation, etc. At a higher layer, Owl supports building machine learning and neural network applications. A parallel and distributed engine is also developed to further maximise developers productivity using Owl.

Benefiting from OCaml's features such as static type checking and functors, Owl provides a concise and yet powerful syntax. For example, the total lines of code (LoC) of the neural network module is only about 4,500, which is much smaller than that in existing machine learning libraries such as TensorFlow. But a complex InceptionV3 [56] deep neural network structure can be built with less than 100 LoC, while yielding high performance at the same time.

The Owl library has been developed and maintained by a small team. I have been participating in the developing and maintaining of Owl library, and have contributed more than 30,000 LoC to the code base[5], upon which this thesis is based. Our team is small, but we have made a lot of efforts in improving its performance towards state of the art. After more than three years of intense development, the current code base contains more than 230K lines of code. On OPAM, the OCaml package manager, the Owl package

[5]Source code available at GitHub: `https://github.com/owlbarn`.



(a) Computation graph (b) Computation graph execution and derivative

Figure 2.1: Simple computation graph example.

is installed more than 700 times in the last month alone. Compared to other available libraries such as lacaml and oml, Owl is already the *de facto* tool for computation intensive tasks in OCaml. It is also included in the Hacker News several times [6]. Currently Owl is still under active development, such as the recent work on ordinary differential equation solver [57] and GPU support [58].

2.2 Computation Graph

Computation graph is first introduced by Karp and Miller in [19]. The idea is to present a sequence of computation with a directed graph. Computation steps correspond to nodes of a graph, and dependency between steps is denoted by edges in the graph. Graph is an intuitive way to represent computation. For example, consider a simple expression $x = (a*b)+(b+1)$. This computation consists of three operations: two additions and one multiplication. To create a computation graph requires making these operations, along with the input variables, into nodes. A direct link connects an input to an output, as shown in Fig. 2.1[7]. A fixed computation graph accepts various inputs. To evaluate this expression, set the input variables a and b in the graph to certain values and compute output of each node by order.

This simple idea is a key to many numerical applications. For example, Algorithmic Differentiation [20] (AD, or Automatic Differentiation), the technique that numerically evaluates the derivative of a function, is one important application. Differentiation is crucial in many scientific related fields, such as finding maximum or minimum values, solving ordinary differential equation, and non-linear optimisation etc. One new crucial application of AD is in machine learning. Compared to numerical or symbolic differentiation, AD can generate exact results with superior speed and memory usage, therefore highly applicable in various real world applications. In AD, a function can be represented by a computation graph. To obtain a derivative of this function, one can use backpropagation [20] on the computation graph. It means computing the partial derivative of each node backward in the computation graph, accumulating them according to the chain rule in calculus, and finally getting gradient of the input.

Graph is also a core data structure in popular deep learning libraries, such as TensorFlow and PyTorch [22]. When constructing a neural network, a computation graph is built (the *forward* graph). A "forward pass" propagates input data through its param-

[6] Owl: OCaml Scientific Computing, Hacker News, `https://news.ycombinator.com/item?id=20449595`

[7] Figure source: `https://colah.github.io/posts/2015-08-Backprop/`

eters towards a loss function to get an error value. This phase alone is also called the inference phase. By calculating the gradient of this output using algorithmic differentiation, a *backward* graph can be built. In this process, the error value is propagated in the reverse direction to get derivatives of variables in the graph. Training a neural network mainly consists of iteratively executing these two steps to keep updating the variables in a network.The *weight* of a neural network consists of the variable values in it.

Recently there are efforts to standardise the neural network format so as to exchange neural networks between various platforms. Such standards include Open Neural Network Exchange (ONNX) [23] and Neural Network Exchange Format (NNEF) [24]. These standards rely on computation graphs to provide unified interfaces for neural networks.

2.2.1 Computation Graph in Machine Learning Libraries

In current machine learning libraries, computation graphs are implemented in the form of either dynamic graphs or static graphs. A dynamic graph is constructed during the runtime, and thus provides good flexibility. PyTorch is a typical dynamic graph library [22]. On the other hand, a static graph has the benefit of high performance since the structure of a graph is known and thus various optimisations can be applied. TensorFlow uses static graphs [21]. Owl [59] builds dynamic graphs and converts them into static graphs at runtime, achieving both expressiveness and performance. The computation graph module in Owl is implemented with a functor stack, parameterised by both number type and device type. At the centre of this stack is the optimisation layer. It implements various traditional optimisation techniques such as constant-folding, operation fuse, and operation-dependent pattern-based optimisation. The work in [60] also exploits reusing memory based on a "pebble game" in the computation graph to efficiently reduce memory during tasks such as DNN training.

Graph is a core abstraction for computation in TensorFlow. A graph is first defined via the Python frontend in a session. This graph is then converted into the C++ defined graph via the C API for backend runtime processing. The backend processing includes operation partitioning, graph pruning, graph optimisation, and execution. Therefore, the whole lifetime of a computation is in the form of a graph in TensorFlow.

Accelerated Linear Algebra (XLA) is a compiler project by Google. Based on TensorFlow, it provides a High-Level Optimization (HLO) IR for specifying multi-linear algebra computations and provides code generation capabilities for various hardware backends, e.g. CPU, GPU and TPU (Tensor Processing Unit) [61]. In TensorFlow, the XLA Just-in-Time (JIT) compilation step is counted as an extra graph optimisation step, and all the operations are fused into a new `XlaLocalLaunchOp` in the graph. The backend computation graph in TensorFlow consists of operations, and `XlaLocalLaunchOp` is one special operation among them. Once the `compute` method of this operation is called, the computation it contains can be executed without involving runtime.

One advantage of XLA is that it targets the TPU as hardware backend, which has shown extraordinary performance on neural network workloads. According to [62], Julia has begun to interface to XLA by generating the HLO IR that XLA uses and then utilises the XLA compiler to export code to TPU devices. PyTorch [63] is also exploring a similar approach.

2.2.2 Neural Network Compilers

Recently, a trend of neural network compilers begins to emerge. These compilers aim to port deep learning workloads across diverse hardware backends. The computation graph

is a key structure in such compilers.

TVM [64] proposes two layers of intermediate representation of computation. The higher layer is computation graph. At this layer, graph optimisation techniques such as constant-folding and operator fusion are applied. At the lower layer, a graph is represented by a tensor expression language which is extended based on Halide [65] so as to support new TVM-specific optimisations, i.e. latency hiding and nested parallelism. Hardware features such as data layout are also considered at this layer. The hardware-specific code can then be generated. Glow [66] presents a similar two-phase design. The high level IR is a data flow node-based graph that is similar to a computation graph. Its lower-level IR is mainly for performing memory-related optimisations such as static memory allocation and copy elimination. One commonly used backend by these compilers is LLVM [67].

In nGraph [68], another deep learning compiler, the lower-level optimisation focuses on partitioning the graph for parallel execution or assigning subgraphs to the most suitable backends. Code is then generated to utilise libraries such as MKL and Eigen. The parallel execution of graphs is also explored in [69]. In this work, the authors propose an execution engine to efficiently run a computation graph in parallel on multi-core CPUs.

Deep Neural Network is not the only targeted application of these compilers. DLVM [70] focuses on the linear algebra computation. It proposes an intermediate layer that features algorithmic differentiation. The optimisations at this layer are mainly algebra simplification, linear algebra fusion, and matrix multiplication reordering, etc. The focus of RLGraph [71] is Reinforcement Learning. Given the existing diversified libraries, the authors propose Component Graph, an abstraction to express the logic in reinforcement learning flexibly. At the low level, a computation graph is generated and executed on various backends.

2.3 Model Serving

Zoo [39] is a subsystem in Owl for service composition and sharing. The idea of Zoo originates from creating a repository of machine learning models that are built with Owl for easy access. This practice is now not uncommon to see in many machine learning libraries, such as the Caffe model collection [72].

For easy access to existing models, a serving system is required. Clipper [25] is a low-latency prediction serving system. It enables users to access models based on multiple machine learning frameworks. It also focuses on reducing serving latency by applying techniques such as caching, batching, model selection, straggler mitigation, etc. The models are utilised as containers. TensorFlow also provides its own serving system, TensorFlow Serving [73]. Google's Cloud ML Engine is very similar to TensorFlow Serving, but does not include containers. Instead it uses TensorFlow's `SavedModel` format for models. Laser [74] is a serving system that focuses on Logistic Regression models to perform prediction on large scale data.

Since the publication of our work on Zoo, several microservice deployment systems have been developed, such as Seldon [75]. Like Clipper, it also uses Docker container for models. The container-orchestration system Kubernetes is also utilised. Seldon defines inference graph based on the model and then deploys the graph with a single command of `kubectl` in deployment or production environments.

Another trend in model serving is the Serverless Computing, where cloud providers run the server, and dynamically manage the allocation of machine resources for user-defined light-weight functions. Utilising the powerful ecosystem of existing cloud providers, the functions can be deployed on different types of devices to get access to resources such as database and cloud files, such as in Amazon Lambda [76] and Azure Functions [77].

OpenLambda [78] is an open source version of these serverless platforms that still relies on cloud infrastructure. Based on the observation that existing serverless systems spend a large portion of time on booting function containers and interaction between functions, the SAND system [26] investigates the combination of different functions. By proposing application-level sandboxing and a hierarchical message bus, this system reduces latency and improves resource utility.

While container is a dominant format of models in serving systems, other potential alternatives exist. In [79], the authors propose a unikernel virtual machine that is based on Xen [27], which achieves faster boost time and provides better resource isolation. MirageOS [27] is a library operating system that constructs unikernels across a variety of cloud computing and mobile platforms. Using a unikernel or similar light-weight virtual machines in deploying computation on various types of device is a promising direction in various scenarios such as for network functions virtualization [80].

2.4 Barrier Control

A computation can be composed and deployed to multiple nodes, and an application often requires these nodes to collaborate among each other. An important example is the training of machine learning models. A lot of research has been conducted on improving the performance in distributed training. In this section, I focus on introducing the background information of barrier control methods, the mechanism that provides trade-off between accuracy and progress.

2.4.1 Distributed Training

One factor in the design of machine learning is how the cluster of computation nodes are organised. Previously, machine learning models were trained in a centralised approach on a single node. This relies on the development of hardware accelerators such as GPU or TPU [81]. With growth in the size of data and models, however, the decentralised machine learning approaches become more popular, with data or model divided among various machines [31].

The decentralised computation nodes can be organised in different forms. In MapReduce [82], the nodes are divided to run one of the two tasks: map, or reduce. The mappers compute in parallel, and the reducers get the output from all mappers and combine them to give the accumulated result. The updated parameters are then broadcast to all machines. The AllReduce abstraction [83] is often implemented by organising the nodes into a tree structure. The communicating consists of two phases: the weights are aggregated up from all nodes to root along the tree, and the results are broadcast from root down to all nodes. The Parameter Server [84] topology is similar. The nodes are divided into servers that hold the shared global view of the up-to-date model parameters, and workers that each holds its own view of the model and executes training tasks. The workers and servers communicate by key-value pairs. In these approaches, the model parameter storage is managed by a set of centralised servers. In contrast, P2P [85] is a fully distributed structure, where each node contains its own copy of the model, and nodes directly communicate with each other.

Federated Learning [30, 86] is a new machine learning application field where the goal is to train models with data distributed across a large number of workers, while the workers are very likely placed in heterogeneous and unreliable environments. The workers could be mobile phones or IoT devices, and therefore not just training accuracy but also

energy usage and communication efficiency are important. This paradigm brings new challenges and opportunities to distributed machine learning.

2.4.2 Barrier Control

The iterative-convergent nature and error-proneness of ML applications mean that the requirement for consistency in the order of computation updates can be relaxed without sacrificing accuracy, with gains in system performance at the same time. This trade-off is decided by the *barrier control* in distributed ML.

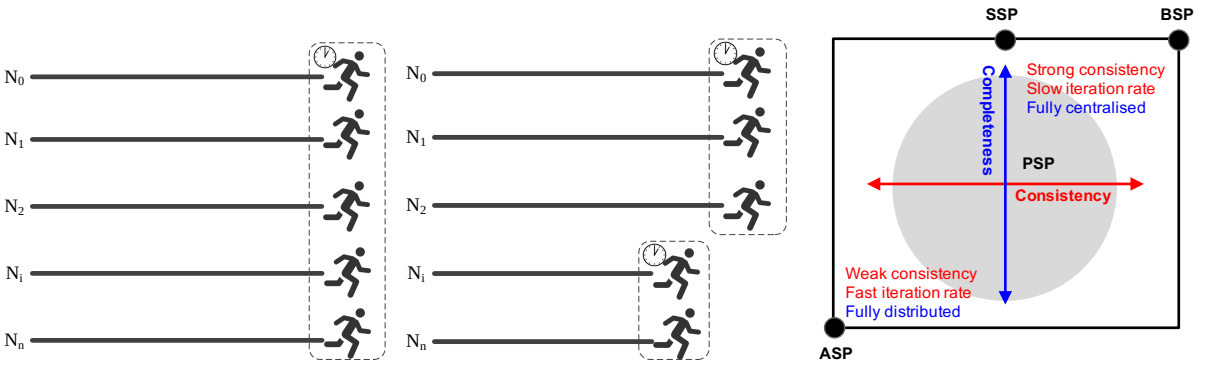
Existing distributed processing systems operate at various points in the space of consistency/speed trade-offs. There are three main stream synchronisation mechanisms: Bulk Synchronise Parallel (BSP), Stale Synchronise Parallel (SSP), and Asynchronous Parallel (ASP).

Bulk Synchronous Parallel (BSP) is a deterministic scheme where workers perform a computation phase followed by a synchronisation/communication phase where they exchange updates [31]. Provided the data and model of a distributed algorithm have been suitably scheduled, BSP programs are often equivalent to sequential computation. This means that the correctness guarantees of the serial program are often realisable, making BSP the most consistent barrier control method [32]. However, as workers must wait for others to finish, BSP leads to a dramatic reduction in performance. Overall, BSP tends to offer high computational accuracy but suffers from poor efficiency in unfavourable environments. BSP is the most strict lockstep synchronisation; all the nodes are coordinated by a central server. One variation of BSP is to allow one execution clock of a worker to contain more than one iteration [87]. Federated Learning in [30] also uses BSP for its distributed computation barrier.

Asynchronous Parallel (ASP) takes the opposite approach to BSP, allowing computations to execute as fast as possible by running workers completely asynchronously [33]. In homogeneous environments, ASP enables fast convergence because it permits the highest iteration throughputs. Typically, P-fold speed-ups can be achieved by adding more computation/storage/bandwidth resources [31]. However, such asynchrony causes updates to be calculated on an old model state and thus harms the training accuracy. ASP is the least strict barrier since no communication among workers is required at all. Every node can progress as fast as it can. It is fast and scalable, but often produces noisy updates. Since ASP was proposed in [33], much work has aimed to relax its tight limit conditions, such as convex functions and sparse updates [88, 89]. In [90] the authors propose a delay-compensated SGD. It tries to mitigate the delayed updates in ASP by compensating the gradients received at the parameter server. Another variant of ASP is introduced in [91]. In the specific scenario of applying ASP in wide-area networks, where communication is a dominant factor, the authors propose to allow insignificant updates to be delayed indefinitely in a WAN.

Stale Synchronous Parallel (SSP) is a bounded asynchronous model that achieves a balance between BSP and ASP. Rather than requiring all workers to be on the same iteration, the system allows the iterations of any two workers in the system to differ by at most s steps, a pre-defined bounded staleness. This limits the potential error. The staleness parameter allows SSP to provide deterministic convergence guarantees [31, 32, 92]. Overall, SSP offers a good compromise between fully deterministic BSP and fully asynchronous ASP, despite the fact that the central server is still needed to maintain the global state to guarantee its determinism nature.

SSP is further exploited in many other research works. The authors of [93] investigate *n-softsync*, a synchronisation method that makes the parameter server updates its weight



(a) BSP (b) Probabilistic BSP (pBSP) (c) The extra dimension exposed by PSP.

Figure 2.2: Probabilistic Synchronise Parallel.

after collecting certain number of updates from any workers. Another variant is to proceed whenever P (the total number of workers) gradients are collected, such as that in [88]. [94] propose to remove a small amount of "longtail" workers, or add a small amount of backup nodes to mitigate this effect while avoiding asynchronous noise.

2.4.3 Probabilistic Synchronous Parallel

In [34], the authors propose a new barrier control technique called Probabilistic Synchronous Parallel. The idea of PSP is simple: in a unreliable environment, the impact of outliers and stragglers can be minimised by guaranteeing parts of the system have synchronised boundaries. The reason that results from a certain portion of workers can be dropped is that, practically, many iterative learning algorithms can tolerate a certain level of errors in the process of converging to final solutions [95]. In PSP, either a central server tracks the progress of each worker or the workers each hold their own local view. Each node only depends on several other nodes to decide its own barrier.

One great advantage of PSP is its compatibility with existing synchronisation methods. For classic BSP and SSP, their barrier functions are invoked by a central server to check whether or not to let a node cross the synchronisation barrier based on the global states. With the proposed `sampling` primitive in PSP, almost nothing needs to be changed except that only the sampled states instead of the global states are passed into the barrier function. There the probabilistic version of BSP and SSP are derived, namely *pBSP* and *pSSP*. Fig. 2.2a and Fig. 2.2b show how BSP and PSP can be composed by dividing the nodes into (possibly overlapping) subsets.

Using sampling allows decoupling the degree of synchronisation from the degree of distribution, introducing *completeness* as a new axis by having each node sample from the population. Within each sampled subgroup, traditional mechanisms can be applied allowing overall progress to be robust against the effect of stragglers while also ensuring a degree of consistency between iterations as the algorithm progresses. The result is a larger design space for synchronisation methods when operating distributed data processing at scale. As Fig. 2.2c summarises, probabilistic sampling shows greater flexibility in designing synchronisation mechanisms in distributed processing systems at scale.

While BSP and SSP require a centralised node to hold the global state, PSP does not require such information and thus can be executed independently on each individual node, further leading to a fully distributed solution. Therefore, PSP is especially suitable for data analytic applications deployed in large and unreliable distributed systems.

2.5 Conclusion

In this chapter, I introduced and reviewed the existing literature about optimisation at different levels in numerical computation. First I showed a classification of existing numerical libraries and the optimisation techniques that can be used. Specifically, I introduced the Owl library that I have participated in developing. These are the background materials during my development of efficient operations in Owl, as I will describe in Chapter 3. Then I briefly introduced the computation graph, its application in existing numerical libraries. Based on this information, in Chapter 4 I will propose the TFgraph system and show the idea of using computation graph as intermediate representation. Next, the current research work and products on model serving systems are related to the Zoo system I will propose in Chapter 5. Finally, to collaborate among computations, one key factor is the barrier control method. I presented an overview of existing barrier control methods in the last section. Based on this work, I propose new metrics to bring insights in evaluating barriers, especially the PSP, as I will show in Chapter 6.

Chapter 3

Optimising Basic Numerical Operations

An efficient implementation of the numerical library is crucial to many data analytics applications. I have been working on the Owl numerical library [35], and specifically I have implemented and improved the performance of several key operations. My contributions in this chapter are based on my first-hand experience in developing Owl. I demonstrate the low-level design of a numerical library and show the related impact on performance optimisation. I also demonstrate the relationship between numerical operations and high level applications such as Deep Neural Network inference and training. I implement new algorithms for key operations, and build an automatic tuning module to further improve performance of the basic operations.

3.1 Numerical Operations

An N-dimensional array (ndarray) is the core data type of a numerical library. It lies in the heart of Owl, and many other libraries. The NumPy focuses solely on providing a powerful ndarray module to the Python world. An ndarray is a container of items of the same type. It consists of a contiguous block of memory, combined with an indexing scheme that maps N integers into the location of an item in the block. A stride indexing scheme can then be applied on this block of memory to access elements.

A numerical library needs to provide basic operations that run on or with ndarray in some fashion, typically returning ndarray(s) as result. According to their implementation mechanism and interfaces, these operations can be categorised as shown in Table 3.1. Advanced applications can be constructed using these basic operations.

3.1.1 From DNN Applications to Operations

As introduced in the previous chapter, the applications of numerical libraries span across numerous fields. Besides computer science, they are widely used in physics, engineering, geometry, etc. To stress the importance of optimisation at the operation level, I first show how a real-world application can be decomposed into the basic operations. Specifically, I introduce real world deep neural network applications that are built with Owl as representative examples. The main reason I choose neural network applications is that they provide complex computation to drive the optimisation of library. Besides, they also provide good examples for the topic of computation graph I will cover in the next chapter. These applications also produce impressive output in demonstration. Note that

Table 3.1: Categorise core operations of ndarray.

Category	Operation example
Unary Arithmetic	copy, abs, exp, log, sqrt, reci, cbrt, sin, tan, asin, sinh, asinh, round, erf, sort, sigmoid
Binary Arithmetic	add, mul, div, pow, hypot, min2, fmod
Reduction Arithmetic	max, sum, prod, cumprod, cummax
Item manipulation	repeat, slicing, partition, sort
Shape manipulation	reshape, flatten, squeeze
Array conversion	copy, view, dump, tostring
Linear Algebra	qr, lu, svd, inv, matmul, eigenvals
Neural Network	convolution, pooling

Figure 3.1: An example of Neural Style Transfer.

Owl is not yet another neural network framework, and the analysis in this chapter can be extended to other applications built with Owl.

Owl provides fully functioning neural network module for deep learning applications. Compared to existing deep learning platforms, Owl utilises some ideal properties of OCaml language: fast, and strong static typing with type inference. Moreover, it is a good example to show the expressiveness of Owl code in constructing neural networks. A user can construct a DNN application with short and elegant code. I have built several real-world DNN applications using Owl, and they will be used later in this thesis.

Image Recognition InceptionV3 [56] is one of Google's latest efforts to perform image recognition. It is trained for the ImageNet Large Visual Recognition Challenge. This is a standard task in computer vision, where models try to classify an image into one of 1000 classes, like "Zebra", "Dalmatia", and "Dishwasher". Compared with previous DNN models, InceptionV3 has one of the most complex network architectures in computer vision. When implemented, InceptionV3 contains of 313 nodes. I have also built other similar architectures, such as ResNet [96], VGG [97] and SqueezeNet [98].

Neural Style Transfer Neural Style Transfer (NST) [2] is the process of using DNN to migrate the semantic content of one image to different styles. The idea is very simple: as Fig. 3.1 shows, this application takes two images A and B as input. Let's say A is a daily street view, and B is The Starry Night of Vincent van Gogh. Then a NST application can produce the same street view, but with the style of Van Gogh. I have implemented an NST application with Owl, which only takes about 180 lines of code. While NST may take a long time to finish, Fast Neural Style Transfer (FST) [99] can further speed up this process to within the order of seconds.

```
let conv2d_layer ?(relu=true) kernel stride nn  =
  let result =
  conv2d ~padding:SAME kernel stride nn
  |> normalisation ~decay:0. ~training:true ~axis:3
  in
  match relu with
  | true -> (result |> activation Activation.Relu)
  | _    -> result

let conv2d_trans_layer kernel stride nn =
  transpose_conv2d ~padding:SAME kernel stride nn
  |> normalisation ~decay:0. ~training:true ~axis:3
  |> activation Activation.Relu

let residual_block wh nn =
  let tmp = conv2d_layer [|wh; wh; 128; 128|] [|1;1|] nn
  |> conv2d_layer ~relu:false [|wh; wh; 128; 128|] [|1;1|]
  in
  add [|nn; tmp|]

let make_network h w =
  input [|h;w;3|]
  |> conv2d_layer [|9;9;3;32|] [|1;1|]
  |> conv2d_layer [|3;3;32;64|] [|2;2|]
  |> conv2d_layer [|3;3;64;128|] [|2;2|]
  |> residual_block 3
  |> residual_block 3
  |> residual_block 3
  |> residual_block 3
  |> residual_block 3
  |> conv2d_trans_layer [|3;3;128;64|] [|2;2|]
  |> conv2d_trans_layer [|3;3;64;32|] [|2;2|]
  |> conv2d_layer ~relu:false [|9;9;32;3|] [|1;1|]
  |> lambda (fun x -> Maths.((tanh x) * (F 150.) + (F 127.5)))
  |> get_network
```

Listing 1: Construct deep neural network using Owl.

To give a glimpse of how a DNN is constructed in Owl, Listing 1 shows an example of using Owl to build a FST network. I have debugged and built the aforementioned application using Owl, and host them online as demos[1].

These DNN applications, as complex as they look, are composed of basic operations in a numerical library. For example, Fig. 3.2 shows the computation graph used when performing inference on a 8-layer DNN. Even for such a small network, the generated graph is still quite complex[2]. This DNN share similar structure as that of the InceptionV3 and Neural Style Transfer networks. In the end, these complex applications are composed of the basic operations such as `convolution`, `add`, and `sum_reduce`, etc. Therefore,

[1] Demo website: `http://demo.ocaml.xyz`

[2] A high resolution version can be viewed at `https://bit.ly/2YYra01`

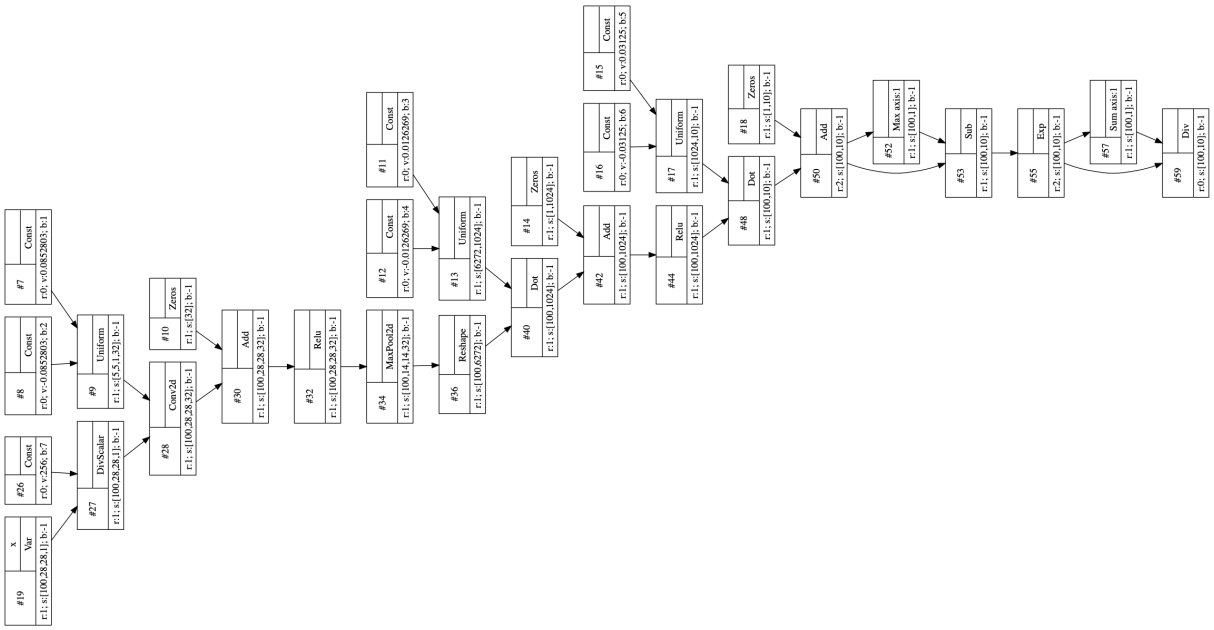


Figure 3.2: Computation graph in the inference phase of a simple deep neural network.

the performances of these basic operations are crucial to the performance of the whole application.

