Deep Learning on a Low Power GPU

Perry Gibson

Undergraduate Dissertation
School of Informatics
University of Edinburgh

2019

Abstract

This report details the design, implementation, and evaulation of “Orpheus”, a tool to benchmark the inference of deep learning systems on heterogeneous devices, and enable fast iteration speed in researching the intersections of layers of the deep learning inference stack.

Motivated by difficulties experienced in previous efforts adapting existing production neural network frameworks, this work shows that there is a need for a specialised tool to investigate across-stack optimisations of neural network inference. Complex software dependencies means that deploying existing production systems to heterogeneous devices can be prohibitively difficult. Fully-featured training infrastructure, and other features, many of which are interdependent make it difficult to investigate the effects of even trivial changes to the inference-only functionality and performance.

Orpheus reproduces only the logic needed to run inference on models, with a modular design which encourages the development and mutual bench-marking of alternative implementations of each component, including changing the data format and linear algebra backend library.

Acknowledgements

I would like to thank Dr José Cano for supervising this project, providing guidance and insights, and Dr Valentin Radu for his suggestions during the development process.

Additional thanks go to my family, friends, and loved ones, for their unconditional support.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

Deep learning systems are continuing their growth in becoming a key part of modern computing workloads, and are seeing deployment in a wide range of hardware and software environments. It has been argued that some of the most promising applications of deep learning systems will be on low powered edge devices, which have a wide range of hardware and software specifications. This makes it difficult to deploy to these devices, as developers must consider the full range of environments their systems may be utilised in, and tune appropriately.

Improving the accuracy of predictions is one of the key metrics in accessing a model's performance. However there are other considerations, especially in embedded systems where resources are more constrained. Inference time, memory footprint, and energy consumption are all factors that designers need to be aware of. Ideally, the particulars of how these problems are tackled should be transparent to machine learning engineers.

There are a wealth of libraries, frameworks, and design heuristics aimed at deep learning, however these systems are often difficult to integrate and experiment with. Optimising the run-time of network operations using low level code such as OpenCL, OpenMP, and CUDA is time-consuming. When new or adapted deep learning approaches emerge, such as depthwise separable convolutions [1], or alternative sequence model cells [2], communities must wait for them to be supported by their preferred accelerator backend (such as ARM Compute Library, or cuDNN) before they can be used in high performance workloads. This limits the the rate of innovation from the machine learning community. Additionally, network level optimisations, such as pruning and quantisation often fail to give their expected performance improvements due to the caveats and lack of coordination with lower layers in the stack. The full neural network execution stack, and the importance of across-stack coordination is described in [3].

The design decisions of established frameworks often make them difficult to understand, and adapt. From the experiences of the previous part of this project (see section 1.2), I felt that a new modular inference-only framework designed from scratch, with

the intention of allowing outside models to be ported in, and alternative layer implementations to be easily deployed would be a useful tool for running experiments. The sentiment that migration difficulties of existing deep learning systems make them ill-suited to embedded systems has been shared with other researchers anecdotally, and is expressed in [4].

1.2 Previous work

This project expands on work done in the 4th year MInf project of the same name. I adapted "Darknet" [5], a deep learning framework written in C, so that it could run the MobileNet image classifier on an ODROID XU4 board. I then added choices of acceleration to the convolutional layers, using the libraries clBLAS, CLBlast, and my own naïvely implemented OpenCL kernels.

Progress on the project was significantly slower than I anticipated. Due to the codebase being undocumented, it took a long time to understand how the system worked. Once I understood the data flows it was difficult to integrate alternative layer types, and benchmark the difference that these made, due to both the code's opacity, and an overall design which did not easily support alternative implementations.

1.3 Objectives and Contributions

This work presents "Orpheus", an inference-only deep learning framework, for the research of novel apporaches in accelerating neural networks on heterogeneous devices.

Orpheus defines a standard workbench environment to test and develop deep learning acceleration techniques. It strives to minimize outside dependencies, which increases its usefulness across heterogeneous systems which may not support them.

Its lightweight, modular design enables fast experimentation, while a suite of unit tests ensure that developers' results remain valid.

Chapter 2

Background

This chapter presents a brief overview of deep learning, the deployment of these systems, and research efforts to improve their performance on embedded devices.

2.1 Convolutional neural networks

Neural networks are directed computation graphs, defined as a series of "layers" corresponding to a composition of a large number of operations applied input data.

Convolutional neural networks are characterised by the use of convolutional layers, which apply a series of small learnable kernels.

2.1.1 MobileNetV1 and Depthwise Separable Convolutions

The main network of interest in this project is MobileNetV1 [6], a convolutional neural network architecture, designed to be lightweight enough for mobile devices, while still achieving high accuracy. Its main way of achieving this is by using a technique called "depthwise separable convolutions" (first proposed in [1]) which replaces a conventional convolutional layer with two layers: a variant called a depthwise convolutional layer, followed by a pointwise convolution (which is a standard convolutional layer with a filter size of 1).

A depthwise convolutional layer is a variant of the convolutional layer, where each kernel is applied to single input channel. Typically this means that there will be as many kernels as input channels to the layer. However, one can have n kernels per input channel, where n is commonly called the *depth multiplier*.

2.1.2 Grouped Convolutions

Another variant of convolutional layers, notable for having fewer parameters, and being computationally cheaper is grouped convolutions. Originally introduced in [7] as a way to better parallelise neural network training across multiple GPUs, it was found to also improve the quality of filters learned.

The pertinent hyperparameter is the number of groups. A conventional convolutional layer has one group. For two groups, the kernels are split into two distinct subsets, with one group being applied to half on the input channels, and the other to the rest. When the number of groups is equal to the number of input channels, then we have a depthwise convolutional layer (with a depth multiplier of 1).

Due to this, grouped convolutions and variants have found utility in the neural architecture search and knowledge distillation communities, such as in [8], which looks at approaches of generating smaller *student* networks from large *teacher* networks, while endeavouring to maintain accuracy. Ensuring that these neural network techniques are well supported on lower levels of the neural network inference stack [3] is key to exploiting the potential of emerging deep learning applications in resource constrained environments.

2.2 Embedded devices

The term 'embedded device' is broad, applied to computing systems which form part of a larger ensemble of processes. Thanks to falling costs, and increased speeds, old products and equipment are seeing embedded computers placed at their core, such as cars, washing machines, farm equipment, and camera equipment. New solutions have also emerged, such as digital assistants, cutting edge medical equipment, and other so called "Internet of Things" (IoT) devices. This trend of low power, low cost computers in a huge range of objects has been called by some *ubiquitous computing*, or *pervasive computing*.

The deployment of neural networks on these systems has risen with popularity of deep learning applications in recent years. However, due to the high computational cost, often data is sent off-device to cloud computing infrastructure for processing, and results are returned. This approach suffers from a number of problems. First, there is latency in sending and receiving data over the internet, especially if the data is large, such as streaming video, or if the device has an unreliable connection. Secondly, there may be privacy risks derived from trusting a third party with potentially sensitive information, such as audio recordings from within domestic spaces.[1] Finally, in resource constrained environments such as embedded devices, the power cost of using on-board remote communication hardware is often untenable.

However, despite the reliance on cloud computing, today with low-cost hardware, and across-stack design to accommodate the various resource constraints, it is possible to run neural network workloads on these devices. Designing across the neural network inference stack involves approaches including: tuning models to suit the hardware environment they are involved in; optimising lower level software libraries to better support the needs of the evolving deep learning community; and ensuring that new generations of hardware have the capability to efficiently process these demanding workloads.

In this project, I work with an ODROID-XU4, a development board produced by Hard-

[1] Note that there are mitigations in development from the deep learning community.

kernel Co. Ltd, a South Korean designer/manufacturer. It features an ARM Mali-T628 GPU, ARM Cortex-A15 and Cortex-A7 big.LITTLE CPU, and and 2GB of shared LPDDR3 RAM. This board is good for investigating the embedded devices, as its specifications and hardware are shared by many modern models of smartphone.