3.2 Operation Optimisation

Despite the efficiency of OCaml, the Owl library relies on C implementation to deliver high performance for core functions. Interfacing to high performance language is not uncommon practice among numerical libraries. If you look at the source code of NumPy, more than 50% is C code. In SciPy, the FORTRAN and C code takes up more than 40%. Even in Julia, about 26% of its code is in C or C++, most of them in the core source code. Besides interfacing to existing C libraries, the Owl team focuses on implementing the core operations in the Ndarray modules with C code.

In Owl, ndarray is built on OCaml's native `Bigarray.Genarray`. The Bigarray module implements multi-dimensional numerical arrays of integers and floating-point numbers, and `Genarray` is the type of `Bigarray` with variable numbers of dimensions. OCaml provides a mechanism for interfacing with C so that the `Bigarray.Genarray` can be manipulated with pointers in C. Once converted properly to the C world, an ndarray can be effectively manipulated with normal C code.

In the previous section, I have shown several DNN applications and how they can be constructed from basic operations in the library. In this section, I choose representative operations from the categories listed in Table. 3.1 and show how we optimise their performance to the state-of-the-art level. The purpose is two-fold: first, to bring insight into the low-level structure design; second, to demonstrate the possible optimisations in implementing these operations[3].

To evaluate the performance of an operation, I measure its execution time and memory usage. The execution time is measured using the functions that the host programming language provides, such as `Unix.gettimeofday()` in OCaml. Each measurement is repeated 20 times to get the average value for more accurate description. For memory

[3]I have conducted evaluation of more operations in Owl; the results can be seen in: `https://ocaml.xyz/chapter/perfcmp.html`.

usage measurement, I use the Valgrind. For both execution time and memory usage measurement, I choose different size and different axes (where applicable) of ndarrays as input.

As comparison, for each operation I compare its performance with that in NumPy and Julia. NumPy is the fundamental package for scientific computing with Python. Julia is a high-level, high-performance dynamic programming language for numerical computing. I choose these two libraries because both are widely used and considered state of the art in numerical computing [44, 45]. Besides, compared with widely used but non-opensource tools such as MATLAB, they provide golden opportunities for me to look deep into the source code and make sure that the performance comparison is as fair as possible. For example, it is necessary to make sure that all the libraries in comparison are linked to the same version of OpenBLAS.

In the performance measurements, I use multiple input sizes, and observe the execution time and memory usage. I focus on edge devices for the evaluation. Recently, there are significant developments in Single Board Computer (SBC) hardware capabilities to support edge compute applications and evaluations [100]. I choose the popular Raspberry Pi 3B (rpi3), Raspberry Pi 4 (rpi4), and Cubietruck boards as the evaluation SBC hardware. The Raspberry Pi 3B model has a 64-bit quad-core ARMv7 CPU of 1.2GHz, while that of the Pi 4 model is a quad-core Cortex-A72 processor of 1.5GHz. The Cubietruck board is less powerful. Its A7 Dual-Core CPU runs at 1GHz. They represent edge devices of different computation power. As to the software, I use the version 1.16.2 of NumPy, version 1.0.2 of Julia, and version 0.8.0 of Owl.

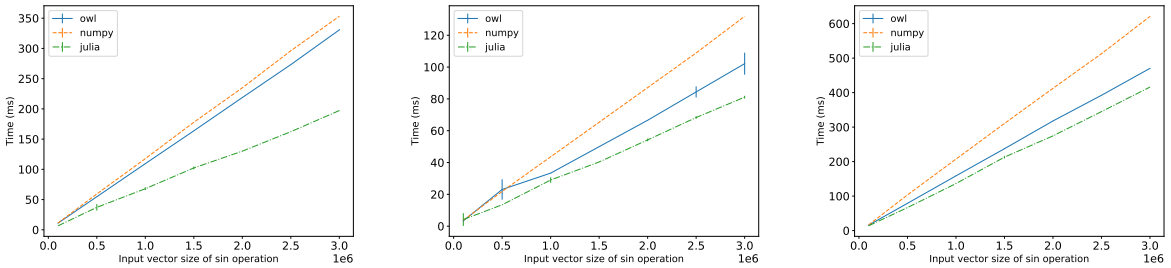
3.2.1 Map Operations

The `map` operations are a family of operations that accept ndarray as input, and apply a function on all the elements in the ndarray. I use the trigonometric `sin` operation as a representative map arithmetic operation in this section. It requires heavy computation. In the implementation, it directly calls the low-level C functions via a single template, as shown below.

```
for (int i = 0; i < N; i++) {
    NUMBER x = *(start_x + i);
    *(start_y + i) = (MAPFN(x));
}
```

It calls function `MAPFN` on one array element-wise, and the result is put in the other ndarray. Here `MAPFN` is a macro in the template. In the case of sine function, it is defined as `sinf` or `sin` function from the C standard library `libc`. Therefore, the performance is mainly decided by the linked low level library, and may be affected by the compiling flags and cost of the wrapper around these libraries. Both vectorisation and parallelisation techniques can be utilised to improve its performance.

Computation-intensive operations such as sine in a for-loop can be vectorised using SIMD instructions. The computational performance can be boosted by executing single instructions on multiple data in the input ndarray. In that way, with only one core, 4 or 8 elements in the for-loop can be processed at the same time. However, the SIMD intrinsics, such as the ones provided by Intel, only support basic operations such as copy, add, etc. To implement functions such as sine and exponential is non-trivial task. There are SIMD implementations of certain mathematical functions such as in [101], and compilers such as GNU GCC also provide options to automatically generate vectorised loops.



(a) Sine operation on rPi3 (b) Sine operation on rPi4 (c) Sine operation on Cubietruck

Figure 3.3: Measure performance of the `sin` operation in Owl, NumPy, and Julia

Map functions can also benefit from parallel execution on the multi-core CPUs, such as using OpenMP. To parallelise a loop in C program, I only need to add a single line of OpenMP compiler directive, as shown below.

```
#pragma omp parallel for schedule(static)
for (int i = 0; i < N; i++) {
    NUMBER x = *(start_x + i);
    *(start_y + i) = (MAPFN(x));
}
```

Now that I have different versions of implementation: normal, SIMD version, and OpenMP version. The mechanism for switching between these different implementations in Owl is to provide another set of operations and switch depending on configuration flags. For example, for the map functions, we have the normal template file `owl_ndarray_maths_map.h`, and then a similar one `owl_ndarray_maths_map_omp.h` where each template uses the OpenMP derivative. We can then switch between these two implementation by simply define or un-define the `_OPENMP` macro, which can easily be done in the configuration file. OpenMP is surely not only utilised in the map function. I also implement OpenMP-enhanced templates for the fold operations, comparison operations, slicing, and matrix swap, etc.

Another optimisation is to remove the memory copy phase by applying mutable operations. A mutable operation does not create new memory space before calculation, but instead utilise existing memory space of input ndarray. This kind of operations does not involve the C code, but rather in the ndarray module, where the destination array and source array are chosen to be the same.

To measure performance, I compare the sine operation in Owl, NumPy, and Julia. The input is a vector of single-precision float numbers. I increase the input size from 100,000 to 5,000,000 gradually. The comparison results are shown in Fig. 3.3. It can be seen that the execution time of this operation grows linearly with input size in all the cases. The three devices show similar results. With a series of optimisations as mentioned previously, Owl slightly out-performs NumPy. Julia performs the fastest in these cases. It is because that Julia utilises NEON, the SIMD architecture extension on ARM architecture. In some cases, NumPy can be compiled with MKL library. The MKL Vector Math functions provide highly optimised routines for trigonometric operations. In this evaluation I use NumPy library that is not compiled with MKL, and it performs close to Owl and C, with slightly larger deviation.

Note the trade-off in code design. My current approach is a common template for all map functions, and relies on the C library for implementation. A specific implementation

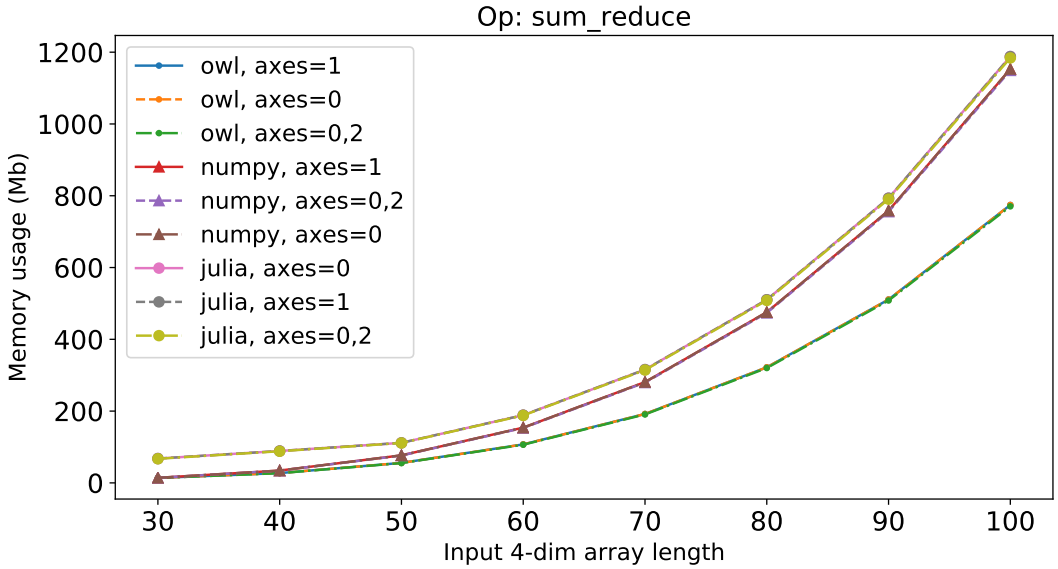


Figure 3.4: Sum reduction operation memory usage.

using SIMD for each operation would perform better, but that would require more complex logic to decide the execution hardware and software environment, and the code structure would be less generic.

3.2.2 Reduction Operations

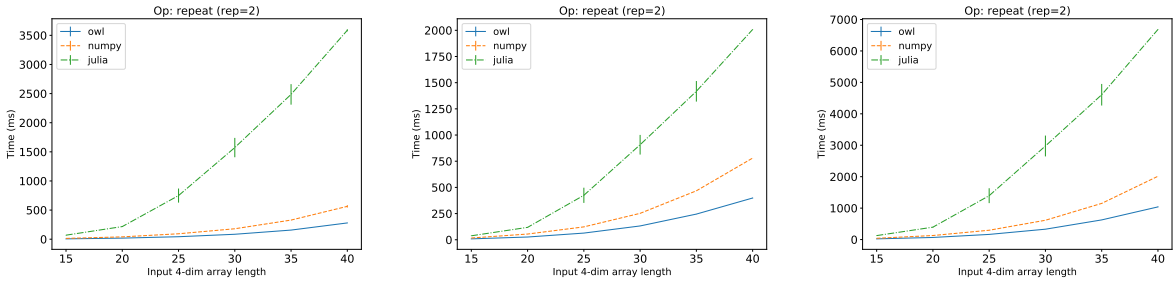
As in the parallel programming model, the map operations are accompanied by another group: the `reduction` operations, or the "fold" operations as they are sometimes called. Reduction operations such as `sum` and `max` accumulate values in an ndarray along certain axes by certain functions. For example, a 1-dimension ndarray (vector) can be reduced to one single number along the row dimension. The result can be the sum of all the elements if the "sum" operation is used, or the max of these elements if it is the "max" operation.

The reduction operations are among the key operation that are key to high level applications. For example, `sum` is used for implementing the BatchNormalisation neuron, which is a frequently used neuron in DNN.

Apparently, the fold operations follow similar pattern, and that leads to the similar design choice as the map operations using templates. The implementation of the reduction operations are summarised into several patterns, which are contained in the corresponding header file as templates. In most of the cases of these templates I only need to define the accumulation function `ACCFN`. Same with the map functions, these macros are defined in the stub file `owl_ndarray_maths_stub.c`. For example, for the sum function of float precision, I define the accumulation function as `#define ACCFN(A,X) A += X`.

The reduction operation often needs a specified axis. One challenge we were faced with is the multi-axis reduction. A naive implementation is to repeat the operation along one axis for each axis specified, and then repeat this procedure on the next axis. However, each single-axis reduction needs extra temporary memory for storing the intermediate result. In applications that heavily utilises the reduction operation such as a DNN, the inefficiency of reduction operations becomes a memory and performance bottleneck.

In a single-axis reduction algorithm, it needs to reduce source ndarray `x` into a smaller destination ndarray `y`. Suppose the dimension to be reduced is of size a, and total number of elements in `x` is n. Then the basic idea is to iterate their elements one by one, but the index in `y` keeps returning to 0 when it reaches $a/n - 1$. I revise this process so that the



(a) Repeat operation speed on rPi3 (b) Repeat operation speed on rPi4 (c) Repeat operation speed on Cubietruck

Figure 3.5: Repeat operation performance comparison.

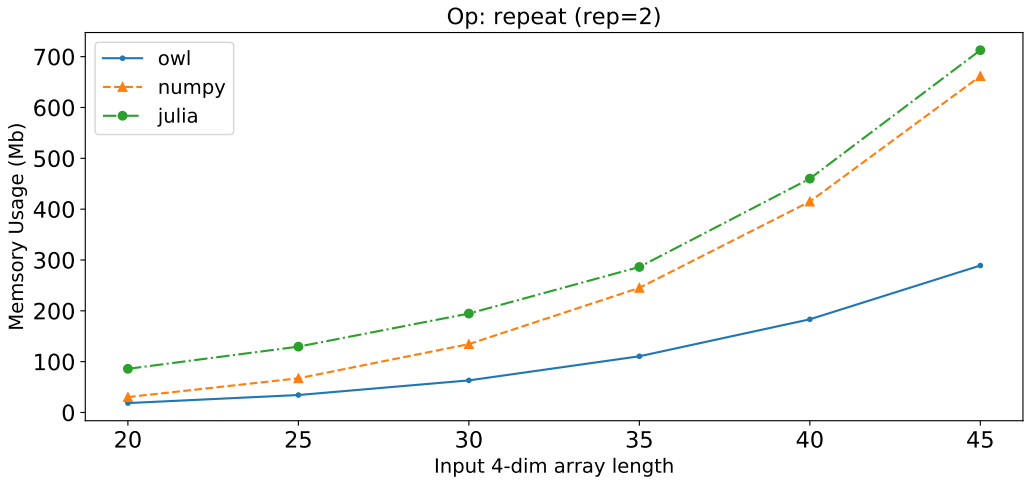


Figure 3.6: Repeat operation memory usage comparison.

index in `y` can keep the re-iterating according to given axes, all using one single piece of intermediate memory.

Since it involves multiple axes, to evaluate the reduction operation, I use a four-dimensional ndarray of float numbers as input. All four dimensions are of the same length. I measure the peak memory usage with increasing length, each for axis equals to 0, 1, and both 0 and 2 dimension. The evaluation result compared with NumPy and Julia is shown in Fig. 3.4. It shows that the proposed algorithm achieves lower memory usage than both NumPy and Julia.

3.2.3 Repeat Operations

The `repeat` operation repeats elements of an ndarray along each axis for specified times. For example, a vector of shape `[2,3]` can be expanded to shape `[4,3]` if repeated along the first axis, or `[2,6]` along the second axis. It consists of inner repeat and outer repeat (or `tile`). The former repeats elements of an input ndarray, while the later constructs an ndarray by repeating the whole input ndarray by specified number of times along each axis.

`Repeat` is another operation that is frequently used in DNN, especially for implementing the Upsampling and BatchNormalisation [102] neurons. While a reduction operation "shrinks" the input ndarray, a repeat operations expands it. Both operations require memory management instead of complex computation. Each repeat along one axis requires

creating extra memory space for intermediate result. Therefore, similar to the reduction functions, to perform multi-axis repeat, simply using existing operations multiple times leads to memory bottleneck for the whole application.

To this end, I implement the multi-axis repeat operation in Owl. The optimisation I use in the algorithm follows two patterns. The first is to provide multiple implementations for different inputs. For example, if only one axis is used or only the highest dimension is repeated, a specific implementation for that case would be much faster than a general solution. The second is to reduce creating intermediate memory. A repeat algorithm is like a reverse of reduction: it needs expand the source ndarray `x` into a larger destination ndarray `y`. Using the elements to be repeated as a block, the repeat operation copies elements from `x` to `y` block by block. The indices in both ndarrays move by a step of block size, though at different cycles. In the revised implementation, the intermediate memory is only created once and the all the iteration cycles along different axes are finished within the same piece of memory.

The evaluation of `repeat` is similar to that of reduction operations. I use a four-dimensional ndarray of float numbers as input. All four dimensions are of the same length. I measure the speed for increasing length, the repetition times is set to 2 on all dimensions. The evaluation results compared with NumPy and Julia on different devices are shown in Fig. 3.5. In all these cases, the repeat operation in Owl outperforms that in the other two libraries. I also measure the peak memory usage in Fig. 3.6a. As can be seen, the repeat operation achieves about half of that in NumPy with regard to both execution speed and memory usage.

Compared to this implementation, the multi-axis repeat operation in NumPy is achieved by running multiple single-axis repeat, and thus is less efficient in both memory usage and execution time. The repeat operation in Julia is much slower than the other two. One reason is that this operation is implemented in pure Julia rather than the efficient C code. Another reason is that `repeat` is not a computation-intensive operation, so the optimisation techniques such as static compilation and vectorisation are of less importance than algorithm design.

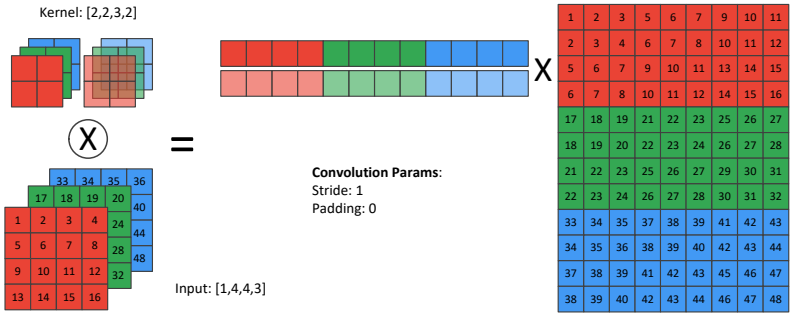
3.2.4 Convolution Operations

Convolution operations takes up the majority of computation involved in deep neural networks. A convolution operation takes two ndarrays as input: image (I) and kernel (F). In a 2-dimensional convolution, both ndarrays are of four dimensions. The image ndarray has B batches, each image has size $H \times W$, and has IC channels. The kernel ndarray has R rows, C columns, the same input channel IC, and output channel K. The convolution can then be expressed as in Eq. 3.1.

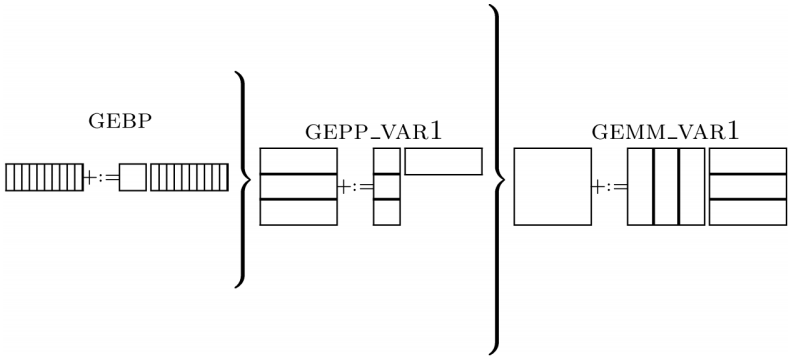
$$CONV_{b,h,w,k} = \sum_{ic=1}^{IC} \sum_{r=1}^{R} \sum_{c=1}^{C} I_{b,h+r,w+c,ic} F_{r,c,ic,k}. \tag{3.1}$$

The convolution operation is first implemented in Owl by interfacing to the Eigen library, which is also used in TensorFlow for CPU convolution implementation. However, interfacing to this C++ library proves to be problematic and leads to a lot of installation issues. Therefore I decide to use C to implement convolutions, which consists of three types: `Conv`, `ConvBackwardKernel`, `ConvBackwardInput`.

The `Conv` operation calculates the output given input image and kernel. Similarly, `ConvBackwardKernel` calculates the kernel given the input and output ndarrays, and `ConvBackwardInput` gets input ndarray from kernel and output. The last two are mainly



(a) `im2col` algorithm [103]



(b) Divide matrix multiplication into blocks [104]

Figure 3.7: Convolution algorithm illustration.

used in the backpropagation phase in training a DNN, but all three operations share a similar calculation algorithm.

A naive convolution algorithm is to implement Eq. 3.1 with nested for-loops. It is easy to see that this approach does not benefit from any parallelisation, and thus is not suitable for production code.

The next version of implementation uses the `im2col` algorithm [105]. This algorithm is illustrated in Fig. 3.7a. In this example, we start with an input image of shape 4x4 and 3 output channels. Each channel is denoted by a different colour. Besides, the index of each element is also show in the figure. The kernel is of shape 2x2, has 3 input channels as the input image. Each channel has the same colour as the corresponding channel of input image. The 2 output channels are differentiated by various level of transparency in the figure. According to the definition of convolution operation, we use the kernel to slide over the input image step by step, and at each position, an element-wise multiplication is applied. Here in this example, we use a stride of 1, and a valid padding. In the first step, the kernel starts with the position where the element indices are `[1,2,5,6]` in the first input channel, `[17,18,21,22]` in the second input channel, and `[33,34,37,38]` in the third input channel. The element-wise multiplication result is filled into corresponding position in the output ndarray. Moving on to the second position, the input indices become `[2,3,6,7,18,19,22,23,34,35,38,39]`. So on and so forth. It turns out that this process can be simplified as one matrix multiplication. The first matrix is just the flattened kernel. The second matrix is based on the input ndarray. Each column is a flattened sub-block of the same size as one channel of the kernel. This approach is the basic idea of the `im2col` algorithm. Since the matrix multiplication is a highly optimised operation in linear algebra packages such as OpenBLAS, this algorithm can be executed efficiently.

However, this algorithm requires generating a large temporary intermediate matrix. Depending on input image size, this matrix can take Gigabytes of memory in applications

such as FST. If you look closely at the intermediate matrix, you will find that it contains a lot of redundant information. Algorithms such as Memory-efficient Convolution [106] aims to reduce the size of this intermediate matrix, but still fail with large input or kernel sizes.

To reduce the memory usage, I apply the method proposed in [104], which is to cut matrices into small blocks so as to fit into the L1/L2 cache of CPU to do high-performance computation while reducing the memory usage, regardless of input size. Fig. 3.7b shows how multiplication of two matrices can be divided into multiplication of small blocks. It still generally follows the previous matrix multiplication approach, but instead of generating the whole intermediate matrix, it cuts the input and kernel matrices into small blocks one at a time so that the memory usage is limited no matter how large the input and kernel are. I implement the method proposed in [104] to calculate suitable block size based on the cache size of the CPU.

To further improve the performance, I use the SIMD intrinsics in filling the temporary matrix from input ndarray. For one thing, depending on the input channel is divisible by the supported data length of SIMD (e.g. 8 float numbers for AVX), I provide two set of implementations for filling the temporary blocks. During loading data from input ndarrays to these matrix blocks, I also use AVX intrinsics such as `_mm256_load_ps` to improve performance. Finally, the matrix multiplication between two small matrix blocks is implemented by the routines in OpenBLAS.

To maximise the performance of caching, I need to make the memory access as consecutive as possible. Depending on whether the input channel is divisible by the supported data length of SIMD (e.g. 8 float numbers for AVX), I provide two set of implementations for filling the temporary blocks. If input channel is divisible by data length, the input matrix can always be loaded consecutively at a step of data length with the AVX intrinsics, otherwise I have to build the temporary matrix blocks with less AVX intrinsics, on only part of the matrix, and then take care of the edge cases.

One more optimisation is that, I have shown the `im2col` method and its disadvantage with memory usage. However, it is still straightforward and fast with small input sizes. Therefore, suitable implementations can be chosen depending on the input size.

The convolution operations consist of three basic types: convolution, convolution backward kernel, and convolution backward input. All are frequently used in the deep neural network applications. The `Conv` operation calculates the output given input image and kernel. Similarly, `ConvBackwardKernel` calculates the kernel given the input and output ndarrays, and `ConvBackwardInput` gets input ndarray from kernel and output. The last two are mainly used in the backpropagation phase in training a DNN, but all three operations share a similar calculation algorithm. The implementation of backward convolutions can also be thought of as matrix multiplication. For `ConvBackwardKernel`, it first reshapes the output ndarray as matrix, and multiplies it with the intermediate input matrix. Similarly, in `ConvBackwardInput`, we need to first multiply the kernel and output matrix to get the intermediate input matrix, and then re-construct the input ndarray based on it.

These implementations can then be easily extended to the three dimension and one dimension cases. Besides, the transpose convolutions and diluted convolutions are only variate of normal convolution and the code only needs to be slightly changed. Above this C implementation level, mutable convolution operations are also provided, so as to further improve performance by utilising existing memory space.

The evaluation method is generally the same as before. I use ndarray of different sizes as input, and compare it to the state of the art Eigen library. One difference is that, convolution operations take two ndarrays in the calculation: the input and the kernel.

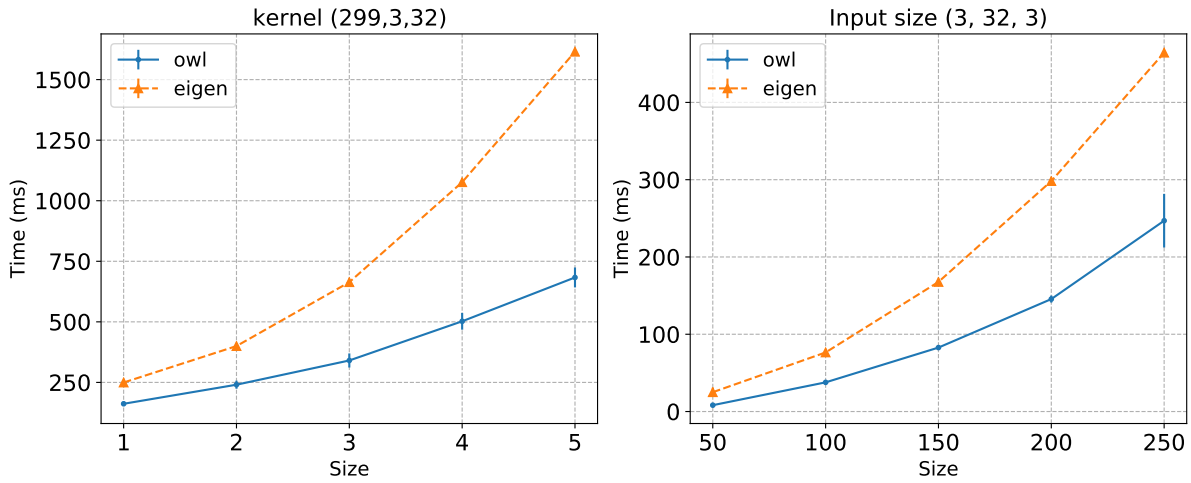


Figure 3.8: Compare the execution time of Conv2D operation of Owl and Eigen.