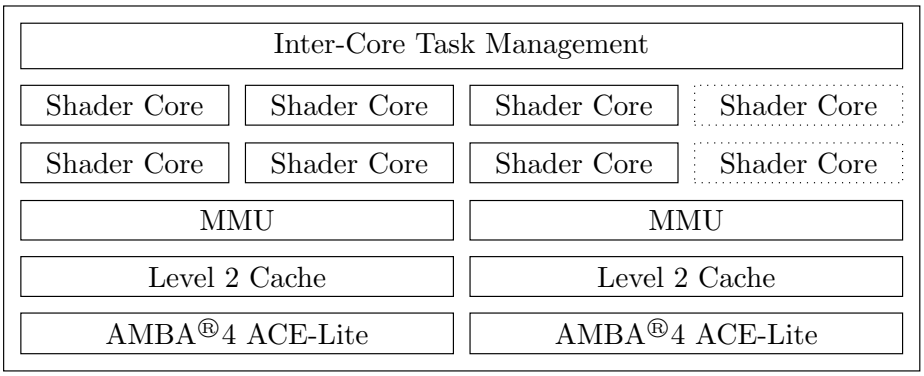
2.2.1 Mobile GPUs and the Mali-T628

Mobile GPUs consist of a small number of shader cores, where each core executes a large number of threads using SIMD units. Compared to desktop GPUs, which typically feature on the order of hundreds, to thousands, of shader cores, mobile GPUs are very low powered with around 1-32 cores. However, when leveraged properly, these are sufficient for accelerating many workloads considerably, including neural networks.

Our target device, the Mali-T628, uses a shared memory model, meaning that it shares its main memory pool with the rest of host device, rather than having its own. This excludes many desktop GPU focused algorithms which optimise data transfer patterns between global and local memory.

The Mali-T628 architecture, shown in Figure 2.1, has support for 1 to 8 shader cores. In the ODROID-XU4, 6 cores are present, split over two OpenCL devices.

Figure 2.1: Mali-T628 GPU architecture



The Mali-T628 offers API support for OpenCL (1.1), OpenGL, DirectX, and Google RenderScript. This enables it to be used for a variety of graphics and compute workloads.

2.3 OpenCL

OpenCL is a parallel programming API standard, supported by CPUs, GPUs, FPGAs, and a variety of other hardware accelerators. Programs which use OpenCL consist of two parts: host-code and device-code.

Device-code is executed on an OpenCL device (e.g. GPU), and describes the operation we want to parallelise (e.g. matrix multiplication) In OpenCL parlance, the functions

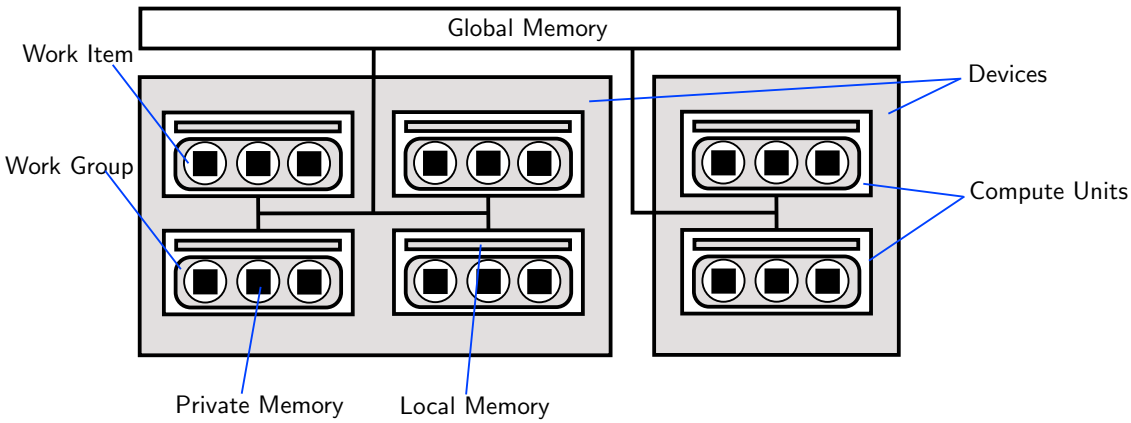
which are executed on devices are known as "kernels"[2].

Host-code is executed on the CPU, and is used to manage the OpenCL environment, preparing and managing data to be used by the device-code, and invoke device-code on target devices.

The specification for OpenCL version 1.0 was released by the Khronos Group in 2008. Subsequent versions have improved upon the original specification, adding device partitioning, shared virtual memory, nested parallelism, and generic address spaces.

OpenCL's model represents CPUs, GPUs, and other computing hardware as **devices**. Each device contains one or more **compute units**, which generally corresponds to a CPU or shader core. A **compute unit** is assigned a **work group**, which is a collection of **work items**, with shared **local memory**. A **work item** is a single invocation of a **kernel**. Each compute unit contains one or more SIMD lanes, which means that **work items** in a group may be executed sequentially or concurrently. Kernels are functions written in OpenCL C, based on C99, though abstraction layers such as SYCL [9] allow programmers to work at a higher level and take advantage of features such as compile-time optimisations.

Figure 2.2: Mali-T628, represented using the OpenCL model



2.4 ARM Compute Library

The ARM Computer Vision and Machine Learning library is a set of functions and utilities optimised for both ARM CPUs and GPUs to accelerate common machine learning workloads. It was first released to the public in March 2017 [10], and is under the MIT license.

Although targeted and optimised for ARM based devices running Linux and Android,

[2]Overloaded term, which in the context of OpenCL should not be confused with kernels in a convolutional neural networks.

it can be built for multiple architectures and operating systems, supporting OpenCL and NEON backends.

In this work, we will only be using its OpenCL backend, which is supported by a series of OpenCL kernels. At the time of writing, the library is still under active development, and is in version 19.02. It has seen some deployment in research [4] [11], and has around 1,300 stars on GitHub (where number of stars is taken as a proxy for project popularity [12] [13]).

2.5 Related Work

tvm [14] is a compiler stack for deep learning systems. Its design goal is to act as an intermediate layer from high level deep learning frameworks such as Tensorflow [15] and PyTorch [16], to the low level vendor-specific deep learning operation libraries. By translating various model formats to its own intermediate representation, it then leverages the Halide [17] and Loopy [18] systems to provide some graph level optimisations. The system *autoTVM* [19] builds upon tvm, by using a model based approach to improve tuning, complete with ad-hoc benchmarks to inform the model what real runtime performance of its candidate solutions will be. Though tvm is an interesting emerging platform, the ambition of its scope means the codebase is large, featuring several dependencies, and is thus difficult to iterate with.

NetAdapt [20] uses iterative pruning techniques, incorporating both inference time and empirically measured power consumption into its performance metrics, to reduce model size on embedded platforms. It uses Tensorflow Lite [21] as its backend. While focused on two edge devices, a mobile CPU and mobile GPU, it does not look into hardware specific considerations beyond the power consumption effects of pruning.

QS-DNN [22] is a reinforcement learning based auto-tuner for DNN primitive selection. Part of the BONSEYES collaborative project [23], it brings together a number of acceleration libraries under its framework, and uses a Q-Learning [24] model to search for optimal primitive choices across libraries, factoring in data format translation costs as discussed in [25]. In the paper, they run on a Nvidia Jetson TX-2, with inference time reduction being the focus. In [26] it is shown how *QS-DNN* can be implemented in an end-to-end workflow. The underlying framework used by QS-DNN seems to have similar design goals to Orpheus, and the model based approach to inference optimisation is one that I will explore and expand upon. However, as the framework is not yet publicly available, I can only speculate as to it similarity to Orpheus.

Chapter 3

Orpheus Framework Design

This chapter describes the design and development of the Orpheus framework, and why certain decisions were made.