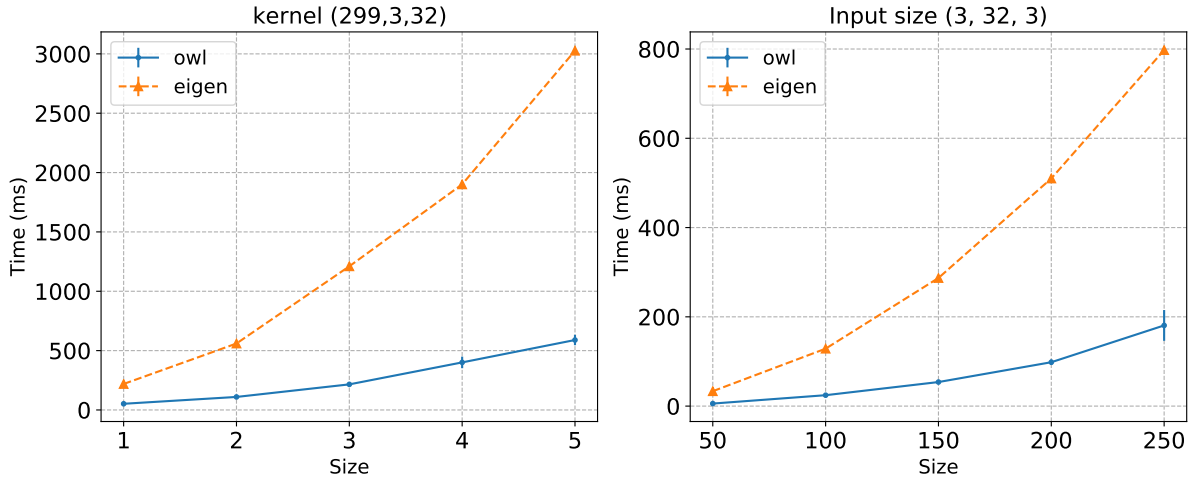


Figure 3.9: Compare the execution time of Conv2D Backward Kernel operation of Owl and Eigen.

Therefore, considering these two variables, I use two sets of evaluation set-up: fixed input size with varying kernel size; and fixed kernel size with varying input size.

The other difference is that, since NumPy and Julia do not provide the convolution operations, I choose to compare it to the Eigen library. I make this choice because Eigen is a widely used library to support high-performance convolution operations, and is used as computation backend of TensorFlow on CPU devices. Eigen is a C++ library. To make sure the comparison is fair, instead of directly comparing Owl code with C++ code, I change the implementation in Owl and interface existing operations to those in the Eigen library[4].

I compare three convolution operations in the evaluation. The evaluations shown in Fig. 3.8a, 3.9, and 3.10 are performed on the single board computer Cubietruck, and the evaluation conducted on other boards such as the Raspberry Pis generate similar results. They show the effectiveness of my implementation of convolution operations compared with that of the Eigen library. This good performance comes from the combination of multiple optimisation techniques as well as choosing suitable implementation according to the input.

[4]The interfacing library from OCaml to Eigen is provided in `https://github.com/owlbarn/eigen`.

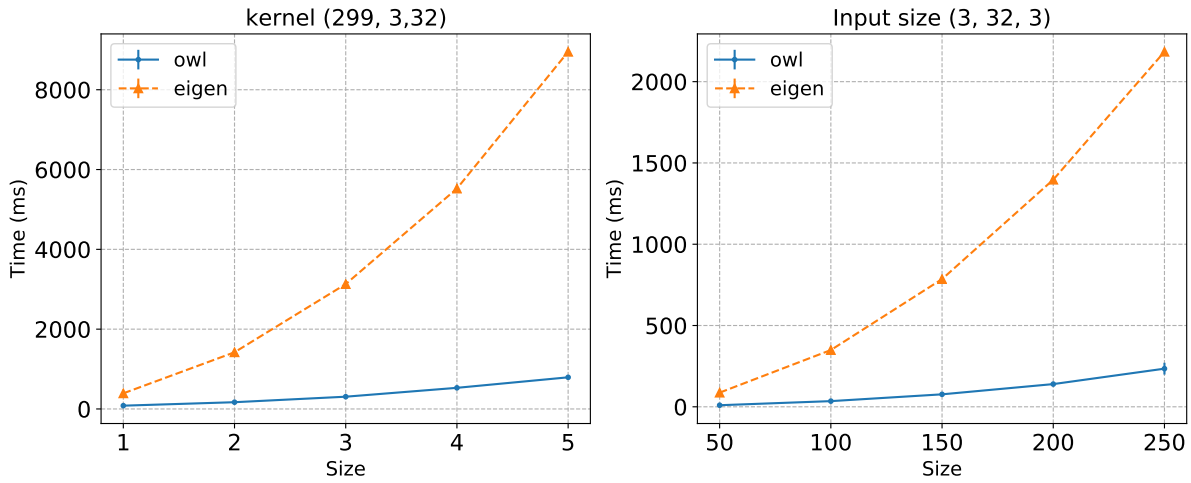


Figure 3.10: Compare the execution time of Conv2D Backward Input operation of Owl and Eigen.

3.3 Automatic Parameter Tuning

Recent research work on parameter tuning mostly focuses on hyper-parameter tuning, such as optimising the parameters of stochastic gradient in machine learning applications. However, tuning code and parameters in low-level numerical libraries is of the same importance. For example, ATLAS [16] and the recent Intel Math Kernel Library (MKL) are both software libraries of optimised math routines for science and engineering computation. They are widely used in many popular high-level platforms such as Matlab and TensorFlow. One of the reasons these libraries can provide optimal performance is that they have adopted the paradigm of Automated Empirical Optimisation of Software, or AEOS. That is, a library chooses the best method and parameter to use on a given platform to do a required operation. One highly optimised routine may run much faster than a naively coded one. Optimised code is usually platform- and hardware-specific, so an optimised routine on one machine could perform badly on the other.

Though Owl currently does not plan to improve the low-level libraries it depends on, as an initial endeavour to apply the AEOS paradigm in Owl, one ideal tuning point is the parameters of OpenMP used in Owl.

Currently manys computers contain shared memory multiprocessors. OpenMP is used in key operations in libraries such as Eigen and MKL. Owl has also utilised OpenMP on many mathematical operations to boost their performance by threading calculation. For example, Fig. 3.11 shows that when I apply the sine function on an ndarray in Owl, on a 4-core CPU MacBook, the OpenMP version only takes about a third of the execution time compared with the non-OpenMP version.

However, performance improvement does not come for free. The overhead of using OpenMP comes from time spent on scheduling chunks of work to each thread, managing locks on critical sections, and startup time of creating threads, etc. Therefore, when the input ndarray is small enough, these overheads might overtake the benefit of threading.

What is a suitable input size to use OpenMP then? This question would be easy to solve if there is one single suitable input size threshold for every operation, but that is not the case. In a small experiment, I compare the performance of two operations, `abs` (calculate absolute value) and `sin`, in three cases: running them without using OpenMP, with 2 threads OpenMP, and with 4 threads OpenMP.

The result in Fig. 3.12 shows that, with growing input size, for the sine operation, the

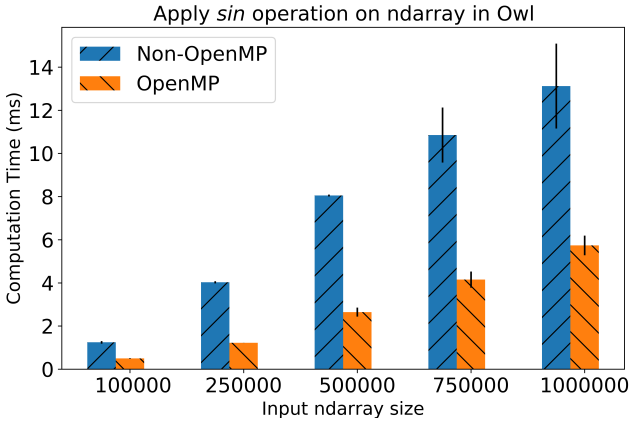


Figure 3.11: Parallel execution of the `sin` operation on ndarray using OpenMP.

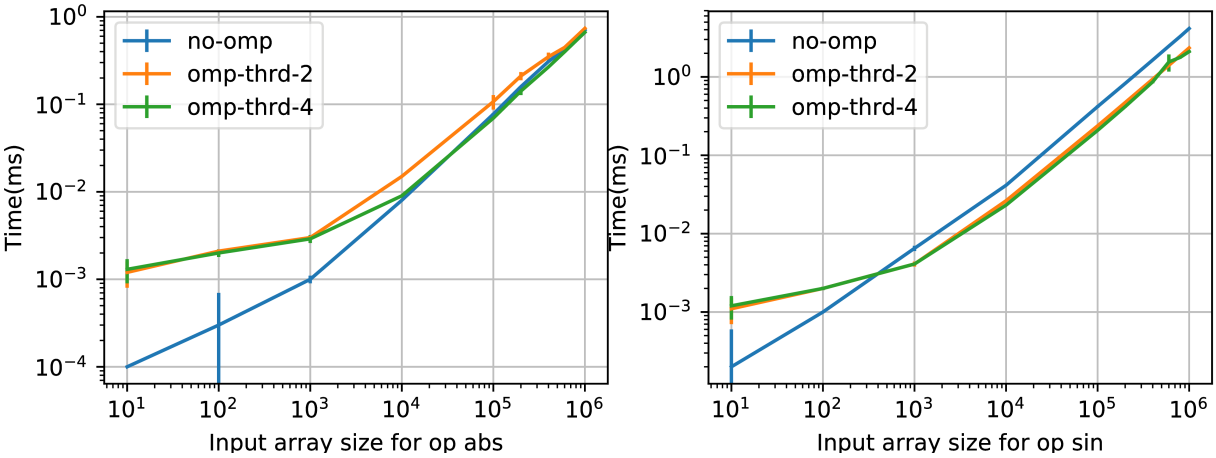


Figure 3.12: Compare the behaviour of `abs` and `sin`when using OpenMP.

OpenMP version outperforms the non-OpenMP version at a size of less than 1000, but for `abs` operation, that cross point is at about 1,000,000. The complexity of math operations varies greatly, and the difference is even starker when we compare their performance on different machines. Note that both axes use log-scale, and that is why a small deviation when the input array size is small looks large in the figure.

This issue becomes more complex when considered in real applications such as DNN, where one needs to deal with operations of vastly different complexity and input sizes. Thus one fixed threshold for several operations is not an ideal solution. Considering these factors, I need a fine-grained method to decide a suitable OpenMP threshold for each operation.

3.3.1 Implementation

Towards this end, I implement the AEOS module in Owl. The idea is to add a tuning phase before compiling and installing Owl, so that each operation learns a suitable threshold parameter to decide if OpenMP should be used or not, depending on input size.

The key idea of parameter tuning is simple. I implement two versions of each operation, one using OpenMP and the other not. I then measure their execution time for various sizes of input. Each measurement is repeated multiple times, and, to reduce the effect of outliers, only the values that are within first and third quartiles are used. After removing outliers, regression is performed to find a suitable input size threshold. According to our initial experiment, linear regression is fit to estimate the OpenMP parameters here.

```
module Sin = struct
(** Tuner type definition. *)
type t = {
    mutable name  : string;
    mutable param : string;
    mutable value : int;
    mutable input : int array array;
    mutable y     : float array
}
val make : unit -> t (** Create the tuner. *)
val tune : t -> unit (** Tuning process. *)
val save_data : t -> unit
(** Save tuned data to csv file for later analysis. *)
val to_string : t -> string
(** Convert the tuned parameter(s) to string to be written on file *)
end
```

Listing 2: Implementation of operations in AEOS module.

Since this tuning phase is executed before compiling Owl, the AEOS module is independent of Owl, and all necessary implementation is coded separately to ensure that future changes of Owl do not affect the AEOS module itself.

The tuned parameters then need to be passed to Owl. When the OpenMP switch is turned on, the AEOS module generates a C header file which contains the definition of macros, each of which defines a threshold for one operation. When this header file is not generated, pre-defined default macro values are used instead. After that, Owl is compiled with this header file and uses these tuned parameters in its math operations. The tuning phase only needs to be performed once on each machine during installation.

The design of the AEOS module focuses on keeping tuning simple, effective, and flexible. Each operation is implemented as a single OCaml module, so that support for new operations can be easily added. The interface of such a module is shown as in Listing 2. I expect that tuning does not have to be only about OpenMP parameters, and that different regression methods could be used in the future. For example, the TheilSen estimator can be plugged in for parameter estimation if necessary. In each module, arbitrary tuning procedures can be plugged in as long as the interface is satisfied.

The AEOS module is implemented in such a way that brings little interference to the main Owl library. Code can be viewed in this pull request, and has been merged into the main branch of Owl. You only need to switch the ENABLE_OPENMP flag from 0 to 1 in the dune file to try this feature.

To evaluate the performance of tuned OpenMP thresholds, I need a metric to compare them. One metric to compare two thresholds is proposed as below. I generate a series of ndarrays, whose sizes grow by certain steps until they reach a given maximum number, e.g. 1,000,000 used in the experiment below. Note that only input sizes that fall between these two thresholds are chosen to be used. I then calculate the performance improvement ratio of the OpenMP version function over the non-OpenMP version on these chosen ndarrays. The ratios are added up, and then amortised by the total number of ndarrays. Hereafter I use this averaged ratio as performance metric.

Table 3.2 presents the tuned threshold values of a five operations on a MacBook with a 1.1GHz Intel Core m3 CPU and a Raspberry Pi 3B. I can see that they vary across different

Table 3.2: Tuned parameters using AEOS module.

Platform	tan	sqrt	sin	exp	sigmoid
Laptop	1632	max_int	1294	123	1880
Raspberry Pi	1189	209	41	0	0

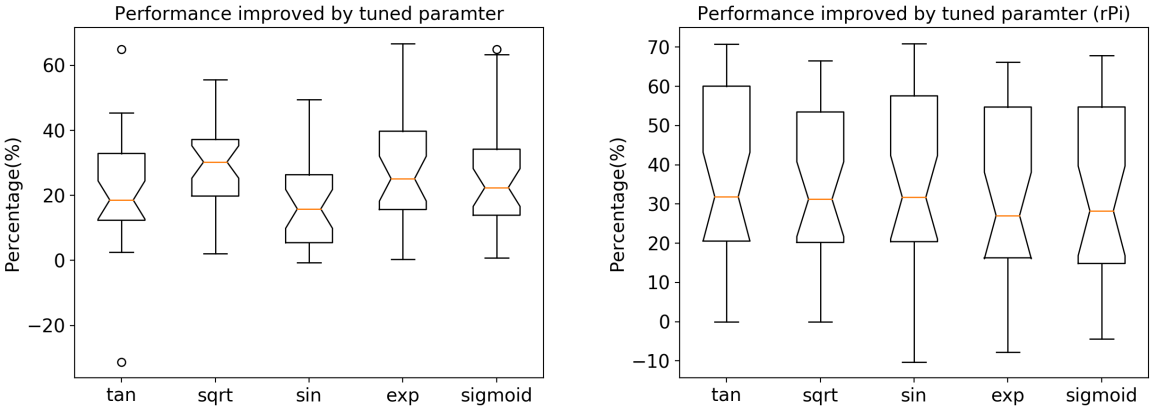


Figure 3.13: Improvement of math operations after applying parameter tuning.

operations and different machines, depending on their computational complexity. For example, on MacBook, the tuning result is"max_int", which means that for the relatively simple square root calculation OpenMP should not be used, but that is not the case on Raspberry Pi. Also, I note that the less powerful Raspberry Pi tends to get lower thresholds.

I then evaluate the performance improvement after applying AEOS. I compare each generated parameter with 30 randomly generated thresholds. These measured average ratios are then presented as a box plot, as shown in the Fig. 3.13.

It can be observed that, in general, more than 20% average performance improvement can be expected on the MacBook. The result on Raspberry Pi shows a larger deviation but also a higher performance gain (about 30% on average). One reason for this difference is that a suitable threshold on Raspberry Pi tends to be smaller, leading to a larger probability to outperform a randomly generated value.

Comparing with the randomly chosen parameters may not be totally convincing. Therefore, I also compare the tuned thresholds with a series of regular thresholds. Specifically, for each operation, I choose ten different thresholds with fixed interval: 0, 100000, 200000... 900000. For each generated threshold, I use 100 numbers between 0 and 1E6 as ndarray sizes. They are also generated with a fixed interval. The execution time on the ndarrays of given sizes for each threshold are then compared with that of the tuned threshold, and the element-wise ratios between these two arrays can be plotted as a barplot for each threshold. For example, the comparison for the square root operation on the MacBook is shown in Fig. 3.14. Here each bar indicates the ratio between the tuned and the chosen threshold. 100 percent means these two are of the same effect on performance, and lower percentage means the tuned threshold leads to faster execution time. This figure shows that regardless of the choice of fixed thresholds, the tuned parameter can always lead to similar or better execution time of operations in the AEOS module.

Note that I cannot claim that the tuned parameters are always optimal, since the figure shows that in some rare cases where the improvement percentages are negative, the randomly found values indeed perform better. Also, the result seems to suggest that

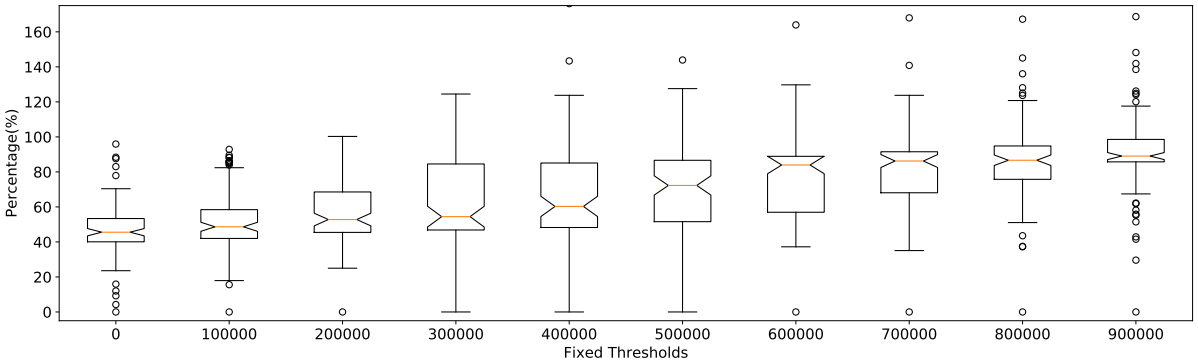


Figure 3.14: Improvement of the square root operation after applying parameter tuning.

AEOS can provide a certain bound, albeit a loose one, on the performance improvement, regardless of the type of operation. These interesting issues require further investigation.

3.4 Conclusion

In this chapter, I showed the connection of advanced application and basic operations in a numerical library. Then I used several representative operations to demonstrate the low level design of the library and related optimisation. Finally, based on the possible optimisation space, I presented an automatic parameter tuning module in the library so as to provide optimal performance on different machines.

The optimisation, however, does not stop at separate operations. In the next chapter, I will show the abstraction of a whole computation, which consists of multiple operations, and the possible related optimisation at a higher level.

Chapter 4

Computation Graph as Intermediate Representation

In the last chapter, I have shown the optimisation of basic operations in the numerical library and how they affect the performance of advanced applications such as DNN. But the optimisation is not limited at operations. In this chapter, I will show the abstraction of a whole computation process, the *computation graph* (CGraph), which consists of multiple operations. This provides optimisation opportunities at a higher level.

4.1 Graph as Intermediate Representation

A computation graph is a way to represent a function in the form of graph. In a CGraph, nodes are either input values or operations for processing values. A node's incoming and outgoing links are its input and output. A CGraph is used as core abstraction of computation in current systems. For example, TensorFlow uses graph to represent its computation, with support of nearly a thousand operations. Owl also provides support for CGraph in the form of a stack of functors.

The CGraph module in Owl brings in a lot of benefits. For example, it enables graph structure and memory optimisation, since the graph structure is fixed and the input shapes are known. One optimisation is reusing previously allocated memory, which is especially useful for those applications involving large ndarray calculations. In fact, this optimisation can also be performed by a compiler by tracking the reference number of allocated memory, a technique referred to as linear types. Besides, the computation graph provides a way to abstract the flow of computations. Therefore it is able to bridge the high-level applications and low-level machinery of various hardware devices. This is why it has natural support for heterogeneous computing. A computation graph can be decomposed into multiple independent subgraphs and each can be evaluated in parallel on different cores or even computers. Maintaining the graph structure also improves fault-tolerance, by providing natural support for rollback mechanisms.

Due to these optimisations, the CGraph has a profound implication to the Owl library. Because the memory allocated for each node is mutable, Algorithmic Differentiation becomes more scalable when evaluating large and complex graphs. At the same time, mutable transformation is handled by Owl so programmers can still write safe functional code. More importantly, with the CGraph module, I find a good chance to export the computation on Owl to other platforms.

One issue that is not well investigated yet in Owl is the support for hardware accelerators such as GPU/TPU. The previous effort using OpenCL requires a lot of engineering effort, and is also non-flexible across different hardware platforms. As shown in the back-

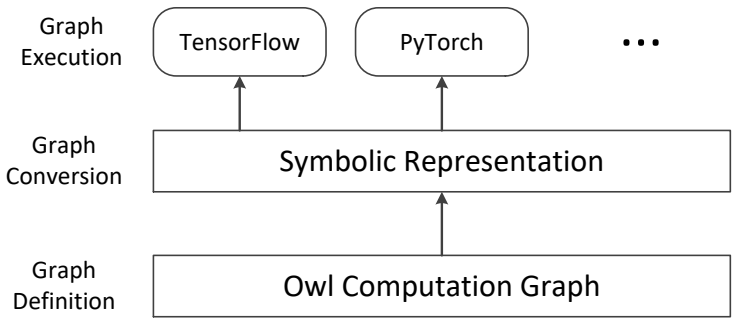


Figure 4.1: Using an symbolic intermediate representation to export computation in Owl to multiple platforms.

ground work, there is a growing trend to separate out the definition of computation, and let the low level compilers to deal with optimisation and code generation etc. to pursue best computation performance. Besides, tasks such as visualising a computation also require some form of intermediate representation (IR).

Motivated by these factors, I explore using CGraph as an IR to transplant computation in Owl across different numerical frameworks and hardware platforms. The target is to have the best of both worlds. On one hand, I can define "how to compute" on Owl with its elegant and powerful syntax; on the other hand, I can execute the computation efficiently across various hardware devices, such as GPU and TPU, that other frameworks support.

As to the choice of this IR, the neural network compilers such as TVM [64] and Glow [66] focus on improving backend computation performance optimisation instead of computation graph interoperability. Also an empirical reason is that back when I started, the trend of computation definition/execution separation was not clear and these work are either not available or in initial status. Towards this end, I choose to develop the IR based on the computation graph of TensorFlow. Compared to the recently proposed neural network standard such as NNEF [24], TensorFlow is a widely accepted industrial standard and supports nearly a thousand operations.

The general workflow of our approach is shown in Fig. 4.1. I focus on building a symbolic intermediate representation layer. It takes the computation graph in Owl as input, and constructs a symbolic graph that can further be executed on various platforms. As stated above, in my work TFgraph, I build this IR based on the TensorFlow computation graph. The graph can be further executed on GPU/TPU devices, or be converted to run on other platforms such as PyTorch.

As to how to perform the conversion, since TensorFlow's Python library interfaces to its C++ implementation, one possible way is to provide similar interfaces for Owl. However, it may need some significant engineering effort. I instead utilise the "Save and Restore" mechanism in TensorFlow. It provides methods for loading CGraph definition from a metagraph file. All I need to do now is to generate this metagraph file from Owl CGraph. Towards this end, I build an experimental system *TFgraph*. It aims to export CGraph defined in Owl and execute it in TensorFlow. Its workflow is:

1. define a CGraph in Owl;
2. convert this graph into a `tensorflow_cgraph` in converter (explained later);
3. parse this `tensorflow_cgraph` into string format and write to a `.pbtxt` file;
4. load the `.pbtxt` file into TensorFlow using its Save/Restore mechanism.

Note that currently I focus on the case of executing one CGraph once, instead of iteratively re-evaluating it (as in the case of DNN training). Also, execution of computation

```
type tensorflow_cgraph = {
  mutable tfmeta  : tfmeta;   (* MetaInfoDef *)
  mutable tfgraph : tfgraph;  (* GraphDef *)
  mutable tfsaver : tfsaver;  (* SaverDef *)
  mutable tfcolls : tfcolls;  (* CollectionDef *)
}
```

Listing 3: Core data structure in TFgraph to represent the CGraph in TensorFlow.

involves two aspects: computation graph, and data. In the TFgraph system, I do not consider passing data from Owl to TensorFlow.

4.2 Design

In designing the system, I start with the abstraction of TensorFlow CGraph. Unlike Owl CGraph, which is almost a plain graph structure, a TensorFlow CGraph contains more meta information, which consists of four parts, as shown in Listing 3.

The `MetaInforDef` contains operations used in a CGraph, and meta information such as version number. The `SaverDef` specifies checkpoint file name, which operation to run when saving and loading variables, and the maximum number of checkpoints to keep, etc. The `CollectionInfo` is for collecting certain nodes and variables in the graph.

The core component of the whole graph lies in `GraphDef`. It is an array of TensorFlow nodes; each has its own attributes. In TFgraph, the core part is a proper representation of TensorFlow nodes. Each node is defined in a stand-alone module that contains the required properties and methods. Listing 4 shows the `TFSin` module that represents the sine operation.

The node module contains several properties: `name` is a node id that is unique in the whole graph; `op_name` shows the type of this node (e.g. "Sin" in this case); `inputs` is the list of id of input nodes to the current node; `out_shp` specifies the shape of the node's output; `dtype` is the type of data that the node accepts, such as float or double; `device` is a string that specifies the device location of the current node; `cls` contains information such as co-location in device partition. These proprieties are similar across different operations but may vary. A node module is constructed using the `create` method. Information of these proprieties can be retrieved from Owl CGraph.

A node in `GraphDef` is generated using `make_nodedef` method. Based on existing properties in a module, it creates an array of attributes that conforms to the specification of TensorFlow[1]. The attribute value is defined as in Listing 5. A node in the graph consists of attributes, together with other information: name, operation name, input nodes and device placement.

Above this layer, I specify the rules about how each Owl node should be mapped to TensorFlow node(s). Given a Owl CGraph, the converter traverses the whole graph and maps the Owl nodes one by one according to those rules. Finally, TFgraph iterates a computation graph in a Owl CGraph, and generates a list of nodes as defined before, as well as information that the other three components requires. Different serialisation methods can then be applied on the `tensorflow_cgrap` structure. Currently I provide the `to_pbtxt` method, since output of the protobuf format is required in the graph conversion

[1]The definition of attribute values in TensorFlow can be found in its source code in the format of protobuf: `core/framework/attr_value.proto`

```
module TFSin = struct
  type t = {
    mutable name    : string;
    mutable op_name : string;
    mutable inputs  : string array;
    mutable out_shp : int array;
    mutable dtype   : string;
    mutable device  : string;
    mutable cls     : string array;
  }

  let opname = "Sin"

  let create ?(cls=[||]) ?(device="") name inputs out_shp = {
    name = name; op_name = opname; inputs = inputs; cls = cls;
    dtype = "DT_FLOAT"; device = device; out_shp = out_shp;
  }

  let make_nodedef n =
    let node_attr = [|
      ("T", (ATTR_Type n.dtype));
      ("_output_shapes", (ATTR_List [|(ATTR_Shape n.out_shp)|]))
    |] in
    let cls_attr = Array.map (fun c ->
            ATTR_String ("loc:@" ^ c)) n.cls in
    let node_attr = if (cls_attr = [||]) then node_attr else
      (Array.append node_attr [| ("_class", ATTR_List cls_attr) |])
    in
    { name = n.name; op_name = opname; input = n.inputs;
      node_attr = node_attr; device = n.device }
end
```

Listing 4: Node module for sine operation in TFgraph.

workflow.