In this project, I was keen ensure that Orpheus had maximum flexibility for whatever changes myself or future researchers wanted to investigate. This was ensured with an object-oriented design, with a system of inheritance and wrapper classes that had low interdependence, the use of templated programming, and a robust set of testing workflows.

3.1 System Design

3.1.1 Design Methodology

There was no formal design phase prior to the implementation of Orpheus, due to the exploratory nature of the work. Instead, an iterative approach was used. Early planning mapped out key requirements of the system, while implementation details were established and design modifications were made in response to new information, progress, or difficulties during development. The growth of my programming experience meant that some features were revised during development.

From the outset, I settled on using C++ as the main language of Orpheus. Its support for object orientated programming made it more attractive than C, and its ubiquity and speed compared to scripting languages such as Python gave it an advantage in the domain of application acceleration. The Khronos Group has officially supported C++ OpenCL bindings, which would enable better interfacing with GPU resources if needed.

The advantages of object orientated programming were clear, since neural networks are composed of layers, of a variety of types, with shared and unique properties. It is regarded as being a very performant language, due to the maturity of its toolchains, and is often said to have zero cost abstraction. With most neural network frameworks and libraries being either written in, or having broad support for C++, and C++ toolchains

being widespread across all but the most "bare-metal" embedded devices, it also made sense from an integration perspective.

Starting from an empty codebase, initial efforts were spent on getting a baseline system implemented from scratch, without accelerators, implementing all of the functionality needed to load and run inference on MobileNetV1. This served the dual purpose of providing a consistent benchmark, and giving me an opportunity to improve my technical experience of implementing neural network operations at a lower level. I hoped this would give me better insight into how I could accelerate these operations. Improvements would be made as new knowledge was gained, and utilities created. I was able to re-purpose some parts of the system from the 4th year project, however this was mostly advisory to new code.

Once the baseline system was working, I would begin to integrate the ARM Compute Library, and create a series of experiments. If time allowed, I would explore adding other accelerators.

The use of this iterative approach is not to suggest that formal design phases are not valuable however. While it made sense in this context, there were several core elements of Orpheus which had to be reworked far into development. These included how layers shared data between each other (since the output of one layer will be in input to another), and the way in which memory for is allocated from parameters and data. With more software development and domain experience, these issues could have been anticipated early on. However, overall the ad-hoc approach proved sufficient to produce a working system that met the design requirements, described below.

3.1.2 System Requirements

Below are the refined system requirements for the Orpheus framework. These were first sketched out at the beginning of the project, after a thorough evaluation of the shortcomings of the 4th year system. They were iterated upon during the initial months, and then referred to guide the remainder of development.

I **Model Portability** Production quality deep learning models should be easily used by Orpheus. This could be done by either creating a system which exports a production framework model to an Orpheus readable format, or writing a parser for model files.

II **Correctness** Orpheus should be able to run the operations of a neural network, returning the correct results every time. A testable architecture and a suite of tests would be able to fulfil this requirement. Models from other frameworks should exhibit the same behaviour in Orpheus as in their point of origin.

III **Modularity** It should be a natural part of the Orpheus workflow to replace components, and benchmark how the replacements fare against each other. Ideally, choosing between alternative components should be handled by a configuration step, rather than altering the codebase and recompiling.

IV **Maintainability** The overall design, and code clarity should be such that future researchers are able to adapt the system with relative ease. Ideally, it should be

easy to change individual aspects of the functionality without being familiar the implementation of the wider system, i.e. low component interdependency. Again, a suite of tests would be helpful, as they would enable researchers to be sure their changes work and do not break other components of the system. A suite of documentation and tutorials would also improve the user/contributor onboarding process.

V **Compatibility** Orpheus should be able to run on almost any platform. This should be achieved by minimising dependencies, such as outside libraries, or environment specific needs such as requiring a particular float length, address space size, or instruction set.

VI **Performance** Ideally Orpheus would have the same performance as production frameworks, or better. However, this overly ambitious. Instead, we expect that the implementation is not so inefficient as to render experiments meaningless. Components should be able to be compared fairly, without bottlenecks from elsewhere impacting their running. This work is focused on inference time, so if steps such as network loading is slow, this is acceptable.

3.2 Implementation

3.2.1 Class Hierarchy

When experimenting, I wanted to be able to quickly change the implementation of any given layer, so that comparisons between possible approaches can be made without requiring major codebase restructuring. This was a problem faced in the 4th year project.

For example, if I wanted to benchmark a new data format for a convolutional layer, Orpheus should allow me to create the implementation of the layer and test it in a network without having to search the entire codebase and replace all references to convolutional layers with a reference to the new one. It should support multiple versions of a given layer type running within a single network.

Hence, I used a “layer-first” class hierarchy, and a factory method pattern. Figure A.1 gives a sketch of this design. With the parent class of “Layer”, each distinct layer type is given its own abstract child class. This child class adds relevant methods and data structures.

Developers can derive their own layer implementations from the abstract child class, and can make any design decisions they want, such as using an outside library, or novel memory layout, as long as they follow the API of the child class.

For each layer type, distinct implementations are each given an identifier, which is used in the network configuration file. This is so that developers can specify what the implementation for each layer of their network will be. For example, specifying: “Use the ARM Compute Library in layers 1 through 4, the Orpheus baseline implementation in all others”.

They can also override the backend for the entire network. When Orpheus loads the network, it passes the layer choice to the factory method for that layer, constructs the layer, and adds it to the network.

By using C++'s templates feature, layers are defined using generic types. Thus, layers can be instantiated to have parameters of any type, which better enables the use of quantisation (since layers are written to support any weight data type), and reduces codebase size.

3.2.2 Layer Configuration

Satisfying Requirement I, the model format used by default in Orpheus is inspired by the one used in Darknet [5]. A model is defined by two files: one which describes the layers and their configurations, another which holds the parameters/weights.

Initially, I looked at using the Protocol buffers format (Protobuf) [27], due to its wide usage by a variety of deep learning frameworks, high level of compatibility across platforms, and maturity as a project. However, it proved too time consuming to extract data from a Keras [28] model protobuf, and thus I opted for a simpler solution.

An example of the layer configuration file can be seen in Appendix A.1. Being in plaintext, this makes it easier for humans to understand and debug. However, it also incurs a penalty when it comes to parsing time, as well as file size. Because of Orpheus' modularity, alternative model file formats can be added, however since this project is not trying to optimise load time, this is left as an area of future development.

The two files are generated by software, where a small Python script parses arbitrary Keras models, and saves them to disk. This automated approach has proved to be invaluable during testing, where small test networks can be made in Keras, exported, and validated against with ease.

3.2.3 Network Structure

In Orpheus, a network is an object with a sequential list of layers, as well as other small data and methods. There are five clearly defined stages to loading and running an Orpheus network, which improves the troubleshooting.

One must pass the weights and configuration files which describe the neural network of interest. Once initialised, a network can be reused to run inference on new input.

1. *Initialise* - load layer configurations from file, stop on error.
2. *Allocate* - allocate memory for layer parameters, inputs and outputs (if there is enough).
3. *Load Params* - copy parameters from weights file to each layer.
4. *Inference* - pass input data through all layers (note that one can also choose an arbitrary stopping point).

5. *Reset* - if any layers have state, for example a scratch buffer to store intermediate computations, then this stage resets this state between runs if required.

This simple to understand design misses some of the advantages of a full computation graph, such as graph level optimisations. However this is not currently in the scope of Orpheus, and the ARM Compute Library works at the granularity of the layer, thus the design is sufficient for performance and ease of use.

3.2.4 Unit Testing

Throughout the project, I used the common practice of test-driven development (TDD). This involved generating some expected outputs for a feature before it was implemented, and then writing the code that would generate those outputs. I used the Google Test Framework [29] to enable this, as it provides convenient testing primitives, and pretty-print of relevant test information.

Being the first large scale project I have undertaken which used TDD, my subjective experience was very positive, and it is a practice I will be carrying forward to future software endeavours.