4.3 Implementation

In the last section I introduced the design of the TFgraph system. In this section, I will discuss some implementation details.

Mapping of nodes The mapping between Owl CGraph nodes and TensorFlow nodes is not always straightforward. For many mathematical operations such as `sin` and `mul`, a one-to-one projection suffices. But there are also cases that multiple Owl operations map to one TensorFlow operation. For example, Owl provides `Conv2d`, `TransposeConv2d`, and `DilatedConv2d` to represent different types of 2D convolution operation, while in TensorFlow they are combined into one. A one-to-many mapping is also possible. A variable in TensorFlow contains different nodes such as initialisers, and one node can also

```
type tftensor = {
  dtype          : string;
  tensor_shape   : int array;
  string_val     : string array option;
  float_val      : float array option;
  int_val        : int array option;
  tensor_content : bytes option;
}

type tfattrvalue =
  | ATTR_Nil
  | ATTR_Int      of int
  | ATTR_Bool     of bool
  | ATTR_Type     of string
  | ATTR_Float    of float
  | ATTR_Shape    of int array
  | ATTR_String   of string
  | ATTR_Tensor   of tftensor
  | ATTR_List     of tfattrvalue array
  | ATTR_Namelist of {name : string; attr: (string * tfattrvalue) array}
```

Listing 5: Definition of a node's attribution in TFgraph.

contain multiple constant nodes as parameters. Moreover, in the case of save and restore mechanism, all the related nodes are only used in TensorFlow CGraph, not in Owl. I find these mapping types are common to be found in the system.

Naming I apply a simple naming rule for each node. For Owl's node, they are each assigned a unique id. If not specifically named by user, they are named in the format of `owlnode+id`. During the Owl-to-TensorFlow node mapping, if it is a one-to-one mapping, the name stays unchanged, otherwise the new nodes will be named in the format of `owlname+id/subnode_type_and_id`.

Serialisation In the `CollectionDef`, the variables need to be serialised and saved for access in the graph. The serialisation uses protobuf format[2]. This step is performed with the help of Python scripts that are generated by protocol buffer compiler.

Device placement There could be multiple computing devices on a machine. Device placement of nodes relates closely with computation performance. It is a topic that is still being investigated. In the implementation, each node is assigned a string such as `/job:worker/replica:0/task:1/device:GPU:3` to specify which device should it be executed on. In TFgraph, each node can be assigned a "device" property, or it can be left blank, in which case the default placement strategy of TensorFlow is applied.

[2]Specified in the source code of TensorFlow: `core/framework/variable.proto`.

```
module N = Dense.Ndarray.S
module G = Owl_computation_cpu_engine.Make (N)
include Owl_algodiff_generic.Make (G)

let f x y =
  let weight = Mat.ones 3 3 in
  Maths.( (pack_flt 2.) * (x *@ weight + y) + (pack_flt 1.))

let x = G.var_arr "x" |> pack_arr
let y = G.var_elt "y" |> pack_elt
let z = f x y

let output = [| unpack_arr z |> G.arr_to_node |]
let input  = [|
  unpack_arr x |> G.arr_to_node;
  unpack_elt y |> G.elt_to_node
|]
let g = G.make_graph ~input ~output "example_graph"
```

Listing 6: Example of using TFgraph (Part I).

```
module T = Owl_converter.Make (G)

let pbtxt = T.(convert g |> to_pbtxt)
let _ = Owl_io.write_file "test_cgraph.pbtxt" pbtxt
```

Listing 7: Example of using TFgraph (Part II).

4.3.1 Examples

In this section, I show the workflow of the TFgraph system using an application that performs simple math operation on ndarray.

Suppose I want to construct such a computation: `f(x, y) = 2 * (x * W + y) + 1`, where `x` and `W` are matrices and `y` is a float number. I can construct the CGraph as shown in Listing 6. I first define a function `f`, then two input placeholders `x` and `y`. After getting the computing output `z`, I create a CGraph `g` by linking output and input nodes together.

To convert this graph into a `pbtxt` file, I simply use the converter with one line of code, as shown in Listing 7. It uses two APIs provided by the converter: `convert` creates a `tensorflow_cgraph`, and then `to_pbtxt` parses this graph into string format.

Then I turn to the Python script. The snippet in Listing 8 reads the generated `test_cgraph.pbtxt` file into a `MetaGraph` data structure, then serialises it to a protobuf binary file. This file can be loaded by the model saver of TensorFlow, as shown in Listing 9. After the graph is loaded into a TensorFlow session, users can get its inputs by names (assuming they are already known), get the output from the "result" collection, and then proceed to evaluation with `sess.run()`.

Similar procedures can be applied to more real world applications, such as the inference and training of DNN. While in an inference phase a forward graph is executed once, in the training phase, two graphs – a forward graph and a backward graph – are executed iteratively. But implementing a training phase of DNN needs manually scheduling, which

```
filename = 'test_cgraph'
with open(filename + '.pbtxt', 'r') as f:
    metagraph_def = tf.MetaGraphDef()
    file_content = f.read()
    text_format.Merge(file_content,metagraph_def)

    graph_io.write_graph(metagraph_def,
        os.path.dirname(filename),
        os.path.basename(filename) + '.pb',
        as_text=False)
```

Listing 8: Example of using TFgraph (Part III).

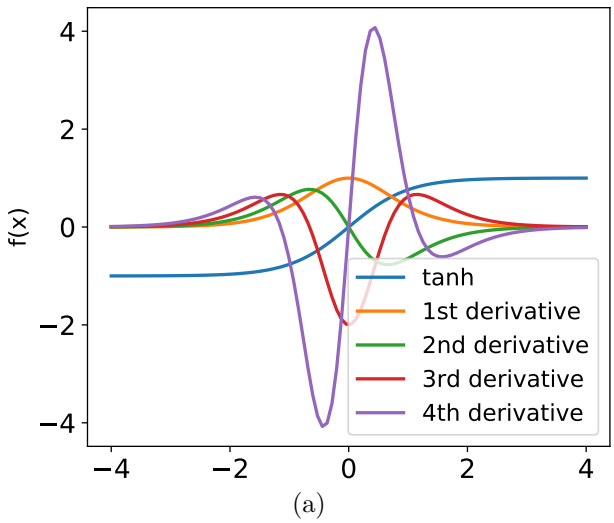
```
with tf.Graph().as_default():
    sess = tf.Session()
    saver = tf.train.import_meta_graph('test_cgraph.pb')
    graph = tf.get_default_graph()

    x = graph.get_tensor_by_name('x:0')
    y = graph.get_tensor_by_name('y:0')
    z = tf.get_collection("result")[0]

    init = tf.global_variables_initializer()
    sess.run(init)

    x_data = np.ones((3, 3))
    y_data = 2.
    result = sess.run(z, feed_dict={x:x_data, y:y_data})
```

Listing 9: Example of using TFgraph (Part IV).

requires non-trivial work from end users. Instead, using the TFgraph approach, changing from inference to training a DNN only requires adding several lines of code.

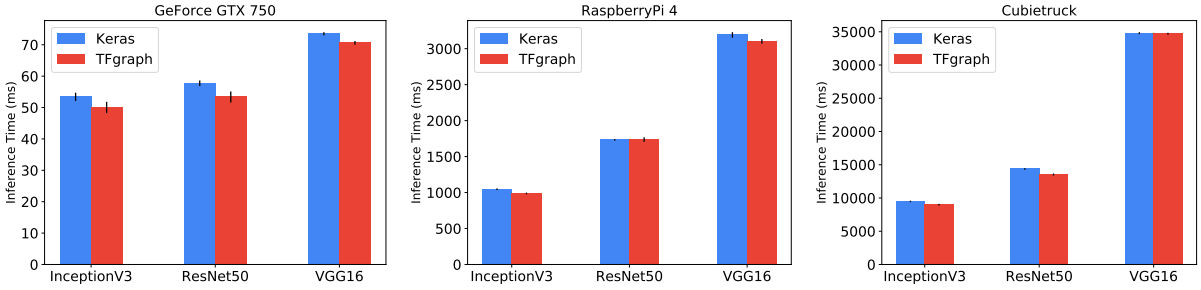
4.4 Evaluation

Expressiveness TFgraph is not limited to neural networks. One example I have created is about higher-order derivatives. It utilises Owl's support of algorithmic differentiation, which is also enabled by using CGraph as representation of computation. I first define a function `tanh`, and then construct the computation graph of the first to the fourth derivative by simply calling the `diff` function from Owl's Algorithmic Differentiation module. Each derivative can be seen as an output in constructing a CGraph. By using TFgraph, these derivative functions can be plotted in TensorFlow, as shown in Fig. 4.2a. This example shows the good integration between TFgraph and the existing powerful features in Owl, such as DNN support and the Algorithmic Differentiation module. Therefore, TFgraph provides a bridge to efficiently express fast prototypes in Owl and then execute it directly on accelerators.



(a)

Figure 4.2: Define higher-order derivatives in Owl, and execute it in TensorFlow.



(a) (b) (c)

Figure 4.3: Use Keras and TFgraph to execute inference on different DNN architectures in TensorFlow.

Performance The target of TFgraph is to enable a computation in Owl to be executed on accelerators such as GPU. One aspect I am going examine is the efficiency of graphs that are generated by TFgraph. Since Keras can also utilise TensorFlow as its computation engine, I use both Keras and TFgraph to execute inference in TensorFlow. For evaluation, I use three DNN architectures: InceptionV3, ResNet50, and VGG16, and measure the inference time on them. The input image is of shape `299x299`. I use three different machines for the evaluation. The first is a GPU machine. It has an Intel Core i7 870 CPU, at 2.93GHz, and also has a GeForce GTX 750 GPU. The other two are single board computers I have used in the previous section: the Raspberry Pi 4, and the Cubietruck. The TensorFlow version is 1.13, and the Owl version is 0.8.0. All subfigures in Fig. 4.3 show similar information, regardless of the difference of computing power or hardware infrastructure. For the same computation, TFgraph generates computation graph that can be executed at very similar speed compared to using existing popular machine learning frontend library Keras.

4.5 Conclusion

In this chapter, I introduced the graph as an abstraction to represent computation, how it is supported in libraries such as Owl and TensorFlow, and presented the TFgraph system that uses a computation graph as an intermediate representation to utilise hardware accelerators. Based on this idea, a computation often needs to be deployed to certain

devices to provide useful services. In the next chapter, I will introduce the questions in computation composition and deployment, and the related Zoo system to address these questions.

Currently the system is still in an initial development phase so there still remains a lot to do. For example, not all Owl CGraph nodes are supported. Besides, iteratively updating variables and passing data from Owl to TensorFlow is not yet considered. Also, operations such as condition and loop are not yet supported in Owl CGraph, but are nevertheless important. Moreover, TFgraph should cope with the existing neural network standards and the neural network compilers. These issues about the future development direction of TFgraph and require further thought.

Chapter 5

Computation Composition and Deployment

In the previous section I have introduced the graph as a structure to represent computation, how it is supported in libraries such as Owl and TensorFlow, and I have also presented a system that uses computation graph as intermediate representation to utilise hardware accelerators.

Just as a computation takes a single operation as a building block, in this chapter, I take a whole computation as a unit, and discuss the problem of computation composition and deployment in a numerical library.

5.1 Introduction

Recently, computation on edge and mobile devices has gained rapid growth, such as personal data analytics in the home [107], DNN application on a tiny stick [108], and semantic search and recommendation on web browser [109]. HUAWEI has identified speed and responsiveness of native AI processing on mobile devices as the key to a new era in smartphone innovation [110]. Many challenges arise when moving machine learning (ML) analytics from cloud to edge devices. One widely-discussed challenge is the limited computing power and working memory of edge devices.Personalising analytics models on different edge devices is also an interesting topic [111].

However, one problem is not yet well defined and investigated: model composition. Training a model often requires large datasets and rich computing resources, which are often not available to normal users. That is one of the reasons that they are bound to the models and services provided by large companies. To this end I propose the idea *Composable Service*. Its basic idea is that many services can be constructed from basic ones such as image recognition, speech-to-text, and recommendation, to meet new application requirements. Modularity and composition will be the key to increasing usage of ML-based data analytics.

Composing components into a more complex entity is not uncommon to see in the computer science. One such example is the composition of web services. A web service is a software application that is identified by a URI and supports machine-to-machine interaction over a network. Messages in formats such as XML and JSON are transferred among web services according to their prescribed interfaces. The potential of the web services application architecture lies in that the developers can compose multiple services and build a larger application via the network. In web service composition, one problem is to select proper participant services so that they can work together properly. A lot research effort has been made on composition methods that consider information such as

interfaces, message types, and dynamic message sequences exchanged [112].

A similar paradigm is the microservices architecture. With this architecture, a large monolithic software application should be decomposed into small components, each with distinct functionalities. These components can communicate with each other via predefined APIs. This approach provides multi-folds of benefits, such as module reusability, service scalability, fault isolation, etc. Many companies, such as Netflix, have successfully adopted this approach. In the composition of different microservices, the application API plays a key role[1]. Another field that advocates the composition approach is the serverless computing, where the stateless functions can be composed into more complex ones [113]. Based on the observation that existing serverless systems spend a large portion of time on booting function containers and interaction between functions, the SAND system [26] investigates the combination of different functions. By proposing application-level sandboxing and a hierarchical message bus, this system reduces latency and improves resource utility.

In this chapter, as contribution, the Zoo system provides a small Domain-specific Language (DSL) to enable composition of advanced data analytics services. Benefiting from OCaml's powerful type system, the Zoo provides type checking for the composition. Besides, the Zoo DSL supports fine-grained version control in composing different services provided by different developers, since the code of these services may be in constant change.

Another challenge in conducting ML based data analytics on edge devices is the deployment of data analytics services. Most existing machine learning frameworks, such as TensorFlow and Caffe, focus mainly on the training of analytics models. On the other hand, end users, many of whom are not ML professionals, mainly use trained models to perform inference. This gap between the current ML systems and users' requirements is growing.

The deployment of service is close to the idea of model serving. The Clipper [25] serving system is used for ML model based prediction, and it features choosing the model that has the lowest latency from models on multiple ML frameworks. It enables users to access models based on multiple machine learning frameworks. These models are implemented in the form of containers. Compared with Clipper, the TensorFlow Serving [73] focuses on using TensorFlow itself as model execution framework. The models are in the form of `SavedModel`, and they can be deployed as container that contains TensorFlow to serve prediction requests. Several microservice deployment systems, such as Seldon [75], have been developed since the publication of our work on Zoo. It uses Docker for deploying models. Seldon defines inference graph based on the model and then deploys the graph with the container-orchestration system Kubernetes in deployment or production environments. Another field that employ the idea of service deployment is in the serverless computing. In serverless platforms such as Amazon Lambda and OpenLambda, utilising the powerful ecosystem of existing cloud providers, the stateless functions provided by users can be deployed on different types of devices to get access to resources such as database and cloud files. For this aspect, as contribution, the Zoo DSL also involves deploying composed services to multiple backends: not only containers, but also Unikernels and JavaScripts.

In summary, this chapter identifies the two challenges that are not yet well explored in the literature about data analytics on edge devices: service composition and deployment, and presents the Zoo system to address the previous two challenges. I design a small DSL to enable script sharing, type-checked composition of different data analytics services with

[1]Engineering Trade-Offs and The Netflix API Re-Architecture. The Netflix Tech Blog. `https://bit.ly/3evFz9g`

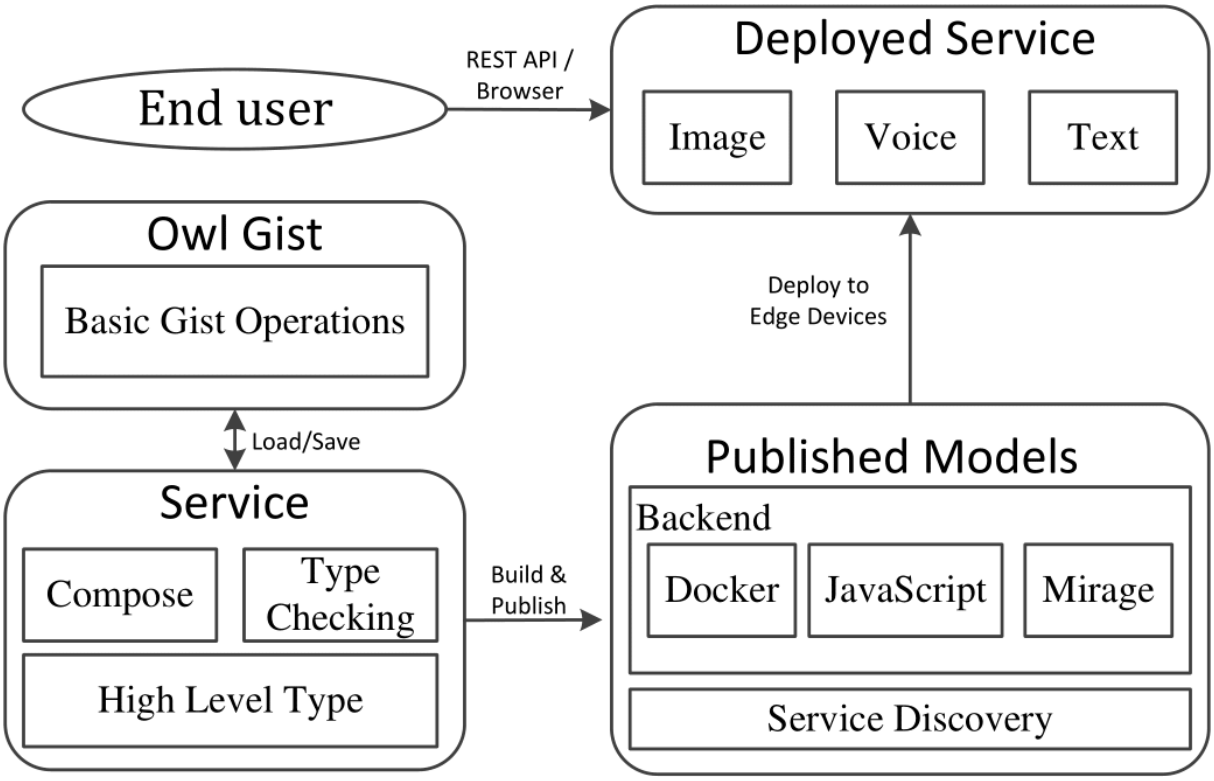


Figure 5.1: Zoo system architecture.

version control, and deployment of services to multiple backends. Finally, this chapter presents a use case to demonstrate the expressiveness of this DSL, and evaluate different deployment backends for analytics services.

5.2 System Design

The Zoo system is implemented on Owl. Initially, the Zoo system is designed to make it convenient for developers to share their OCaml code snippets. The design principle is to make the whole ecosystem open, flexible, and extensible. One typical scenario for using the basic functions of Zoo can be described as follows. Developer A creates a script, uploads it to GitHub Gist, and then shares it using a string Gist id. When developer B gets this id, he can use the functions from A's scripts by simply using the "#zoo" directive in his code. All the OCaml files in the Gist will be imported as modules for B to use. Gist is a service provided by GitHub for the user to dump and share their code snippets with version control. A gist may contain one or more files. Each gist can be accessed using a unique id string.

Based on these basic functionalities, I extend the Zoo system to address the composition and deployment challenges. First, I would like to briefly introduce the workflow of Zoo as shown in Fig. 5.1. The workflow consists of two parts: *development* on the left side and *deployment* on the right.

Development concerns the design of interaction workflow and the computational functions of different services. One basic component is the Gist. By using Zoo, a normal Gist script will be loaded as a module in OCaml. To compose functionalities from different Gists only requires a developer to add one configuration file to each Gist. This file is in JSON format. It consists of one or more name-value pairs. Each pair is a signature for a function the script developer wants to expose as a service. These Gists can be imported and composed to make new services. When a user is satisfied with the result, she can

save the new service as another Zoo Gist.

Deployment takes a Gist and creates models in different backends. These models can be published and deployed to edge devices. It is separated from the logic of development. Basic services and composed ones are treated equally. Besides, users can move services from being local to remote and vice versa, without changing the structure of the constructed service. Deployment is not limited to edge devices, but can also be on cloud servers, or a hybrid of both cases, to minimise the data revealed to the cloud and the associated communication costs. Thus, by this design, a data analytics service can easily be distributed to multiple devices.

In the rest of this section, I will elaborate on the design and give details of different parts of this workflow.

5.2.1 Service

Gist is a core abstraction in Zoo. It is the centre of code sharing. However, to compose multiple analytics snippets, Gist alone is insufficient. For example, it cannot express the structure of how different pieces of code are composed together. Therefore, I introduce another abstraction: `service`.

A service consists of three parts: *Gists*, *types*, and the *dependency graph*. *Gists* is the list of Gist ids this service requires. *Types* is the parameter types of this service. Any service has zero or more input parameters and one output. This design follows that of an OCaml function. *Dependency graph* is a graph structure that contains information about how the service is composed. Each node in it represents a function from a Gist, and contains the Gist's name, id, and a number of parameters of this function.

Zoo provides three core operations about a service: create, compose, and publish. The *create_service* creates a dictionary of services given a Gist id. This operation reads the service configuration file from that Gist, and creates a service for each function specified in the configuration file. The *compose_service* provides a series of operations to combine multiple services into a new service. A compose operation does type checking by comparing the "types" field of two services. An error will be raised if incompatible services are composed. A composed service can be saved to a new Gist or be used for further composition. The *publish_service* makes a service's code into such forms that can be readily used by end users. Zoo is designed to support multiple backends for these publication forms. Currently it targets Docker container, JavaScript, and MirageOS [27] as backends.

5.2.2 Type Checking

As mentioned in Section 5.2.1, one of the most important tasks of service composition is to make sure the type matches. For example, suppose there is an image analytics service that takes a PNG format image, and if I connect to it another one that produces a JPEG image, the resulting service will only generate meaningless output for data type mismatch. OCaml provides primary types such as integer, float, string, and Boolean. The core data structure of Owl is ndarray. However, all these types are insufficient for high level service type checking as mentioned. That motives us to derive richer high-level types.

To support this, I use generalised algebraic data types (GADTs) in OCaml. There already exist several model collections on different platforms, e.g. Caffe [72] and MxNet [114]. I observe that most current popular deep learning models can generally be categorised into three fundamental types: `image`, `text`, and `voice`. Based on them, I define sub-types for each: PNG and JPEG image, French and English text and voice, i.e. `png img`, `jpeg img`, `fr text`, `en text`, `fr voice`, and `en voice` types. More can be further added easily

in Zoo. Therefore type checking in OCaml ensures type-safe and meaningful composition of high level deep learning services.

5.2.3 Backend

Recognising the heterogeneity of edge device deployment, one key principle of Zoo is to support multiple deployment methods. Containerisation as a lightweight virtualisation technology has gained enormous traction. It is used in deployment systems such as Kubernetes. Zoo supports deploying services as Docker containers. Each container provides a RESTful API for end users to query.

Another backend is JavaScript. Using JavaScript to do analytics aside from front end development has begun to attract interest from academia [109] and industry, such as TensorFlow.js and Facebook's Reason language. By exporting OCaml and Owl functions to JavaScript code, users can do complex data analytics on web browser directly without relying on any other dependencies.

Aside from these two backends, I also initially explore using MirageOS as an option. MirageOS is an example of Unikernel, which builds tiny virtual machines with a specialised minimal OS that hosts only one target application. Deploying to Unikernel has proved to be of low memory footprint, and thus is quite suitable for resource-limited edge devices.

5.2.4 DSL

Zoo provides a minimal DSL for service composition and deployment.

Composition To acquire services from a Gist of id *gid*, I use $\$gid$ to create a dictionary, which maps from service name strings to services. I implement the dictionary data structure using `Hashtbl` in OCaml. The # operator is overloaded to represent the "get item" operation. Therefore,

$$\$\text{gid}\#\text{sname}$$

can be used to get a service that is named "sname". Now suppose I have n services: f_1, f_2, ..., f_n. Their outputs are of type t_{f1}, t_{f2}, ..., t_{fn}. Each service s accepts m_s input parameters, which have type t_s^1, t_s^2, ..., $t_s^{m_s}$. Also, there is a service g that takes n inputs, each of them has type t_g^1, t_g^2, ..., t_g^n. Its output type is t_o. Here Zoo provides the `$>` operator to compose a list of services with another:

$$[f_1, f_2, \ldots, f_n]\$ > g$$

This operation returns a new service that has $\sum_{s=1}^{n} m_s$ inputs, and is of output type t_o. This operation does type checking to make sure that $t_{fi} = t_g^i, \forall i \in 1, 2, \ldots, n$.

Deployment Taking a service s, be it a basic or composed one, it can be deployed using the following syntax:

$$s\$@ \text{ backend}$$

The `$@` operator publish services to certain backend. It returns a string of URI of the resources to be deployed.

Note that the `$>` operator leads to a tree-structure, which is in most cases sufficient for our real-world service deployment. However, a more general operation is to support a graph structure. This will be my next-step work.

5.2.5 Service Discovery

The services require a service discovery mechanism. For simplicity's sake, each newly published service is added to a public record hosted on a server. The record is a list of items, and each item contains the Gist id that service based on, a one-line description of this service, string representation of the input types and output type of this service, e.g. "image → int → string → text", and service URI. For the container deployment, the URI is a DockerHub link, and for JavaScript backend, the URI is a URL link to the JavaScript file itself. The service discovery mechanism is implemented using an off-the-shelf database.

5.2.6 Version Control

Developers would modify and upload their scripts several times. As such, each version of a script is assigned a unique id in Gist. Zoo supports specifying a version of a Gist.

The naming scheme of a Gist is `gid/[vid|latest]/pin`. A user can either choose a specific version id, or he can use the latest version, which means the newest version on local cache. Obviously, using `latest` introduces cache inconsistency. The latest version on one machine might not be the same on the other. To get the up-to-date version from a Gist server, the download time of the latest version on a local machine will be saved as metadata. The newest version on server will be pulled to the local cache after a certain period of time, if `latest` flag is set in the Gist name. Ideally, every published service should contain a specific version id, and `latest` should only be used during development.

Zoo can analyse dependency information of a Gist and save it. When the `pin` flag is set, the Gist dependency graph of current script will be saved or loaded.

5.3 Use Case

To illustrate the workflow above, let us consider a synthetic scenario. Alice is a French data analyst. She knows how to use ML and DL models on existing platforms, but is not an expert. Her recent work is about testing the performance of different image classification neural networks. To do that, she needs to first modify the image using the DNN-based Neural Style Transfer (NST) algorithm. NST takes two images and outputs to a new image, which is similar to the first image in content and the second in style. This new image should be passed to an image classification DNN for inference. Finally, the classification result should be translated to French. She does not want to put academic-related information on Google's server, but she cannot find any single pre-trained model that performs this series of tasks.

Here comes the Zoo system to help. Alice finds Gists that can do image recognition, NST, and translation separately. Even better, she can perform image segmentation to greatly improve the performance of NST [115] using another Gist. All she has to provide is some simple code to generate the style images she needs to use. She can then assemble these parts together easily using Zoo.

Note that the Gist id used in the code is shortened from 32 digits to 5 due to column length limit. Once Alice creates the new service and publishes it as a container, she can then run it locally, send a request with image data to the deployed machine, and get image classification results back in French.

```
open Zoo
(* Image classification *)
let s_img = $ "aa36e" # "infer";;
(* Image segmentation *)
let s_seg = $ "d79e9" # "seg";;
(* Neural style transfer *)
let s_nst = $ "6f28d" # "run";;
(* Translation from English to French *)
let s_trans = $ "7f32a" # "trans";;
(* Alice's own style image generation service *)
let s_style = $ alice_Gist_id # "image_gen";;

(* Compose services *)
let s = [s_seg; s_style] $> s_nst
  $> n_img $> n_trans;;
(* Publish to a new Docker Image *)
let pub = (List.hd s) $@
  (CONTAINER "alice/image_service:latest");;
```

Listing 10: An example of using the Zoo DSL.

5.4 Evaluation

5.4.1 Backends

First, I compare the performance of different backends I use. Specifically, I observe three representative groups of applications: (1) `map` and `fold` operations on ndarray; (2) using gradient descent, a common numerical computing subroutine, to get `argmin` of a certain function; (3) conducting inference on complex DNNs, including SqueezeNet [98] and a VGG-like convolution network. The evaluations are conducted on a ThinkPad T460S laptop with Ubuntu 16.04 operating system. It has an Intel Core i5-6200U CPU and 12GB RAM.

Owl library provides a "base" library in pure OCaml that shares the core functions of the Owl library, which mixes both OCaml code and C code to improve the computational performance. Note that, for convenience, I refer to the pure implementation of OCaml and the mix implementation of OCaml and C as `owl-base` and `owl` separately, but they are in fact all included in the Owl library.

The OCaml compiler can produce two kinds of executables: bytecode and native. Native executables are compiled specifically for certain architectures and are generally faster, while bytecode executables have the advantage of being portable. A Docker container can adopt both options. Therefore, in the evaluation infrastructure, I use `native-owl` and `bytecode-owl` to represent both kinds of executables for the `owl` library; the `native-base` and `bytecode-base` represent that for the `owl-base` library.

For Mirage compilation, I use both libraries, denoted by `mirage-owl` and `mirage-base`. For the JavaScript, since the Owl library contains functions that are implemented in C, it cannot be directly supported by `js-of-ocaml`[2], the tool I use to convert OCaml code into JavaScript. Therefore I can only use the `owl-base` to implement the required computation in pure OCaml, and then convert it to JavaScript code.

[2]Compiler from OCaml to JavaScript. `http://ocsigen.org/js_of_ocaml/`

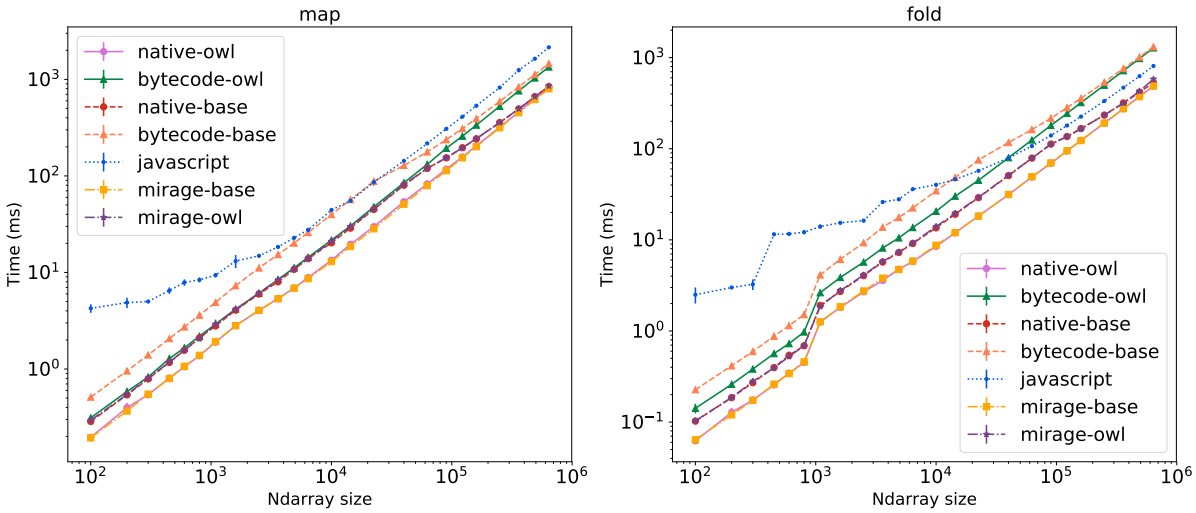


Figure 5.2: Performance of map and fold operations on laptop.

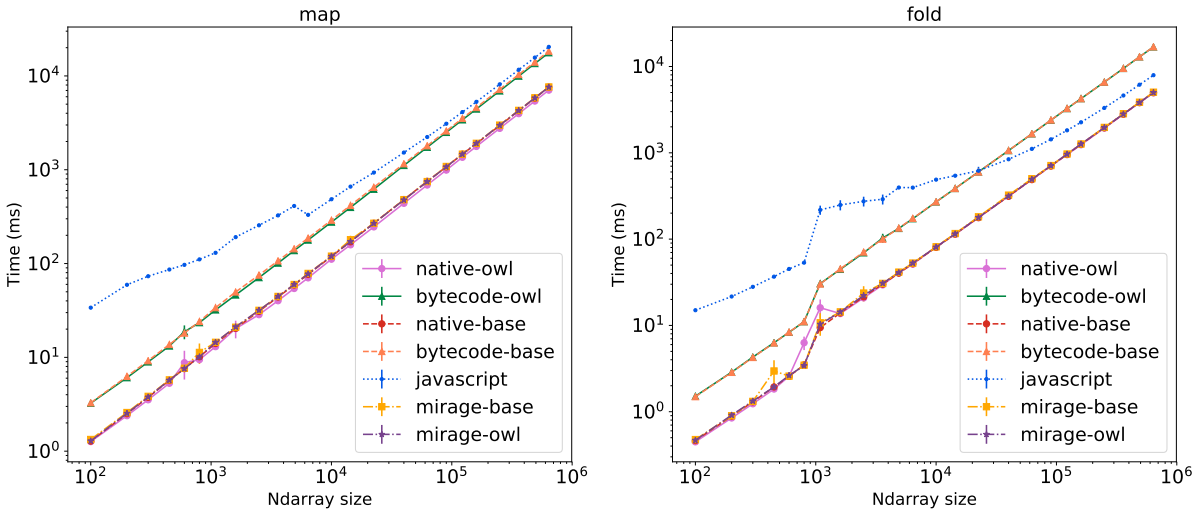


Figure 5.3: Performance of map and fold operations on Raspberry Pi.

Fig. 5.2 shows the performance of map and fold operations on ndarray. I use simple functions such as plus and multiplication on 1-dimensional (size $< 1,000$) and 2-dimensional arrays. The log-log relationship between total size of ndarray and the time each operation takes keeps linear. For both operations, `owl` is faster than `owl-base`, and native executables outperform bytecode ones. The performance of Mirage executables is close to that of native code. Generally JavaScript runs the slowest, but note how the performance gap between JavaScript and the others converges when the ndarray size grows. For the fold operation, JavaScript even runs faster than bytecode when input size is sufficiently large.

I have also conducted the same evaluation experiments on RaspberryPi 3 Model B. Fig. 5.3 shows the performance of fold operation on ndarray. Despite that the performance is much slower than that on the laptop machine, the results are similar. Note that in both Fig. 5.2(b) and Fig. 5.3(b), there is an obvious increase in time used at around input size of 10^3 for fold operations, while there is not such change for the map operation. That is because I change the input from one dimensional ndarray to two dimensional starting that size. This change does not affect map operation, since it treats an input of any dimension as a one dimensional vector. On the other hand, the fold operation considers the factor of dimension, and thus its performance is affected by this change.

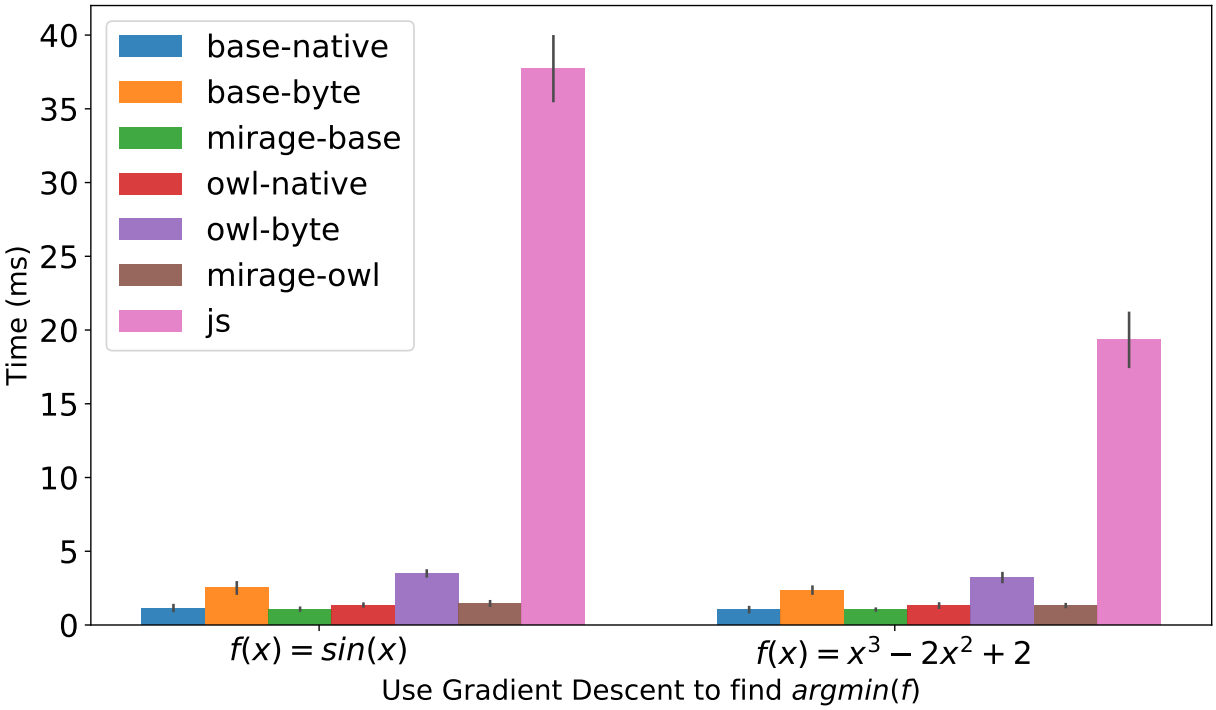


Figure 5.4: Performance of gradient descent on function f to find $argmin(f)$.

Table 5.1: Inference speed of deep neural networks.

Time (ms)	VGG	SqueezeNet
owl-native	7.96 (± 0.93)	196.26(± 1.12)
owl-byte	9.87 (± 0.74)	218.99(± 9.05)
base-native	792.56(± 19.95)	14470.97 (± 368.03)
base-byte	2783.33(± 76.08)	50294.93 (± 1315.28)
mirage-owl	8.09(± 0.08)	190.26(± 0.89)
mirage-base	743.18 (± 13.29)	13478.53 (± 13.29)
JavaScript	4325.50(± 447.22)	65545.75 (± 629.10)

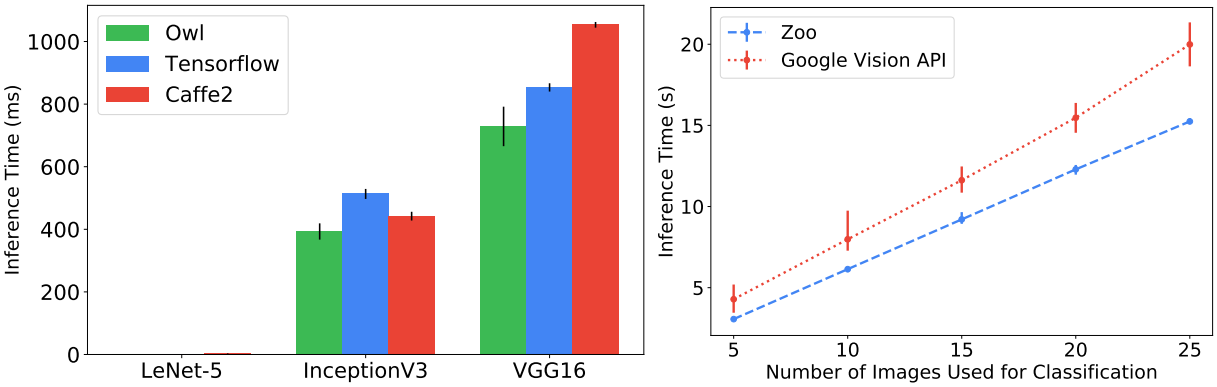
In Fig. 5.4, I want to examine if the above observations still hold true in more complex numerical computation. I choose to use a Gradient Descent algorithm to find the value that locally minimises a function. I choose the initial value randomly between $[0, 10]$. For both $sin(x)$ and $x^3 - 2x^2 + 2$, I can see that JavaScript runs the slowest, but this time the `owl-base` slightly outperforms `owl`.

I further compare the performance of DNN, which requires large amount of computation. I compare SqueezeNet and a VGG-like convolution network. They have different sizes of weight and networks structure complexities. In Table. 5.1, each item consists of the average execution time and the standard deviation based on 20 repeated measurements. It shows that, though the performance difference between `owl` and `owl-base` is not obvious, the former is much better. So is the difference between native and bytecode for `owl-base`. JavaScript is still the slowest. The core computation required for DNN inference is the convolution operation. Its implementation efficiency is a key to these differences.

Besides the fact that all backends run about one order of magnitude slower than that on the laptop, previous observations still hold. This figure also implies that, on resource-limited devices such as the RaspberryPi, the key difference is between native code and bytecode, instead of `owl` and `owl-base` for this operation.

Table 5.2: Size of executables generated by backends.

Size (KB)	native	bytecode	Mirage	JavaScript
base	2,437	4,298	4,602	739
native	14,875	13,102	16,987	-



(a) Comparison of inference time of image recognition service between Owl and other DNN frameworks.

(b) Performance of local image classification with Machine Learning Vision service.

Figure 5.5: Performance evaluation of Zoo services.

Finally, I also briefly compare the size of executables generated by different backends. On different backends, I build the image classification DNN inference application using the SqueezeNet architecture. The size of different resulting executables are shown in Table 5.2. It can be seen that `owl` executables have larger size compared to `owl-base` ones, and JavaScript code has the smallest file size.

It can be seen that there does not exist a dominant method of deployment for all these backends. It is thus imperative to choose suitable backend according to deployment environment.

5.4.2 Performance of Services

Since the Zoo relies on inference using Owl's Neural Network module, I want to compare the inference time on Owl and the other state-of-art deep learning platforms. In this evaluation I use the TensorFlow [21] and Caffe2 [21] for comparison. I choose three representative DNN models that vary greatly in both architectural complexity and parameter sizes: 1) one small neural network (LeNet-5 [116]) that only consists of 8 nodes and contains about 240KB parameters (each parameter in a model is represented by a 32-bit float number) for the MNIST handwriting recognition task; 2) a VGG16 model that has a simple architecture with 38 nodes but a large number of parameters (500MB) for real-world image recognition tasks; 3) an InceptionV3 model also for image recognition, with fewer parameters (100MB), but a far more complex architecture (313 nodes). I compare the time it takes for each model to finish its inference task using different frameworks: Owl, TensorFlow, and Caffe2. Each measurement is repeated 20 times.

The results are shown in Fig. 5.5a. The result on LeNet-5 architecture may seem absent, but that's only because its inference time is small compared to the other two network architectures, due to its simple structure. Regardless of great diversities in these models' architectures and sizes, Owl takes less time to do inference than TensorFlow and Caffe2. It means that Owl can achieve both expressiveness and good performance. The superior performance of Owl on large models is attributed to its efficient math operations.

Next, I investigate the performance of Zoo system compared with Google ML API.

I deploy services on local devices, so what is their performance compared with popular cloud-based analytic solutions such as Google ML API?

The Google Cloud Vision API [117] encapsulates machine learning models in a REST API. It can classify images into thousands of categories as well as detecting individual objects and faces within images, and finds and reads printed words contained within images. Its workflow is simple: a user creates a service token on the Google Cloud Platform (GCP), and passes the token and an image to Google's server for processing, and then the processed results will be returned to the user as response in the form of JSON. In this evaluation I compare the image classification model.

I deploy Vision API service on GCP. The network connection bandwidth is 34 Mbps, measured using `speedtest.net`. This bandwidth is averaged from 20 different measurements. In the previous section I have shown a deployment example of our image classification service on local devices with Zoo. For this evaluation, I deploy this service on a ThinkPad T460s laptop with a Intel Core i5-6200U CPU. First, I have collected 100 animal images as a dataset[3]. These images are of different sizes, ranging from 7KB to 1243KB, to better simulate users' requests in real world. Specifically, I compare the time required for both methods to process different numbers of images. The measurement at each point is repeated for 10 times.

The results are shown in Fig. 5.5b. It shows that our Zoo service achieves lower response latency. Even with such a moderate level of network connection, the performance of the cloud API is still not satisfactory. When the number of input images from a user gradually increases from 5 images to 25 images, the response time of the Zoo service increases linearly, which means the process time of each image basically keeps constant despite the size difference of input images, while the increase speed Google service's response time seems to grow with more input images. Of course, the difference of bandwidth can significantly influence the inference time using the cloud service, where a better network connection tends to yield faster inference time for the end user. However, one advantage of the Zoo service, as shown in this result, is that the response time keep stable (about 0.6s per image) with very small deviation, and thus predictable. The cloud service, on the other hand, shows relatively large deviation, and with the change of network connection, the response time could easily fluctuate. For services that run one edge devices in unstable network environment, local deployment and execution is a preferable choice.

5.5 Discussion

One thing to note is that, in service composition, type checking is a nice property to have, but not the only one. From web services to microservices, the industry and researchers have been studied the composition issue for years. Besides checking the static information such as message types and interfaces etc., sometimes the dynamic behaviour between services should also be checked [118]. It is the same in our data analytics services composition scenario.

For example, the Generative Adversarial Network (GAN) is a huge family of networks [119]. A GAN consists of two parts: generator and discriminator. The generator tries its best to synthesise images based on existing parameters. The discriminator takes the images produced by the generator and tries its best to separate the generated data from true data, using a Boolean or percentage value. This mutual deception process is iterated until the discriminator can no longer tell the difference between the generated data and the true data. Using Zoo, the users may want to compose a generator with different

[3]Available at `https://goo.gl/BJqiBD`

discriminators to see which combination produces the most trustworthy fake images. To do this, only matching the types of these two services is not enough. The users also need to specify the dynamic information such as the order and number and messages exchange in between.

To solve this problem, some kind of formalisms may need to be introduced in as theoretical foundation to structure interaction and reason over communicating processes between services. One such option is the Session Types [120]. Session types are a type discipline for communication-centric programming. It is based on the π-calculi, and its basic idea is that the communication protocol can be described as a type, which can be checked at runtime or statically. The Session Types has gained much attention recently, and is already implemented in multiple languages, including OCaml. This approach can effectively enhance the type checking in Zoo, and is a promising future direction to pursue in my next step on this work.

5.6 Conclusion

This chapter identified the two challenges that were not yet well explored in the literature about data analytics on edge devices: service composition and deployment, and presented the Zoo system to address the previous two challenges. Zoo provides a small DSL to enable script sharing, type-checked composition of different data analytics services with version control, and deployment of services to multiple backends. It benefits from OCaml's powerful type system. A use case was presented to demonstrate the expressiveness of this DSL in composing advanced ML services such as image recognition, text translation, etc. The Zoo DSL also enables deploying composed services to multiple backends: containers, unikernels, and JavaScripts; service deployment often requires choosing a suitable one. I evaluated the performance of different backends using three representative groups of numerical operations as workload. The results showed that the performance on mirage unikernels is similar to that on containers regarding both the Owl library and the pure OCaml implementation `owl-base`. JavaScript runs the slowest, but provides small executables and good portability.

Once the computations are deployed, they may need to collaborate with each other, and *barriers* are required to control the synchronisation among different nodes. This topic will be discussed in the next chapter.

Chapter 6

Computation Synchronisation

Computations can be composed and deployed, and, as is often the case, collaboration among the deployed computations might be required. One important example is the training of machine learning models in distributed learning.

As discussed in Sec. 2.4, a plethora of research has been conducted on improving the performance in distributed training. In this chapter, I focus on the *barrier control*, the mechanism that provides trade-off between computation accuracy and progress. Understanding the behaviour of a barrier control method is crucial in improving the performance of distributed computation.

My work in this chapter is based on Probabilistic Synchronous Parallel (PSP) that is proposed in [34]. The PSP is a new barrier control method that introduces a new dimension of trade-off using a `sampling` primitive. It can be applied to both centralised and distributed solutions, and thus is especially suitable to be used in heterogeneous environments.

PSP enables a large tuning space in barrier control method design. To explore this space and get a better understanding of its impact on performance, I aim to conduct a thorough evaluation to compare PSP with existing barrier methods. I find two challenges during evaluation. The first is to evaluate a barrier's impact on performance without being affected by other factors such as training hyper-parameters. The reason is that algorithms such as stochastic gradient descent are immune to error to a certain degree, so the traditional metric of model accuracy does not fully show this impact of barriers. The second is to provide intuitive insights on the theoretical analysis of PSP, such as convergence properties of barrier methods.

In this chapter, as my contributions, I summarise two types of inconsistency from existing literature as quantitative metrics to address these two challenges. I also conduct evaluation of barrier control methods using these metrics, as well as the two commonly used ones, step progress and model accuracy. By choosing proper barrier control methods with suitable parameters, the performance of computation can further be improved at a high level.

This chapter mainly consists of one paper I participated in that was based on [34] with the original authors. This paper was submitted to SysML2020. I led this paper and contributed the study on inconsistency metrics and system evaluation. I followed the evaluation method in the original paper, especially about the step progress in Sec. 6.2.1, but I have built a new simulation platform and real-world experiments to re-run all the previous experiments and verify the conclusions in [34], besides the new evaluations I have added.

6.1 Inconsistency in Barrier Control

I propose to use two metrics to denote the inconsistency in distributed training: progress inconsistency and sequence inconsistency. They are two orthogonal aspects: the first one presents the accuracy of training without the influence of the specific application used, and the second metric shows the deviation of real training progress from an ideal one. Besides the existing step progress and accuracy, a study of these two metrics generates a full picture of the performance of barrier control methods. These two metrics are widely mentioned in previous literature, but they are not yet clearly recognised and stated. A lot of work mentions only one of them in a qualitative way, and without deep study.

System Model To analyse the consistency property, I follow a common system model. Assume there are P workers, and each of them pushes additive updates to a shared parameter $x \leftarrow x + u$. A worker makes updates u to x by step (or *clock*). Every worker has its own clock that starts with value 0. Every time it finishes its unit of job and pushes the updated model to the parameter server, its clock increases by one. A server maintains a vector C of size P, each element C_w indicating the current newest clock of worker w. Following the modelling in [92] and [121], a parameter server accepts a stream of updates from workers. An update $u_{p,c}$ denotes the update created by worker p at its own clock c. I assume that one clock corresponds to an iteration of work.

6.1.1 Progress inconsistency

Following the system model, in a Parameter Server paradigm with data parallelism, a worker at time t_1 pulls a weight value from the parameter server and begins to train a local model. Once the job is done and updates are pushed to the server at time t_2, the weight at parameter server probably has already been updated several times by other workers. This lag creates model inconsistency [12].

Here I derived the definition of *progress inconsistency*: the number of updates that have occurred between its corresponding read and update operations. To use the notation described above, one worker cached the clock vector C^{t_1} when it pulls the newest parameters from server at time t_1. When it pushes its own update to the server at time t_2, the current clock at server has already become C^{t_2}. The number of lagged updates is then $C^{t_2} - C^{t_1}$.

Due to algorithm's tolerance of this inconsistency, the final training performance varies. Therefore, model accuracy is both affected by hyper-parameters such as learning rate, and the progress inconsistency. To study one factor, the other one needs to be fixed. Existing literature mostly fixes the latter and focuses on tuning the hyper-parameters [88, 90, 93, 121]. In such work, the authors simply assume the progress inconsistency is bounded by either a constant value or a value that is related to gradient bound and time. These assumptions are made without explicit explanation or simply because of "empirical observation".

The importance of introducing this metric is at least two-fold. First, the progress inconsistency shows the accuracy of model without the influence of hyper-parameters of specific applications and algorithms used. Second, it provides a qualified way to back up various assumptions made about an inconsistency bound in future research on barrier control.

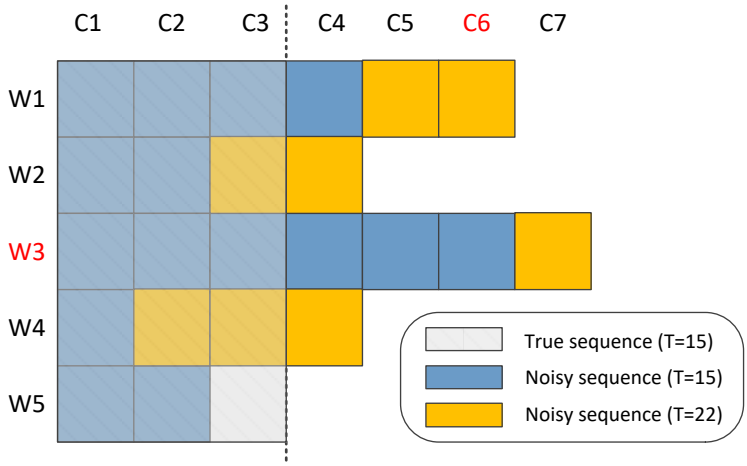


Figure 6.1: An illustrative example of two types of inconsistency.

6.1.2 Sequence inconsistency

In a distributed machine learning process, the workers keep generating updates, and a shared model is updated with them continuously. I count these updates by first looping over all workers at one clock, and then across all the clocks. In this process, each one is incrementally indexed by integer t. The total length of this update sequence is T. Ideally, in a fully deterministic system, the ordering of updates in this sequence should be fixed. This sequence is called a *true sequence*. However, in reality, what a system generates is often a *noisy sequence*, where updates are reordered irregularly. These two sequences share the same length. I define *sequence inconsistency* as the number of index difference between these two sequences. It shows how much a series of updates deviate from an ideal case. If sequence inconsistency is bounded, it means that given enough time, the difference between a true sequence and a noisy sequence is also bounded. While progress inconsistency focuses on how lagged a worker is compared with other workers, the sequence inconsistency shows how much a series of updates deviate from an ideal case.

This metric is a key instrument in theoretically proving the convergence property of an asynchronous barrier method [32, 87, 92]. For example, in proving the convergence of PSP, it is essential to show that, as long as the difference between the noisy update sequence and the ideal sequence is bounded, and that the nodes in the system do not lag behind too far away, PSP guarantees that (with probability) the difference between the actual and optimal result diminishes as more updates are generated by workers.

6.1.3 Example

The description of these two types of inconsistencies might seem confusing at first sight. To better understand them, I illustrate them with an example. Fig. 6.1 demonstrates an example of the difference. This example contains 5 workers, denoted with w_i; each worker has its own clock denoted by c_j. In the same amount of time, different workers may proceed with different number of clocks. Each clock corresponds to one update from that worker. In this example, let us focus on observing two time screenshots: the "current" time t_1, denoted by blue squares, and a later time t_2, at which point the clocks of the workers progress to yellow squares.