One of the most useful examples of the unit testing approach was when late in the project I was compelled to change how the core data format of Orpheus worked. The old format required that data was passed between layers, which I came to understand later was a source of inefficiency. This is due to the high cost of data transfer. Thus, the new format did away with data passing, and moved to a data sharing model, with data dependencies being organised at the *Initialise* stage.

Once the design of the data format was established, I embarked on a major reworking of the entire codebase to reflect the redesign. However, this was achieved in a single day, since each change required could be tested in isolation.

This has bred confidence that even if future research calls for major restructuring of the Orpheus system, this will not be overly problematic. However, I believe that the core design as it now stands is sufficient for most needs.

3.2.5 Python Bindings

Though not originally part of the design, I found it necessary to build Python bindings for Orpheus, to aid in debugging. Later, they proved to be useful in other areas such as experimental workflows.

The motivation for the development of the bindings was the following:

An early version of Orpheus had all layers implemented with passing unit tests, however when running the exported MobileNet network, Orpheus returned predictions that were incorrect. Unit tests alone could not explain the error, and neither could my reasoning about the wider system. Thus, I had to find a way to access the data structures used when the network was running, and manipulate them to determine the sources of error. Since this error only appeared to occur when running a large network, visual

inspection of data outputted to a console would not be sufficient given the number of data points.

This exploratory work seemed best suited to a scripting language, rather than compiled language such as C++. Since the model exporting system, and Keras framework were accessed using Python, it made sense to connect Orpheus to Python.

I considered three options for this:

- Boost.Python [30]: A mature system, part of the Boost library project.
- PyBind11 [31]: A lightweight header only library, heavily influenced by Boost.
- SWIG [32]: Another mature library, which supports interfacing to multiple scripting languages beyond Python (such as Perl, Ruby, Javascript, and more thanks to the wide community).

Ultimately, PyBind11 was chosen, as it had fewer barriers to entry for both getting a minimum working example running, and integrating it into my build system (see Section 3.2.7). The bindings allowed direct access to Orpheus objects and methods, which negated the requirement for any abstractions.

With the bindings working, I was able to access data at various stages of computation from Orpheus, and manipulate them using tools in a Python environment (such as numerical computing library NumPy, and plotting library Matplotlib). Thus, I could iterate and experiment with the flexibility that scripting languages such as Python enable, using tools that I could trust were not the source of my error.

The first major bug I uncovered with this approach was weight transcription. To ensure correctness, I extracted to Python the layer parameters (such as weights and kernels) that Orpheus had loaded, and compared them to the Keras model weights. Since the former was extracted from the latter, I expected all weights to be identical. However, a small proportion of weights were showing small disparities, on the order of 10^{-8}. This had a "snowball effect", where the error would be negligible for any given layer, but would be magnified over successive layers. The problem ultimately lay in my model extraction script, which truncated weights under certain conditions.

A second major bug which was identified with the batch normalisation layer. Each of the layer types passed their respective unit tests, and when running the full MobileNet system, there were correct predictions on some test images. However, the full prediction vector did not exactly match that of the Keras model, and some test images would be misclassified relative to the Keras model. Thus the top-5 accuracy over a large test set was lower on Orpheus despite ostensibly being the same model.

To investigate, I took a test image, and passed it through the network, extracted the outputs of each layer, and compared that with the outputs for the same input in the Keras model, which should be identical. Using NumPy to compare the two tensors element-wise, errors were thrown if they were different by some threshold.

With this, I was able to identify that the batch normalisation layer was where errors started to occur. To dig deeper, I generated single layer networks in Keras, and ran them through Orpheus. However, the error did not reproduce. I then generated networks

Keras composed of a convolutional then batch normalisation layer. Figure 3.1 shows plots of the differences between the expected output and the actual output for some input. A working implementation would have no difference, and would plot a line $y = 0$.

To better identify the problem, six networks were created, each keeping all parameters constant. Each network then selected a single set of parameters (var, mean, gamma, beta) to be set as random noise. The purpose of this was to identify which of the four main operations might be causing the error, or if the memory regions of one of the parameter sets was being overwritten.