Think from the perspective of worker w_3. Currently at time t_1, the clock of w_3 is 6,

while the clock of w_1 is 4, etc. At this point, the total length of the update sequence is $T = 15$, which means that the server has already accepted 15 updates from these 5 workers. However, given the same T, an ideal true sequence should have been what the opaque grey area looks like. Compared with the true sequence, the noisy sequence (blue squares) should have had (w_2, c_3), (w_4, c_2), (w_4, c_3), and (w_5, c_3), but in reality it does not; also, it should not have (w_1, c_4), (w_3, c_4), (w_3, c_5), and (w_3, c_6). Each tuple here represents one square. Together, these two sets of disjoint updates account for the *sequence inconsistency*. w_3 knows the current progress of all the other workers, since they have achieved a synchronisation at t_1 through the parameter server.

Let's keep following the perspective of worker w_3. Moving forward in time, the workers continue their job of computing updates. w_3 continues to compute the next update based on the synchronised status at t_1. After it finishes its work at t_2, workers w_1 and w_2 have already pushed two new updates to the server, and w_4 has pushed three new updates (indicated by yellow squares). The w_5 is slow and has not yet finished any update. Apparently, the update produced by w_3 at clock c_7 does not include the newest updates from the other workers during t_1 to t_2, i.e., the yellow squares. This yellow area is what I called *progress inconsistency*.

Other definitions of inconsistency exist besides these two. For example, in [121], the authors define the inconsistency metric as the number of updates that rely on the same model replica, to facilitate their algorithm. One other definition is that the longest sequence of updates from other workers immediately before an update. However, these definitions are rarely used and will not be further discussed.

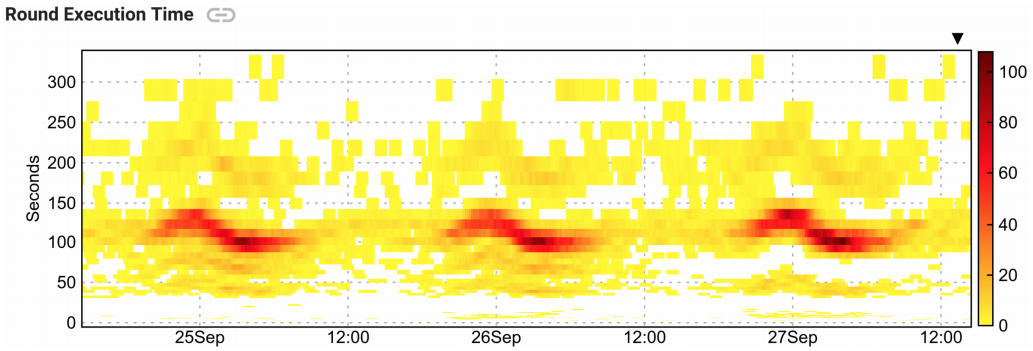
6.2 Evaluation of Barrier Methods

In the evaluation, I investigate the performance of PSP. I use four metrics of barrier strategies: the step progress (Sec. 6.2.1), accuracy (Sec. 6.2.2), and the two types of inconsistency I have proposed (Sec. 6.2.3 and Sec. 6.2.4). For each metric, I examine the impact of barrier parameters, stragglers in the system, and network size.

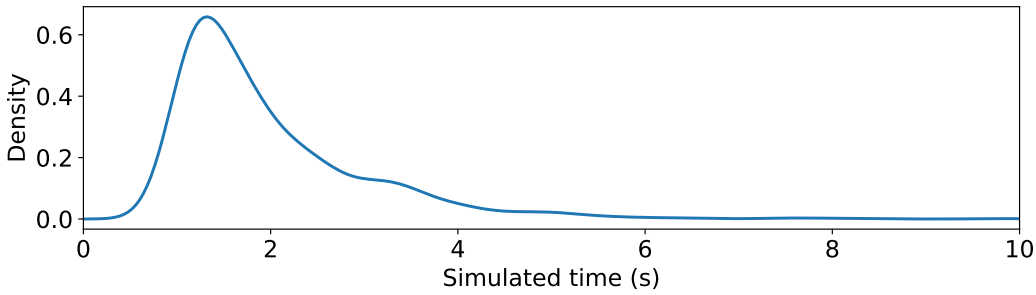
Models and algorithms I choose three applications for the evaluation: training a LDA topic model using collapsed sampling (CGS) method [122], training a Deep Neural Network (DNN) using SGD, and SGD-based Matrix Factorisation (MF) [123]. For the DNN training, I use a 9-layer structure that is similar to the DNN that [116] propose. For the LDA model, I implement the standard CGS algorithm as described in [124]. The MF is implemented using a similar algorithm as described in [125] with regularisation.

Datasets For LDA, I use a New York Times dataset that contains 8447 documents (D) and 3012 vocabularies (W). The topic number K is set to 10. Parameter α is set to $50/K$ and β is 0.1 as in [122]. For DNN, I use the MNIST handwritten digits dataset [116]. The learning rate has a decay factor of $1e4$. For MF, I use the MovieLens [126] dataset, which contains about 1 million movie ratings from 6,049 users and 3,052 movies. The original dataset is divided into a training set (90%) and a test set (10%). I choose latent features number $K = 100$, and a regularisation parameter $\beta = 0.1$. In all three applications the data are divided among all the workers.

The main idea of PSP is that the impact of outliers and stragglers can be minimised by dropping updates from a certain portion of workers and allowing for synchronisation from only part of the system. Though the PSP barrier control method is compatible with existing synchronisation methods and can be used in any distributed computing



(a) Round exection time distribution [30]



(b) Simulated time distribution

Figure 6.2: Simulation of real-world processing time in distributed machine learning.

scenario if necessary, it is proposed to address the synchronisation problem in unreliable environment where many distributed devices participate in, such as in the Federated Learning. Towards this end, I have built a simulation platform[1] to reach the necessary large evaluation scale in a controllable dynamic environment.

As an enhancement, I also use the real-world experiments to evaluate different barrier control methods. The experiments run on six nodes with the Actor distributed engine that Owl provides, where all the barrier methods are implemented. In both simulations and real experiments, I use the Parameter Server framework as described in [84]. It consists of one server and many worker nodes. In each step, a worker takes a chunk of training data, calculates the new model weight values as update, and then aggregates these updates to the parameter server, thus updating the shared model iteratively. A worker pulls new model from the server after it is updated.

In the simulation, I make several assumptions. The focus of this chapter is to evaluate the barrier methods, so I do not consider techniques such as optimised SGD, merging multiple updates or compressing them before sending them to the server, even though these techniques have proved to be useful in practice.

During training, the time each worker takes can be divided into three steps: computation of updates, waiting for the other workers according to the barrier method, pushing its updates to and pulling from the parameter server. To simplify this model, in the simulation, I consider only the computation and barrier-based waiting time, with the former representing both computation and transmission time. At each step, a worker's calculation time is chosen randomly from exponential distribution with $\lambda = 1$ plus 1. This set up could be justified by what Google has observed in [30] from its deployed federated learning production systems.

Fig. 6.2a shows the distribution of *round execution time* of devices. A round here includes both training and data/result transmission. According to this figure, in the

[1]Source code on GitHub: `https://github.com/jzstark/Ninox`, accessed Sep-2019.

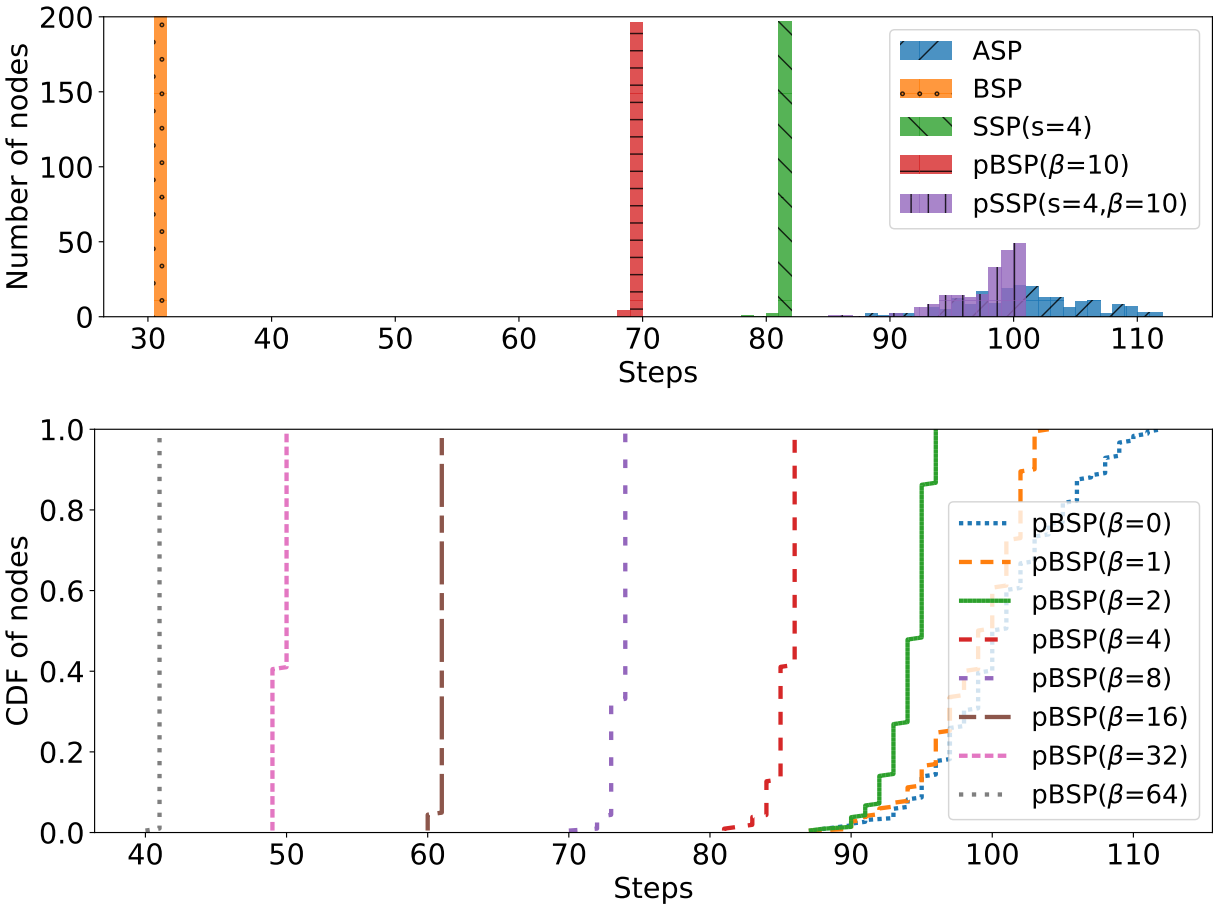


Figure 6.3: (a) Progress distribution in steps; (b) pBSP parameterised by different sample sizes, from 0 to 64. Increasing the sample size make the curves shift from right to left with decreasing spread, covering the whole spectrum from the most lenient ASP to the most strict BSP. [34]

daytime, most devices finish a round within a bounded range, with a small number of slower or faster devices around. Fig. 6.2b is the distribution I use in simulating time.

Also, note that the size of model is far larger than that of messages, I assume the only traffic in the network is caused by parameter updates and models, which are transmitted between workers and the server.

By applying sampling to existing BSP and SSP, I get pBSP and pSSP, the probabilistic versions of BSP and SSP respectively. For the rest of this section, if not explicitly mentioned, I use a network of 100 workers for 100 simulated seconds.

In the rest of this section, I evaluate the performance of PSP with various metrics. I aim to show the wide range of tuning space enabled by the sample size parameter, denoted by β, and how existing barrier methods can be incorporated into PSP. Aside from showing the good performance of PSP, this section also provides a multi-perspective understanding of PSP so that the most suitable sample size can be chosen based on different application scenarios.

6.2.1 Step Progress

The first metric to evaluate is the *step progress*, i.e. iteration speed of different barrier methods. The simulation uses 200 workers and lasts for 200 simulated seconds. The distribution of all workers' step progress after simulation is finished is shown in Fig. 6.3a.

As expected, using the strictest BSP, the system achieves a tightly bounded step

distribution where all the workers are of the same step. However, the workers also progress very slowly. After the simulation, they only proceed to about the 30th step. On the other hand, using ASP can generate a much faster progress which spans from 90 to about 110 steps. However, ASP also leads to a much more loosely spread distribution. It shows the lack of synchronisation among workers in ASP. SSP then allows certain staleness (four in the experiment) and its performance sits between BSP and ASP, with most of the workers share the similar and fast step progress.

Compared to these traditional barrier methods, PSP provides another dimension of performance tuning. The sample size β is to 10, i.e. a sampling ratio of only 5% with a total number of 200 workers. The figure shows that by using pBSP, the workers have steps that are almost as tight as BSP and also faster than BSP. The same goes for comparing pSSP and SSP. It can be seen that, in both cases, PSP can improve the iteration efficiency while limiting dispersion.

In the rest evaluations, I mainly change the same two parameters: the staleness s, and the sampling number β. The choice of sampling number will be based on this evaluation in the rest of this chapter. As to the staleness, I will not focus on varying this parameter since it is the main part of SSP. In PSP the point is only to show how PSP can be compatibly used with existing barrier methods, SSP included. The evaluation results and conclusions of PSP are not sensitive to the specific choice of the staleness parameter. Therefore, I choose suitable staleness setting such as $s = 4$ based on previous SSP papers such as [31].

The next experiment focuses on BSP to further investigate the impact of sample size, as shown in Fig. 6.3b. Here the sample size varies from 0 to 64. At sample size 0, as expected, the pBSP works just like ASP. With increasing sample size, the curve shifts from right to left with tighter and tighter spread, indicating less variance in workers progress. During this process, the pBSP gets more and more similar to SSP and BSP.

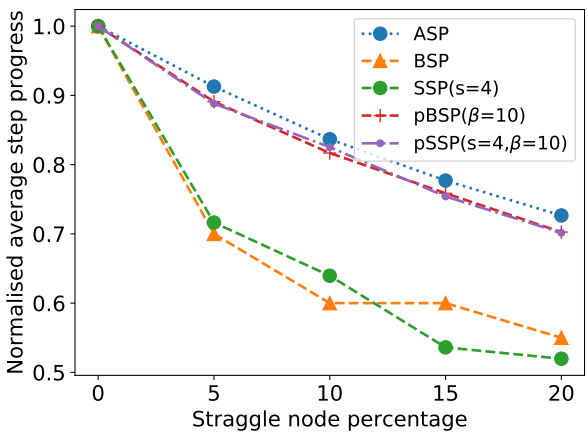
Small sample size means small communication cost on each individual worker. Note that even with a very small sample size, the communication cost of pBSP is almost the same as that of ASP, but it can already effectively synchronise most of the workers. Using larger sample size can further shorten the tail caused by stragglers. This result shows that a small sample size can already effectively increase the probabilistic convergence guarantee even for a system that contains a large number of workers. This point shows the good scalability of the PSP.

Stragglers

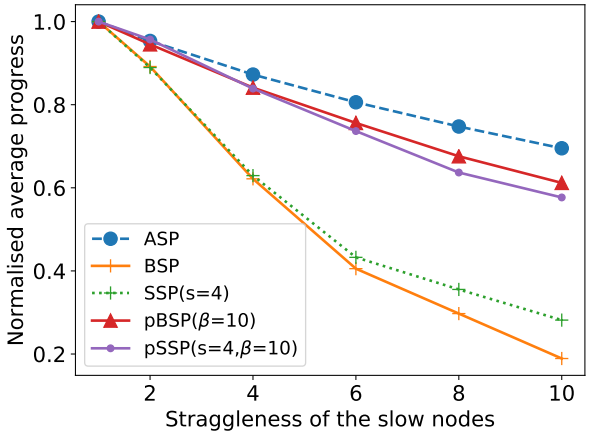
In a realistic distributed training scenario, stragglers are common to see. The stragglers can degrade the training performance to various degrees. Therefore, I will investigate this factor and how PSP can help to mitigate the effect caused by stragglers. This section starts with how stragglers affect the step progress with different barriers.

In this evaluation setting, I inject a certain percentage of slow workers, which are two times slower to finish one iteration than the normal workers. The percentage of slow nodes is gradually increased from 0% to 30%. I measure the average progress of all the workers at 100s and calculate the ratio of this progress with and without stragglers.

Fig. 6.4a shows that both BSP and SSP are sensitive to stragglers. As long as there exist stragglers in the system, the progress of both barrier methods slows down significantly. Since SSP allows for certain amount of staleness, it performs slightly better than BSP. As a comparison, both pBSP and pSSP are very similar to ASP by using the `sampling` primitive. The step progress degradation in a system with stragglers is close to sub-linear. This observation is expected. Recall that the slow nodes are 2 times slower. With more

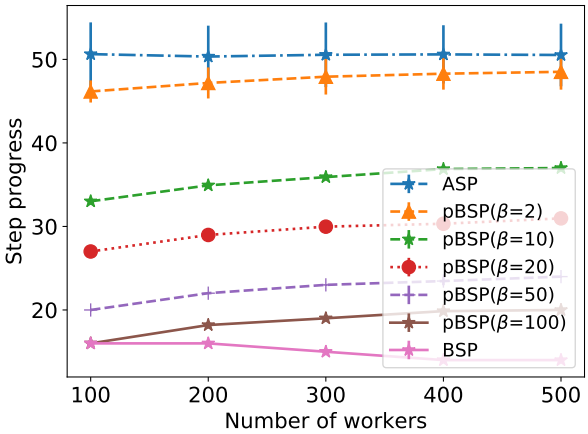


(a) Normalised average speed as a function of percentage of the slow nodes from 0% to 30%.

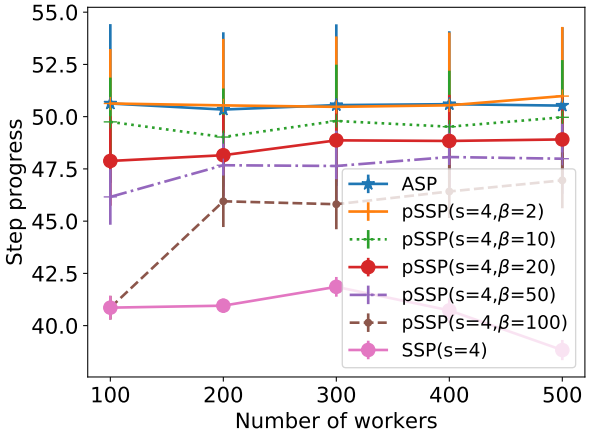


(b) Keep 5% slow nodes, slowness increased from 2x to 16x slow.

Figure 6.4: Stragglers impact system performance. Probabilistic synchronisation control by sampling primitive is able to mitigate such impact.



(a) The step progress of pBSP compared with BSP.



(b) The step progress of pSSP compared with SSP.

Figure 6.5: Scalability of PSP with regard to step progress.

slow workers being added in the system, its step progress performance will approach 50% of the original system.

In another experiment, the system keeps having 5% slow workers, and the "slowness" is increased from no slowing down to 16 times slower. Fig. 6.4b plots the progress distribution as a function of slowness. This figure again shows that both BSP and SSP are dominated by the stragglers, where a small amount of stragglers is able to influence the system step performance greatly. Meanwhile, pBSP, pSSP, and ASP are less influenced by stragglers. Again, note that, compared to ASP, both pBSP and pSSP are more robust with regards to model error.

Scalability

After investigating the factor of stragglers, in this section, I am going to check how well the barrier control methods scale with the number of workers. I compare both SSP/pSSP and BSP/pBSP, and increase the number of workers from 100 to 500. Here the ASP is only used as a benchmark since it does not require any synchronisation and not affected by system scale. The step progress performance of BSP and pBSP is shown in Fig. 6.5a.

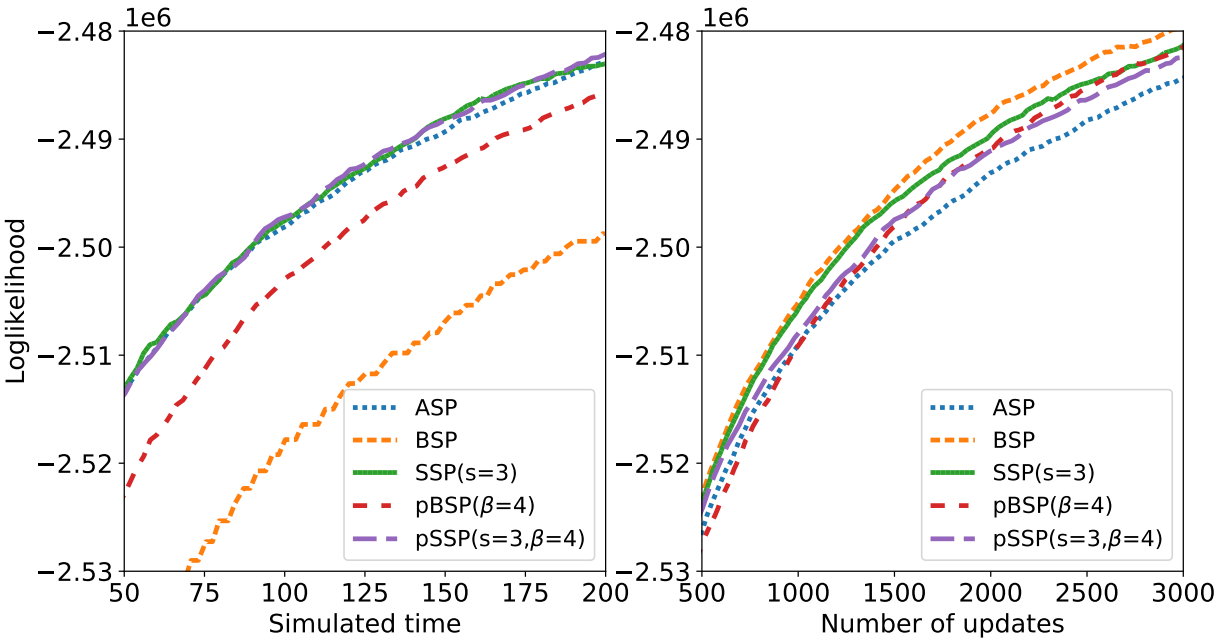


Figure 6.6: Log-likelihood as a function of time and number of updates in the LDA training using different barriers (32 nodes in total).

When network size increases, the progress of BSP decreases, since one single slow node can slow down the progress of all the others. On the other hand, by sampling in a larger body, pBSP increases the step progress with only small variation. Fig. 6.5b shows a similar story in comparing SSP and pSSP. When network size grows, SSP becomes slower, but pSSP is unaffected or slightly faster. During this process, the synchronisation requirement becomes tighter and the step progress gets smaller, also with smaller variation.

In summary, the evaluation results in this section shows that the `sampling` primitive allows users to cover the whole spectrum of synchronisation controls from the most strict, BSP, to the least strict, ASP, without requiring any single node to maintain the global state. Furthermore, the PSP can achieve good step progress performance even with stragglers in the system, and it also scales well.

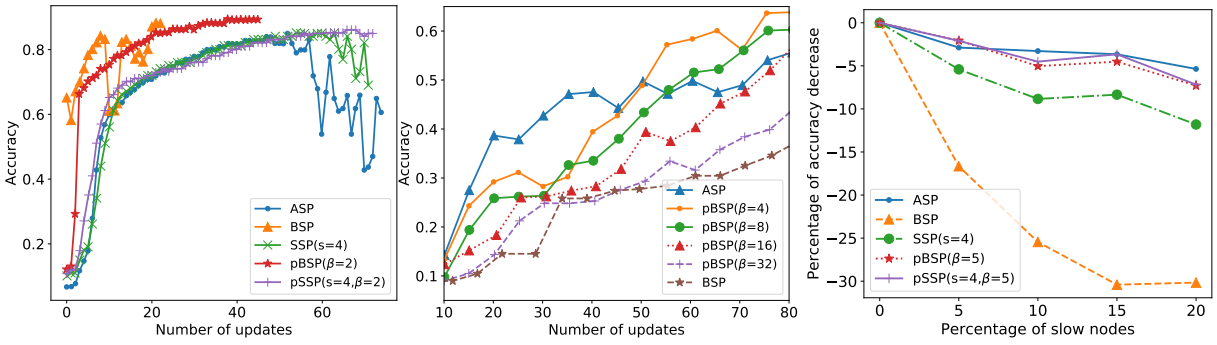
6.2.2 Accuracy

Besides the fast progress in step, a more attractive feature of PSP is that it can achieve higher accuracy. To demonstrate this point, I use three set of experiments: LDA topic modelling, MNIST-based DNN training, and Matrix Factorisation. All three are commonly used in evaluations of distributed learning and barrier control methods in existing literature [32, 92].

Latent Dirichlet Allocation

First, I compare the performance of five barrier methods in training a LDA topic model. In this experiment I set sample size to 4, and the staleness parameters of SSP and pSSP are both set to 3. In topic modelling, log-likelihood is usually used to measure how well the topic model fits documents. I use it to denote the training performance (the higher the better). The number of workers is set to 32.

I use both simulated time and number of updates during training as x-axis to observe the same simulation result. The left side of Fig. 6.6 shows that ASP, SSP, and pSSP achieve better model log-likelihood at a fixed time than pBSP and BSP. However, high



(a) (b) (c)

Figure 6.7: (a) MNIST training using 6 workers. (b) Compare trained model accuracy of a MNIST-based DNN using different barriers, with various sample size (64 nodes in total). (c) Decreased model accuracy as a function of percentage of slow nodes from 0% to 20%.

log-likelihood of ASP is achieved at the price of much more updates than the others. A more precise description of the quality of update is to use the number of updates as x-axis, as shown in the right side of Fig. 6.6. Given the same number of updates, BSP achieves the highest log-likelihood, while that of ASP is the lowest. As expected, the update's quality of pBSP and pSSP is lower than that of BSP and SSP respectively, but with much faster step progress. As a result, they achieve similar or better log-likelihood. On the left side of Fig. 6.6, pBSP achieves close-to-optimal log-likelihood, much better than BSP. pSSP performs similar as SSP, and at the same time benefits from a much lower communication overhead. This evaluation has shown that, PSP can achieve both good update quality and fast update, and thus achieves good model accuracy.

Deep Neural Network

Next, I evaluate barrier control methods in training a deep neural network using the MNIST dataset. I use inference accuracy on the test dataset as measurement of performance.

In Fig. 6.7a, I conduct an experiment using 6 worker nodes in the Parameter Server framework provided by the Actor system. I run the training process for a fixed amount of time, and observe the performance of barrier methods given the same number of updates. The result agrees with that in Fig. 6.6. It shows that BSP achieves the highest model accuracy with the least of number of updates, while SSP and ASP achieve lower efficiency. With training progressing, both methods show a tendency to diverge. Due to the random nature of SGD, sometimes the accumulation of updates may decreases accuracy, as shown in the BSP line. This effect is later mitigated by the following updates. By applying sampling, pBSP and pSSP avoid accumulating too much updates towards one direction and thus achieve smoother accuracy progress than BSP and SSP.

I further conduct evaluations of larger scale using the simulation platform. In Fig. 6.7b, I compare the performance of applying ASP, BSP, and pBSP, using 64 workers. By applying the most inconsistent ASP, the model accuracy changes with large jitter and, as a result, the accuracy of model cannot be guaranteed. On the other hand, by applying BSP, the accuracy grows steadily, but is limited by small numbers of updates. By applying sampling to BSP, pBSP can effectively explore the tuning space in between. With a smaller sampling size such as 4, there is still small scale of jitters, but the number of updates is also close to that of ASP, so the overall accuracy keeps high. The jitter quickly diminishes when there is a minor increase (e.g. from 4 to 8) in sample size. By changing

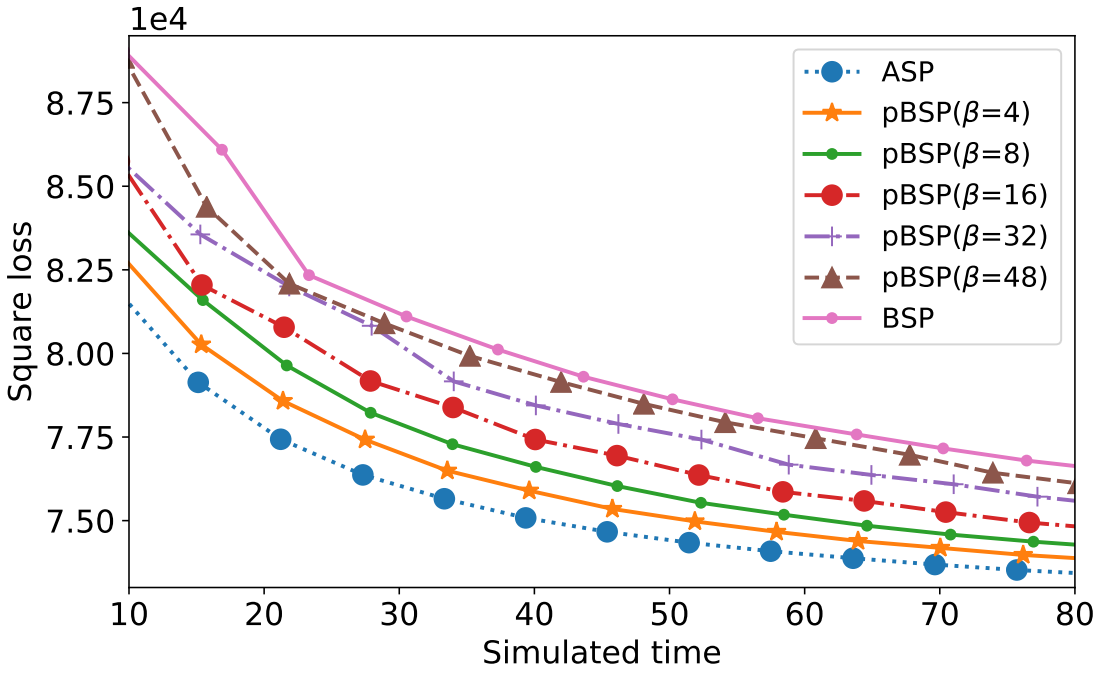


Figure 6.8: Square loss against time for pBSP in Matrix Factorisation with different sample size (64 nodes in total).

sample size, I can get the point at which the training quality (larger increase in accuracy and smaller jitter means better quality) and number of updates arrive at a good balance.

Straggler is an important factor that affects the model accuracy, especially in a federated learning scenario, where stragglers are common in workers [30]. For BSP and SSP, stragglers slow down the progress and delay the convergence; for ASP and others, stragglers may submit outdated updates which introduce noise and destroy previous updates to diverge the learning. Fig. 6.7c plots the decrease of accuracy (due to stragglers) as a function of the percentage of stragglers in the system. I first measure the model accuracy after training for 60 simulated seconds when there are no stragglers, and then I increase the percentage of stragglers step by step as before and measure the model accuracy again after running for the same period of time. A straggler is set to be 5 times slower. The results show that BSP is the most sensitive to stragglers regarding model accuracy, dropping 30% accuracy when 15% of workers are stragglers in the system. In a strict barrier such as BSP, all nodes have to wait for stragglers to finish work, which leads to fewer updates. Both ASP and PSP stay insensitive to stragglers since they are faster in convergence and slowed step progress does not lead to change in accuracy. Using sampling also helps to avoid stragglers in PSP.

Matrix Factorisation

Matrix Factorisation (MF) is a key technique in analysing the latent relationship between two entities, such as the recommendation systems [123]. It differs from DNN training in that it updates the parameters sparsely, and the computational load on each worker is lightweight compared to that of DNN. In this section I examine how PSP performs in an application with different characteristics.

I use 64 workers in total. For each training iteration, a worker uses a random batch of its local data. The training batch size is 10,000, about 1% of total training data. I use the squared prediction error on test data as model accuracy metric (the lower the better).

Unlike the DNN model, the sparsity of the MF model means that the model accuracy

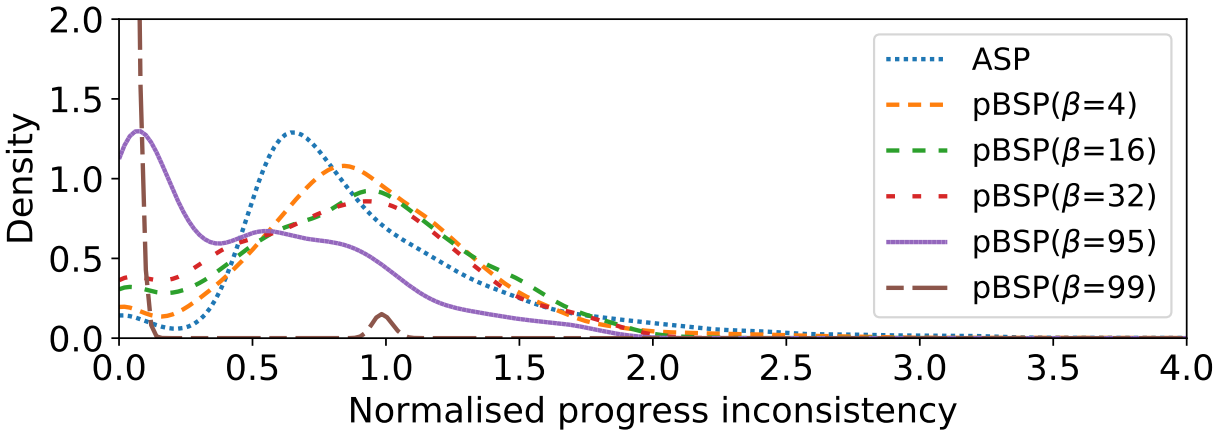


Figure 6.9: PSP provides a wide tuneable space regarding progress inconsistency, changing from BSP to ASP.

is less affected by inconsistency than in DNN, and thus an asynchronous method has advantage in performance. In Fig. 6.8, I vary the sample size of pBSP. A full sampling of pBSP equals BSP; when sample size becomes smaller, the convergence behaviour gets closer to that of ASP.

Note that even though all three applications have different features with regard to optimisation algorithms, data, and parameter sparsity, etc., PSP can easily fit into these systems and be tuned to get good performance by changing the sample size parameter β.

6.2.3 Progress Inconsistency

As stated previously, progress inconsistency means that when one worker decides to push its update, its information about the global parameters is always outdated. All the workers and the server hold their own version of progress vectors, the length of which is equal to that of worker numbers. Every time a node pushes its update to the server, its progress inconsistency value is calculated as the sum of the difference between its own progress vector and that of the server. During the whole process, I collect this inconsistency value of each node at each of its steps. This metric indicates the training accuracy of using a given barrier, excluding the influence of other factors such as the specific application or the hyper-parameters used. If not mentioned otherwise, I set the default number of nodes to 100.

Sample Size

First, I show the distribution of progress inconsistency across a whole training process for different barrier methods. The inconsistency value is normalised by the network size. I plot the distribution of all these inconsistency values in Fig. 6.9. For each barrier method, I estimate its probability density function with kernel density estimation.

Specifically, I investigate the impact of changing parameters of pBSP. ASP has the most widely spread inconsistency distribution. When sample size increases slightly from 0 to 4, the inconsistency behaviour shifts away from ASP, more concentrated around the average value. Keeping increasing the sample size, the density curve moves closer to 0, which means it behaves more like BSP. Note the gap between $p = 95$ and $p = 99$, where the small decrease of sample size leads to large change of inconsistency distribution. This result visualises the tuning process from ASP to BSP with regard to progress inconsistency by changing sample size. The result is similar for SSP and pSSP.

Statistics of Inconsistency Distribution To further investigate the impact of stragglers and network sizes, I need to study some statistics of the distribution of inconsistency. Fig. 6.9 shows how the distribution of progress inconsistency shifts using different barriers. As is shown, sampling provides a full spectrum of tuning space by simply changing the sample size. In this figure, the *mean* value of a curve is a rough estimation of inconsistency. For example, some applications only need to know if the inconsistency is proportional to the number of workers. The *standard deviation* indicates the degree of noise in the system. Varying sample size has different impact on these statistics in different scenarios. Good parameters can thus be decided according to use case in a flexible way. In the rest of this chapter, I will use both the mean and deviation of the progress inconsistency as metrics in measuring the performance of barriers.

Scalability

Next, I investigate the relationship between number of nodes and progress inconsistency. All simulations run for 100 seconds, and I increase worker numbers from 50 to 500. I measure the average and variance of inconsistency, both normalised by the number of workers.

The average inconsistency of ASP is mostly unaffected by size. With smaller sample size, that of pBSP becomes close to ASP, but note that only the initial increase of network size has big impact. With sample size fixed and network size growing, the average inconsistency grows sub-linearly, which is an ideal property. As to the standard deviation values of pBSP, they mostly keep stable regardless of network size.

Fig. 6.10a shows the statistics of progress inconsistency in pBSP. The average inconsistency of ASP is mostly unaffected by size. With smaller sample size, that of pBSP becomes close to ASP, but note that only the initial increase of network size has a big impact. With sample size fixed and network size growing, the average inconsistency grows sub-linearly, which is an ideal property. As to the standard deviation values of pBSP, they mostly keep stable regardless of network size. Larger sample size only has marginal effect in lowering the variance. Note that a small number of samples (e.g. $\beta = 5$) can achieve lower variance. This can be explained by observing that, with more sampled nodes, the probability of including in stragglers also increases. So pBSP can achieve a small sample size with regard to inconsistency variance.

BSP always has a normalised inconsistency of 0 because, for each node, its own view is always synchronised with that of the server. For the other barriers, their average inconsistencies are all close to 1. This value is mostly affected by the fact that a node pushes one update to server when it finishes processing. Recall that in my simulation, the calculation time of each node is similar, generated according to an exponential distribution. Therefore, when one node proceeds, it expects all the other nodes to also proceed by one iteration on average, thus resulting in a normalised inconsistency of 1.

Fig. 6.11a tells a similar story for pSSP. The variance in inconsistency grows slightly with a larger number of workers, but it keeps being in a small range. The average progress inconsistency is close to that of ASP, and the impact of sampling size is limited. The average inconsistency of SSP tends to decrease with more workers. However, the cost of decreased inconsistency is that with larger number of workers the progress is more likely to be slowed down by stragglers. According to these observations, for PSP, both the average and variance of its progress inconsistency grow sub-linearly towards a certain limit with increasing network size, limited by that of ASP and BSP/SSP.

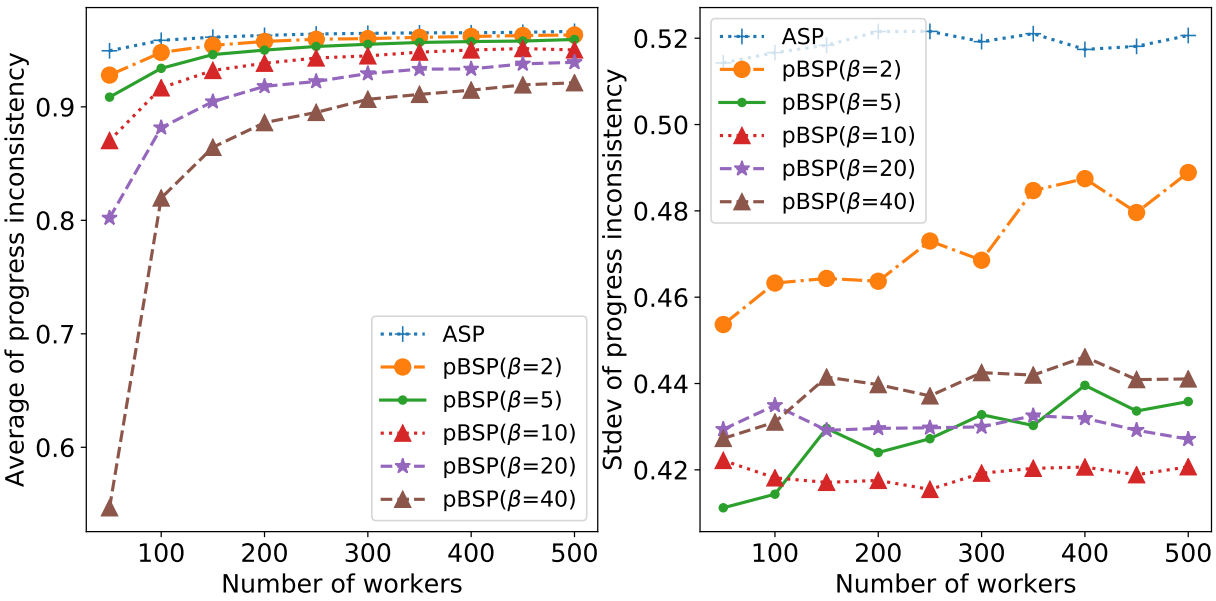


Figure 6.10: Statistics of progress inconsistency in pBSP.

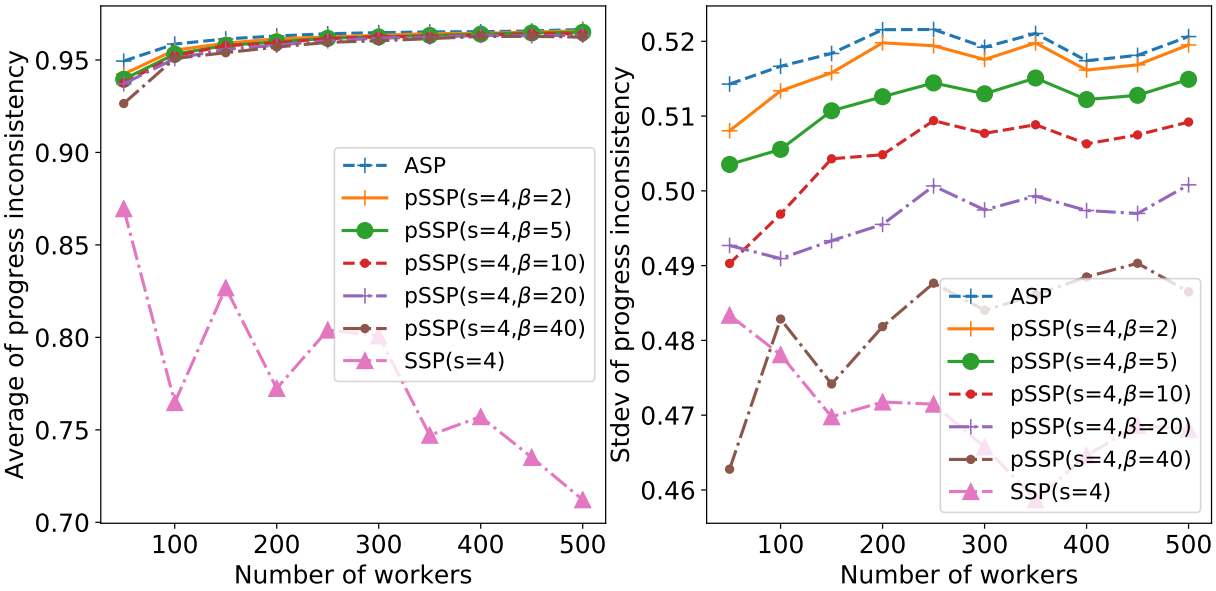


Figure 6.11: Statistics of progress inconsistency in pSSP.

Stragglers

In this part, the impact of stragglers on progress inconsistency was investigated. I set certain percentage of nodes in the system to be stragglers, and they all have the same level of *straggleness*. I fix the straggleness to be 4 (which means that the simulated execution time of each slow node is multiplied by 4), and then increase the percentage of stragglers from 0 to 30%. I observe how the mean and standard deviation of the progress inconsistency change with stragglers, both for pBSP and pSSP.

In Fig. 6.12, the general trend is that more sampling of pBSP leads to smaller mean and deviation, as can be expected; also, the mean value is only slightly affected by the straggler percentage, while the deviation keeps increasing with more stragglers. Note that when sampling percentage is between 5% and 80%, both the mean and deviation of progress inconsistency are very close, which partly explains the effectiveness of sampling.

For pSSP, the results are similar, but with one point to note. In Fig. 6.13, the stragglers initially reduce the mean of inconsistency. It is because an extremely slow node can force

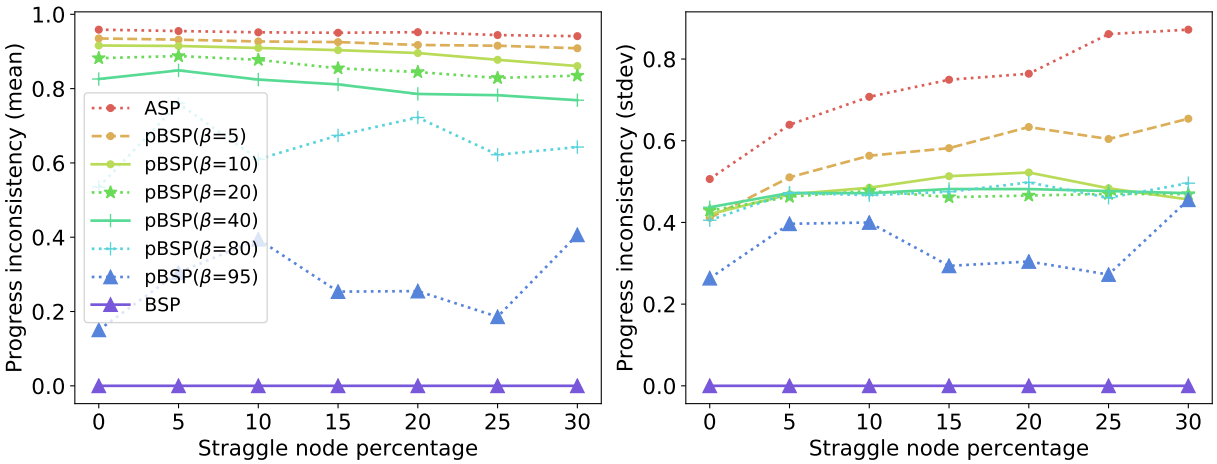
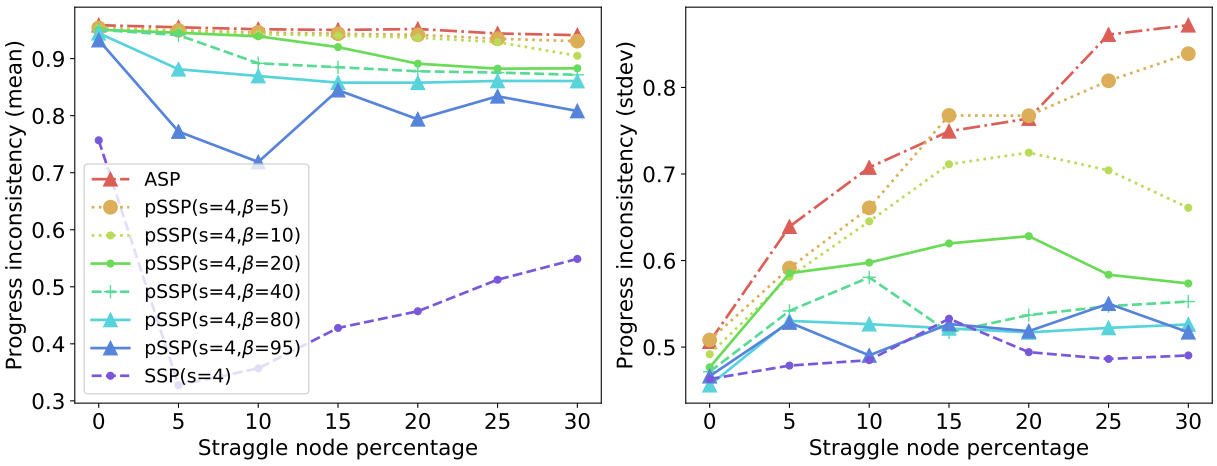


Figure 6.12: Effect of straggle percentage on the statistics of normalised progress inconsistency in pBSP.



(a)

Figure 6.13: Effect of straggle percentage on the statistics of normalised progress inconsistency in pSSP.

all the other nodes to wait for it, leading to a similar effect as BSP, but at the price of slower step progress and larger deviation.

6.2.4 Sequence Inconsistency

The second type of inconsistency is the difference between the real-world "noisy" update sequence and an ideal and orderly "true" sequence. In this case, I observe the inconsistency value directly instead of a whole distribution. I run each experiment for 100 seconds, and measure the number of differences between the true and noisy sequences at a fixed intervals. The aim of this section of evaluation is to show the convergence performance of different barrier methods.

Fig. 6.14a shows the sequence inconsistency number normalised by total number of workers, with sequence length as x axis. It shows that the sequence inconsistency of ASP keeps growing linearly. By using SSP, the inconsistency grows and decreases within a certain bound. Appling sampling to SSP relaxes that bound but, unlike ASP, inconsistencies using pSSP grow sub-linearly with sequence length. BSP is omitted in the figure, since its true and noisy sequence is always the same. pBSP shows a tight bound (about 0.5) even with only 5% sampling, which is better than SSP. In the PSP paper [34], the proof of its

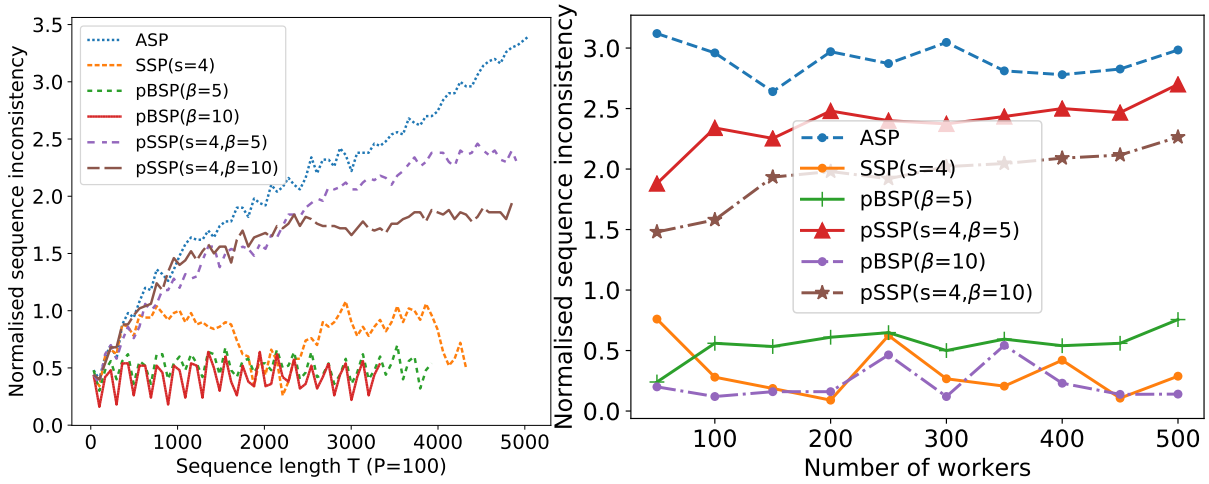
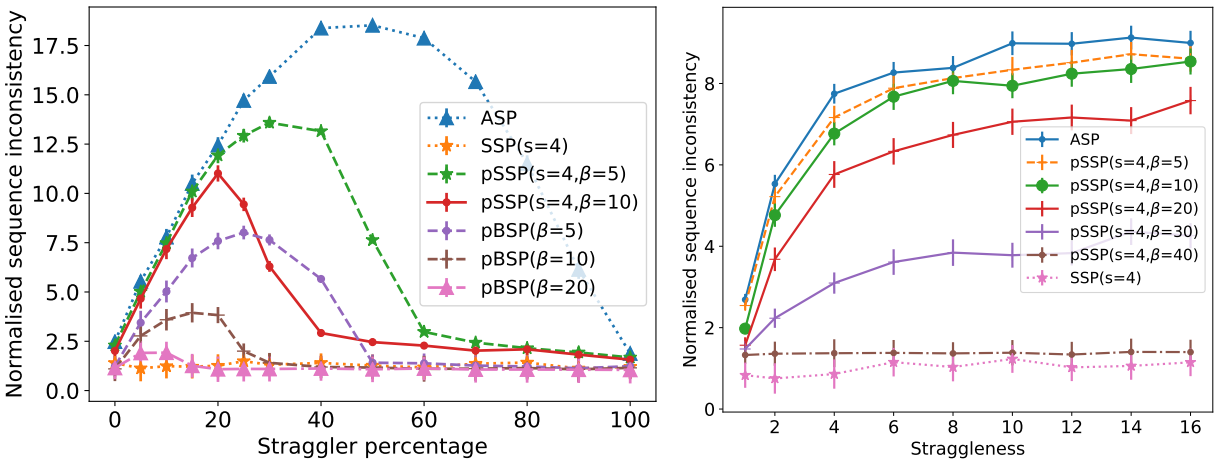


Figure 6.14: (a) Sequence inconsistency; (b) the effect of worker size on converged bound of normalised sequence inconsistency.



(a) The impact of straggler percentage on barrier methods.

(b) The impact of straggleness on pSSP.

Figure 6.15: Straggleness and straggler percentage both affect the sequence inconsistency.

convergence is achieved by proving the average and variance of sequence inconsistency are both probabilistically bounded in theory. Fig. 6.14a demonstrates this point intuitively.

Also note that previous work suggest that the staleness parameter only provides a upper bound on the sequence inconsistency, and the empirical value should be lower. My observation of SSP in Fig. 6.14a confirms his point.

Next, I explore how network size and stragglers affect the sequence inconsistency. For each barrier, I calculate its sequence inconsistency every 30 steps, using the last 10 stabilised values, and then get their average and deviation for one observation on the figures.

In Fig. 6.14b, I increase the network size from 50 to 500. It shows that the network size does not have a large impact on the inconsistency per node, or that sequence inconsistency grows linearly with the network size.

Fig. 6.15a shows the effect of increasing straggler percentage. As the number of stragglers grows, the inconsistency first increases, and then drops back to the original level. This is expected, because when all the nodes are equally slow, the sequence inconsistency should be the same as when all the nodes are equally fast. For ASP, the "turning point" happens when exactly half of the nodes are stragglers. SSP is mostly unaffected

by stragglers because of its bounded staleness. Using PSP, after the turning point, the inconsistency decreases until stable, and this point can be freely chosen as needed. Larger sample size leads to the maximum inconsistency being smaller.

Fig. 6.15b shows tuning the sampling size in pSSP with changing straggleness level. For all the barriers, the straggleness has a diminishing effect on them. Note how the sampling size changes its behaviour from ASP to SSP. The most effective drop in sequence inconsistency happens with around 20%-40% sampling. When more than 40% of nodes are sampled, the sequence inconsistency of pSSP becomes very close to that of SSP.

6.3 Conclusion

Once a computation is deployed to different devices, the question about how various computations collaborate arises. A key element in computation collaboration is the control of synchronisation by using different barrier methods. In this chapter, based on the previous work on Probabilistic Synchronous Parallel (PSP), I conducted a thorough evaluation of PSP by comparing it with existing barrier control methods. I used the traditional metrics such as step progress of nodes and model accuracy, and I also summarised two types of inconsistencies from existing literature as two metrics that can bring new insights into the evaluation of barrier control methods.

Chapter 7

Conclusions

In Chapter 1, I briefly introduced the field of numerical computation. I explained the motivation to optimise computation in a bottom-up approach based on my experience participating in developing a numerical library, and outlined the structure of the remainder of this thesis. In Chapter 2, I gave a detailed introduction into the background material involved in the original work in this thesis. In Chapter 3, I described the low-level architectural design of the Owl library. Focusing on several representative operations, I showed the low-level design of these basic operations and optimisations by comparing with existing NumPy libraries. I also presented a performance tuning module in Owl, to provide tuned performance of operations on different machines. In Chapter 4, I presented the idea of using the Computation Graph, which consists of multiple basic operations, as an intermediate representation to enable computation interoperability. I presented the design and implementation of the TFgraph system, which aims to convert the computation graph from Owl to TensorFlow. Treating a whole computation as a unit, in Chapter 5 I identified two challenges about data analytics on edge devices: service composition and deployment, and then presented the Zoo system that provides a concise Domain-specific Language to address these challenges. Chapter 6 focused on the collaboration of deployed computation. I first introduced several applications of collaboration of computation on multiple nodes, especially the DNN training. I then proposed to use two metrics to evaluate the barrier methods, and conducted a thorough evaluation of existing barrier control methods, especially PSP.

7.1 Future work

In Chapter 3, I presented the AEOS module to tune system parameters on different machines so as to get optimal performance. Currently, the factor I consider is mainly multiprocessing using OpenMP, but the AEOS system should not be limited by only one factor. As I have shown in that chapter, the optimisation of low-level operations involves multiple factors, from vectorisation using SIMD, multiple implementation for specific input size of type, reduction of memory copy, memory read by order, to algorithm redesign, etc. Together they form a large tuning space. These factors can be all incorporated in a parameter tuning module so as to achieve optimal computation performance on different platforms.

I have presented several examples using TFgraph that define computations in Owl and execute them on TensorFlow. However, the system is not yet a mature tool. Operations such as condition and loop are not yet supported in Owl CGraph, but are nevertheless important. Tools such as Python script automatic generation or data conversion should also be provided. Furthermore, current interfacing with TensorFlow relies on mechanisms

that are not standardised. A next step would be to develop interfaces with existing graph standards such as ONNX, which can be supported by many existing numerical systems.

As to the Zoo system, many opportunities exist to improve the implementation. To enable a Gist script to be deployed as service, a user needs to specify the function names and signatures in the configuration, but would be more usable if the signature can be inferred automatically. The serialisation method of using Base64 can also be updated. The current system is limited by language, but the principle of computation composition, static type checking, and deployment should be extended beyond this limitation. Moreover, optimisation of a composed computation at graph level will further improve its performance.

In Chapter 6, I showed the large tuning space in barrier control methods via thorough evaluation. One question remains unanswered, which is how to find suitable parameters for different applications. Like performance tuning in numerical computation, prior knowledge and empirical measurement based tuning could be useful. Machine Learning algorithms can also be applied. PSP provides an effective mechanism to adapt to different application scenarios, but a solution to tune parameters dynamically remains interesting work.

7.2 Final remarks

In this thesis, I presented the optimisation of computation using a bottom-up approach, based on my experience in participating in the development of the numerical library Owl. Basic operation optimisation lies at a low level; a computation graph consists of multiple operations and can be used as intermediate representation to be executed on different hardware accelerators; multiple computations can be composed and deployed on edge devices in different forms; and, finally, the deployed computation need to collaborate with each other to make an efficient application. I believe my techniques could broaden the scope of CS research by providing a full stack perspective from the top level application to the basic performance of a single operation in the numerical library. I hope my work stimulates further collaboration between the research areas of system, machine learning, and scientific computing. I believe that further attention should be given to an application-driven collaboration between these different areas to help the adaptation of numerical optimisation to the ever-changing requirement of numerical applications.

Bibliography

[1] Alex Graves, Abdel-rahman Mohamed, and Geoffrey Hinton. Speech recognition with deep recurrent neural networks. In *2013 IEEE international conference on acoustics, speech and signal processing*, pages 6645–6649. IEEE, 2013.

[2] Leon A Gatys, Alexander S Ecker, and Matthias Bethge. Image style transfer using convolutional neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 2414–2423, 2016.

[3] M Vardhana, N Arunkumar, Sunitha Lasrado, Enas Abdulhay, and Gustavo Ramirez-Gonzalez. Convolutional neural network for bio-medical image segmentation with hardware acceleration. *Cognitive Systems Research*, 50:10–14, 2018.

[4] Yuchi Tian, Kexin Pei, Suman Jana, and Baishakhi Ray. Deeptest: Automated testing of deep-neural-network-driven autonomous cars. In *Proceedings of the 40th international conference on software engineering*, pages 303–314. ACM, 2018.

[5] David A Fournier, Hans J Skaug, Johnoel Ancheta, James Ianelli, Arni Magnusson, Mark N Maunder, Anders Nielsen, and John Sibert. Ad model builder: using automatic differentiation for statistical inference of highly parameterized complex nonlinear models. *Optimization Methods and Software*, 27(2):233–249, 2012.

[6] George Efstathiou, M Davis, SDM White, and CS Frenk. Numerical techniques for large cosmological n-body simulations. *The Astrophysical Journal Supplement Series*, 57:241–260, 1985.

[7] Li Da Xu, Wu He, and Shancang Li. Internet of things in industries: A survey. *IEEE Transactions on industrial informatics*, 10(4):2233–2243, 2014.

[8] Mehdi Mohammadi, Ala Al-Fuqaha, Sameh Sorour, and Mohsen Guizani. Deep learning for iot big data and streaming analytics: A survey. *IEEE Communications Surveys & Tutorials*, 20(4):2923–2960, 2018.

[9] Mary Meeker and Liang Wu. Internet trends report 2018. *Kleiner Perkins*, 5:30, 2018.

[10] Dina Srinivasan. The antitrust case against facebook: A monopolist's journey towards pervasive surveillance in spite of consumers' preference for privacy. *Berkeley Business Law Journal*, 16(1):39, 2019.

[11] Stephen Cass. Taking ai to the edge: Google's tpu now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17, 2019.

[12] Tal Ben-Nun and Torsten Hoefler. Demystifying parallel and distributed deep learning: An in-depth concurrency analysis. *arXiv preprint arXiv:1802.09941*, 2018.

[13] Adam Coates, Brody Huval, Tao Wang, David Wu, Bryan Catanzaro, and Ng Andrew. Deep learning with cots hpc systems. In *International conference on machine learning*, pages 1337–1345, 2013.

[14] Michaël Mathieu, Mikael Henaff, and Yann Lecun. Fast training of convolutional networks through ffts. In *International Conference on Learning Representations (ICLR2014), CBLS, April 2014*, pages http–openreview, 2014.

[15] Andrew Lavin and Scott Gray. Fast algorithms for convolutional neural networks. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 4013–4021, 2016.

[16] R Clint Whaley, Antoine Petitet, and Jack J Dongarra. Automated empirical optimizations of software and the atlas project. *Parallel Computing*, 27(1-2):3–35, 2001.

[17] Intel Software. Intel math kernel library. https://software.intel.com/en-us/mkl, 2019.

[18] David W Walker and Jack J Dongarra. Mpi: a standard message passing interface. *Supercomputer*, 12:56–68, 1996.

[19] Richard M Karp and Rayamond E Miller. Properties of a model for parallel computations: Determinacy, termination, queueing. *SIAM Journal on Applied Mathematics*, 14(6):1390–1411, 1966.

[20] Andreas Griewank et al. On automatic differentiation. *Mathematical Programming: recent developments and applications*, 6(6):83–107, 1989.

[21] 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. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 265–283, Savannah, GA, November 2016. USENIX Association.

[22] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. Automatic differentiation in pytorch. 2017.

[23] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. `https://github.com/onnx/onnx`, 2019.

[24] The Khronos Group. Neural network exchange format. `https://www.khronos.org/nnef`. Accessed: 2019-08-30.

[25] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[26] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. Sand: Towards high-performance serverless computing. In *2018 USENIX Annual Technical Conference (USENIX ATC'18)*, pages 923–935, 2018.

[27] Anil Madhavapeddy, Richard Mortier, Charalampos Rotsos, David Scott, Balraj Singh, Thomas Gazagnaire, Steven Smith, Steven Hand, and Jon Crowcroft. Unikernels: Library operating systems for the cloud. *Acm Sigplan Notices*, 48(4):461–472, 2013.

[28] Yicong Wang, Gustavo de Veciana, Takayuki Shimizu, and Hongsheng Lu. Performance and scaling of collaborative sensing and networking for automated driving applications. In *2018 IEEE International Conference on Communications Workshops (ICC Workshops)*, pages 1–6. IEEE, 2018.

[29] Richard Olaniyan and Muthucumaru Maheswaran. Synchronous scheduling algorithms for edge coordinated internet of things. In *2018 IEEE 2nd International Conference on Fog and Edge Computing (ICFEC)*, pages 1–10. IEEE, 2018.

[30] Keith Bonawitz, Hubert Eichner, Wolfgang Grieskamp, Dzmitry Huba, Alex Ingerman, Vladimir Ivanov, Chloe Kiddon, Jakub Konecny, Stefano Mazzocchi, H Brendan McMahan, et al. Towards federated learning at scale: System design. *arXiv preprint arXiv:1902.01046*, 2019.

[31] Eric P. Xing, Qirong Ho, Pengtao Xie, and Wei Dai. Strategies and principles of distributed machine learning on big data. *CoRR*, 2016.

[32] Qirong Ho, James Cipar, Henggang Cui, Jin Kyu Kim, Seunghak Lee, Philip B. Gibbons, Garth a. Gibson, Gregory R. Ganger, and Eric P. Xing. More effective distributed ml via a stale synchronous parallel parameter server. *NIPS'13: Advanced Neural Information Processing Systems*, 2013.

[33] Feng Niu, Benjamin Recht, Christopher Re, and Stephen J. Wright. Hogwild!: A lock-free approach to parallelizing stochastic gradient descent. *NIPS'11 Proceedings of the 24th International Conference on Neural Information Processing Systems*, pages 693–701, 2011.

[34] Liang Wang, Ben Catterall, and Richard Mortier. Probabilistic synchronous parallel. *arXiv preprint arXiv:1709.07772*, 2017.

[35] Liang Wang. Owl: A general-purpose numerical library in ocaml. *arXiv preprint arXiv:1707.09616*, 2017.

[36] Roy Frostig, Matthew James Johnson, and Chris Leary. Compiling machine learning programs via high-level tracing. *Systems for Machine Learning*, 2018.

[37] Todd A Anderson, Hai Liu, Lindsey Kuper, Ehsan Totoni, Jan Vitek, and Tatiana Shpeisman. Parallelizing julia with a non-invasive dsl. In *31st European Conference on Object-Oriented Programming (ECOOP 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[38] Iain Dunning, Joey Huchette, and Miles Lubin. Jump: A modeling language for mathematical optimization. *SIAM Review*, 59(2):295–320, 2017.

[39] Jianxin Zhao, Tudor Tiplea, Richard Mortier, Jon Crowcroft, and Liang Wang. Data analytics service composition and deployment on edge devices. In *Proceedings of the 2018 Workshop on Big Data Analytics and Machine Learning for Data Communication Networks*, pages 27–32. ACM, 2018.

[40] Netlib. Blas (basic linear algebra subprograms). https://www.netlib.org/blas/, 2017.

[41] Netlib. Lapack–linear algebra package. https://www.netlib.org/lapack/, 2019.

[42] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691. IEEE, 2012.

[43] Gaël Guennebaud, Benoît Jacob, et al. Eigen library. http://eigen.tuxfamily.org, 2010.

[44] Jeff Bezanson, Alan Edelman, Stefan Karpinski, and Viral B Shah. Julia: A fresh approach to numerical computing. *SIAM review*, 59(1):65–98, 2017.

[45] Stéfan van der Walt, S Chris Colbert, and Gael Varoquaux. The numpy array: a structure for efficient numerical computation. *Computing in Science & Engineering*, 13(2):22–30, 2011.

[46] 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. *arXiv preprint arXiv:1512.01274*, 2015.

[47] François Chollet et al. Keras. https://keras.io, 2015.

[48] Chris Lomont. Introduction to intel advanced vector extensions. http://shorturl.at/chsW9, 2011.

[49] Matthias Boettcher, Bashir M Al-Hashimi, Mbou Eyole, Giacomo Gabrielli, and Alastair Reid. Advanced simd: Extending the reach of contemporary simd architectures. In *2014 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–4. IEEE, 2014.

[50] Leonardo Dagum and Ramesh Menon. Openmp: An industry-standard api for shared-memory programming. *Computing in Science & Engineering*, pages 46–55, 1998.

[51] Asim YarKhan, Jakub Kurzak, Piotr Luszczek, and Jack Dongarra. Porting the plasma numerical library to the openmp standard. *International Journal of Parallel Programming*, 45(3):612–633, 2017.

[52] Niranjan Hasabnis. Auto-tuning tensorflow threading model for cpu backend. In *2018 IEEE/ACM Machine Learning in HPC Environments (MLHPC)*, pages 14–25. IEEE, 2018.

[53] Jeff Bezanson, Jiahao Chen, Benjamin Chung, Stefan Karpinski, Viral B Shah, Jan Vitek, and Lionel Zoubritzky. Julia: dynamism and performance reconciled by design. *Proceedings of the ACM on Programming Languages*, 2(OOPSLA):120, 2018.

[54] Mark Hoemmen, Jayesh Badwaik, Matthieu Brucher, and John Michopoulos. P1417r0: Historical lessons for c++ linear algebra library standardization. 2019.

[55] Srinivas Chellappa, Franz Franchetti, and Markus Püschel. How to write fast numerical code: A small introduction. In *International Summer School on Generative and Transformational Techniques in Software Engineering*, pages 196–259. Springer, 2007.

[56] Christian Szegedy, Vincent Vanhoucke, Sergey Ioffe, Jonathon Shlens, and Zbigniew Wojna. Rethinking the inception architecture for computer vision. *CoRR*, 2015.

[57] Marcello Seri and Ta-Chu Kao. Owlde: making odes first-class owl citizens. *ICFP OCaml 2019*, 2019.

[58] Jianxin Zhao. Executing owl computation on gpu and tpu. *ICFP OCaml 2019*, 2019.

[59] Liang Wang. Owl: A general-purpose numerical library in ocaml. *CoRR*, abs/1707.09616, 2017.

[60] Pierre Vandenhove. Functional design of computation graph. *arXiv preprint arXiv:1812.03770*, 2018.

[61] Norman P Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, et al. In-datacenter performance analysis of a tensor processing unit. In *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, pages 1–12. IEEE, 2017.

[62] Keno Fischer and Elliot Saba. Automatic full compilation of julia programs and ml models to cloud tpus. *arXiv preprint arXiv:1810.09868*, 2018.

[63] PyTorch. PyTorch XLA, Github. `https://github.com/pytorch/xla/`, 2019. [Online; accessed Aug-2019].

[64] Tianqi Chen, Thierry Moreau, Ziheng Jiang, Lianmin Zheng, Eddie Yan, Haichen Shen, Meghan Cowan, Leyuan Wang, Yuwei Hu, Luis Ceze, et al. Tvm: An automated end-to-end optimizing compiler for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI'18)*, pages 578–594, 2018.

[65] Jonathan Ragan-Kelley, Connelly Barnes, Andrew Adams, Sylvain Paris, Frédo Durand, and Saman Amarasinghe. Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines. In *Acm Sigplan Notices*, volume 48, pages 519–530. ACM, 2013.

[66] Nadav Rotem, Jordan Fix, Saleem Abdulrasool, Garret Catron, Summer Deng, Roman Dzhabarov, Nick Gibson, James Hegeman, Meghan Lele, Roman Levenstein, et al. Glow: Graph lowering compiler techniques for neural networks. *arXiv preprint arXiv:1805.00907*, 2018.

[67] Chris Lattner and Vikram Adve. Llvm: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the international symposium on Code generation and optimization: feedback-directed and runtime optimization*, page 75. IEEE Computer Society, 2004.

[68] Scott Cyphers, Arjun K Bansal, Anahita Bhiwandiwalla, Jayaram Bobba, Matthew Brookhart, Avijit Chakraborty, Will Constable, Christian Convey, Leona Cook, Omar Kanawi, et al. Intel ngraph: An intermediate representation, compiler, and executor for deep learning. *arXiv preprint arXiv:1801.08058*, 2018.

[69] Linpeng Tang, Yida Wang, Theodore L Willke, and Kai Li. Scheduling computation graphs of deep learning models on manycore cpus. *arXiv preprint arXiv:1807.09667*, 2018.

[70] Richard Wei, Lane Schwartz, and Vikram Adve. Dlvm: A modern compiler infrastructure for deep learning systems. *arXiv preprint arXiv:1711.03016*, 2017.

[71] Michael Schaarschmidt, Sven Mika, Kai Fricke, and Eiko Yoneki. Rlgraph: Flexible computation graphs for deep reinforcement learning. *arXiv preprint arXiv:1810.09028*, 2018.

[72] Berkeley Vision and Learning Center. Caffe Model Zoo. `https://github.com/BVLC/caffe/wiki/Model-Zoo`, 2017. [Online; accessed Mar-2019].

[73] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[74] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182. ACM, 2014.

[75] Seldon Technologies Ltd. Seldon Core. `https://github.com/SeldonIO/seldon-core`, 2018. [Online; accessed Aug-2019].

[76] Amazon. AWS Lambda. `https://aws.amazon.com/lambda/`, 2016. [Online; accessed Aug-2019].

[77] Microsoft. Azure Functions. `https://azure.microsoft.com/en-gb/services/functions/`, 2017. [Online; accessed Aug-2019].

[78] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless computation with openlambda. In *8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16)*, 2016.

[79] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, pages 218–233. ACM, 2017.

[80] Tytus Kurek. Unikernel network functions: A journey beyond the containers. *IEEE Communications Magazine*, 57(12):15–19, 2019.

[81] Gu-Yeon Wei, David Brooks, et al. Benchmarking tpu, gpu, and cpu platforms for deep learning. *arXiv preprint arXiv:1907.10701*, 2019.

[82] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *OSDI: Sixth Symposium on Operating System Design and Implementation*, 2004.

[83] Alekh Agarwal, Olivier Chapelle, Miroslav Dudík, and John Langford. A reliable effective terascale linear learning system. *The Journal of Machine Learning Research*, 15(1):1111–1133, 2014.

[84] Mu Li, David G. Andersen, Jun Woo Park, Alexander J. Smola, Amr Ahmed, Vanja Josifovski, James Long, Eugene J. Shekita, and Bor-Yiing Su. Scaling distributed machine learning with the parameter server. In *11th USENIX Symposium on Operating Systems Design and Implementation (OSDI 14)*, pages 583–598, Broomfield, CO, 2014. USENIX Association.

[85] Róbert Ormándi, István Hegedűs, and Márk Jelasity. Gossip learning with linear models on fully distributed data. *Concurrency and Computation: Practice and Experience*, 25(4):556–571, 2013.

[86] Qiang Yang, Yang Liu, Tianjian Chen, and Yongxin Tong. Federated machine learning: Concept and applications. *ACM Transactions on Intelligent Systems and Technology (TIST)*, 10(2):12, 2019.

[87] Henggang Cui, James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Abhimanu Kumar, Jinliang Wei, Wei Dai, Gregory R Ganger, Phillip B Gibbons, et al. Exploiting bounded staleness to speed up big data analytics. In *2014 USENIX Annual Technical Conference (USENIX ATC'14)*, pages 37–48, 2014.

[88] Xiangru Lian, Yijun Huang, Yuncheng Li, and Ji Liu. Asynchronous parallel stochastic gradient for nonconvex optimization. In *Advances in Neural Information Processing Systems*, pages 2737–2745, 2015.

[89] Christopher M De Sa, Ce Zhang, Kunle Olukotun, and Christopher Ré. Taming the wild: A unified analysis of hogwild-style algorithms. In *Advances in neural information processing systems*, pages 2674–2682, 2015.

[90] Shuxin Zheng, Qi Meng, Taifeng Wang, Wei Chen, Nenghai Yu, Zhi-Ming Ma, and Tie-Yan Liu. Asynchronous stochastic gradient descent with delay compensation. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 4120–4129. JMLR. org, 2017.

[91] Kevin Hsieh, Aaron Harlap, Nandita Vijaykumar, Dimitris Konomis, Gregory R Ganger, Phillip B Gibbons, and Onur Mutlu. Gaia: Geo-distributed machine learning approaching lan speeds. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI'17)*, pages 629–647, 2017.

[92] Wei Dai, Abhimanu Kumar, Jinliang Wei, Qirong Ho, Garth Gibson, and Eric P Xing. High-performance distributed ml at scale through parameter server consistency models. In *Twenty-Ninth AAAI Conference on Artificial Intelligence*, 2015.

[93] Wei Zhang, Suyog Gupta, Xiangru Lian, and Ji Liu. Staleness-aware async-sgd for distributed deep learning. *arXiv preprint arXiv:1511.05950*, 2015.

[94] Jianmin Chen, Xinghao Pan, Rajat Monga, Samy Bengio, and Rafal Jozefowicz. Revisiting distributed synchronous sgd. *arXiv preprint arXiv:1604.00981*, 2016.

[95] James Cipar, Qirong Ho, Jin Kyu Kim, Seunghak Lee, Gregory R. Ganger, Garth Gibson, Kimberly Keeton, and Eric Xing. Solving the straggler problem with bounded staleness. *HotOS Usenix*, 2013.

[96] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 77–778, 2016.

[97] Karen Simonyan and Andrew Zisserman. Very deep convolutional networks for large-scale image recognition. *CoRR*, abs/1409.1556, 2014.

[98] Forrest N Iandola, Song Han, Matthew W Moskewicz, Khalid Ashraf, William J Dally, and Kurt Keutzer. Squeezenet: Alexnet-level accuracy with 50x fewer parameters and¡ 0.5 mb model size. *arXiv preprint arXiv:1602.07360*, 2016.

[99] Logan Engstrom. Fast style transfer. `https://github.com/lengstrom/fast-style-transfer/`, 2016.

[100] Philip J Basford, Steven J Johnston, Colin S Perkins, Tony Garnock-Jones, Fung Po Tso, Dimitrios Pezaros, Robert D Mullins, Eiko Yoneki, Jeremy Singer, and Simon J Cox. Performance analysis of single board computer clusters. *Future Generation Computer Systems*, 102:278–291, 2020.

[101] Giovanni Garberoglio. Avx implementation of math functions. `http://software-lisc.fbk.eu/avx_mathfun/avx_mathfun.h`, 2012. Accessed Sep-2019.

[102] Sergey Ioffe and Christian Szegedy. Batch normalization: Accelerating deep network training by reducing internal covariate shift. *CoRR*, abs/1502.03167, 2015.

[103] Sharan Chetlur, Cliff Woolley, Philippe Vandermersch, Jonathan Cohen, John Tran, Bryan Catanzaro, and Evan Shelhamer. cudnn: Efficient primitives for deep learning. *arXiv preprint arXiv:1410.0759*, 2014.

[104] Kazushige Goto and Robert A Geijn. Anatomy of high-performance matrix multiplication. *ACM Transactions on Mathematical Software (TOMS)*, 34(3):12, 2008.

[105] Kumar Chellapilla, Sidd Puri, and Patrice Simard. High performance convolutional neural networks for document processing. In *Tenth International Workshop on Frontiers in Handwriting Recognition*. Suvisoft, 2006.

[106] Minsik Cho and Daniel Brand. MEC: memory-efficient convolution for deep neural network. *CoRR*, abs/1706.06873, 2017.

[107] Richard Mortier, Jianxin Zhao, Jon Crowcroft, Liang Wang, Qi Li, Hamed Haddadi, Yousef Amar, et al. Personal data management with the databox: What's inside the box? In *Proceedings of the 2016 ACM Workshop on Cloud-Assisted Networking*, pages 49–54. ACM, 2016.

[108] Movidius. Movidius neural compute stick. `https://developer.movidius.com/`, 2017. Accessed Nov. 11, 2017.

[109] Liang Wang, Sotiris Tasoulis, Teemu Roos, and Jussi Kangasharju. Kvasir: Scalable provision of semantically relevant web content on big data framework. *IEEE Transactions on Big Data*, 2(3):219–233, 2016.

[110] Huawei-News. HUAWEI Reveals the Future of Mobile AI at IFA 2017. `http://consumer.huawei.com/en/press/news/2017/ifa2017-kirin970/`, 2017. [Online; accessed Nov-2017].

[111] Sandra Servia Rodríguez, Liang Wang, Jianxin R. Zhao, Richard Mortier, and Hamed Haddadi. Privacy-preserving personal model training. *Internet-of-Things Design and Implementation (IoTDI), The 3rd ACM/IEEE International Conference on*, 2018.

[112] Schahram Dustdar and Wolfgang Schreiner. A survey on web services composition. *International journal of web and grid services*, 1(1):1–30, 2005.

[113] Ioana Baldini, Perry Cheng, Stephen J Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, pages 89–103, 2017.

[114] MxNet. MxNet Model Zoo. `https://gluon-cv.mxnet.io/model_zoo/index.html`, 2017. [Online; accessed Mar-2019].

[115] Fujun Luan, Sylvain Paris, Eli Shechtman, and Kavita Bala. Deep photo style transfer. *CoRR*, abs/1703.07511, 2017.

[116] Yann LeCun, Léon Bottou, Yoshua Bengio, and Patrick Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[117] Google. Google cloud vision api. `https://cloud.google.com/vision`, 2017. Accessed Nov. 11, 2017.

[118] Tevfik Bultan, Jianwen Su, and Xiang Fu. Analyzing conversations of web services. *IEEE Internet Computing*, 10(1):18–25, 2006.

[119] Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial nets. In *Advances in neural information processing systems*, pages 2672–2680, 2014.

[120] Kohei Honda, Vasco T Vasconcelos, and Makoto Kubo. Language primitives and type discipline for structured communication-based programming. In *European Symposium on Programming*, pages 122–138. Springer, 1998.

[121] Jiawei Jiang, Bin Cui, Ce Zhang, and Lele Yu. Heterogeneity-aware distributed parameter servers. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 463–478. ACM, 2017.

[122] Thomas L Griffiths and Mark Steyvers. Finding scientific topics. *Proceedings of the National academy of Sciences*, 101(suppl 1):5228–5235, 2004.

[123] Yehuda Koren. Factorization meets the neighborhood: a multifaceted collaborative filtering model. In *Proceedings of the 14th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 426–434. ACM, 2008.

[124] Han Xiao and Thomas Stibor. Efficient collapsed gibbs sampling for latent dirichlet allocation. In *Proceedings of 2nd Asian Conference on Machine Learning*, pages 63–78, 2010.

[125] Yehuda Koren, Robert Bell, and Chris Volinsky. Matrix factorization techniques for recommender systems. *Computer*, pages 30–37, 2009.

[126] F Maxwell Harper and Joseph A Konstan. The movielens datasets: History and context. *Acm transactions on interactive intelligent systems (tiis)*, 5(4):19, 2016.