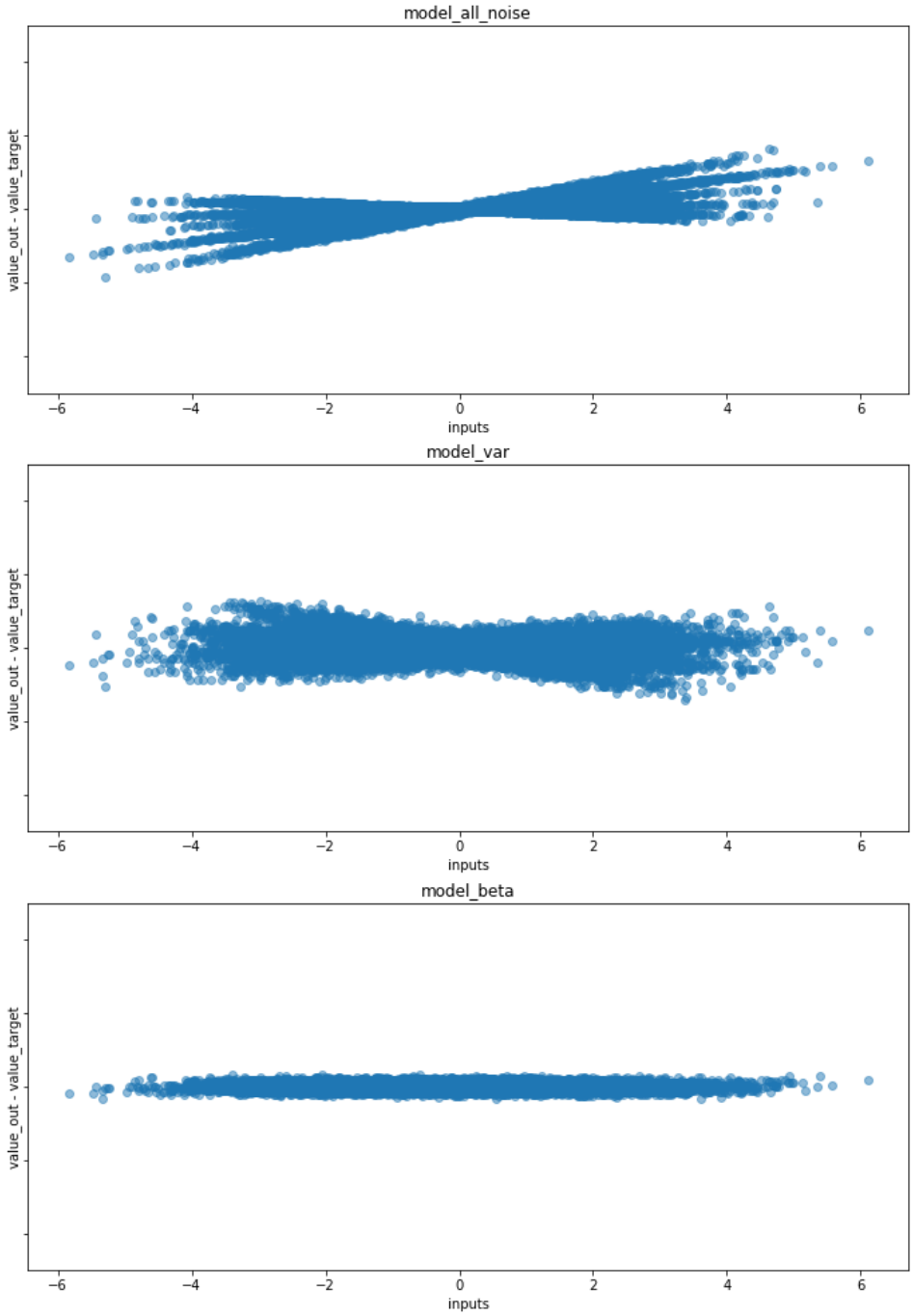
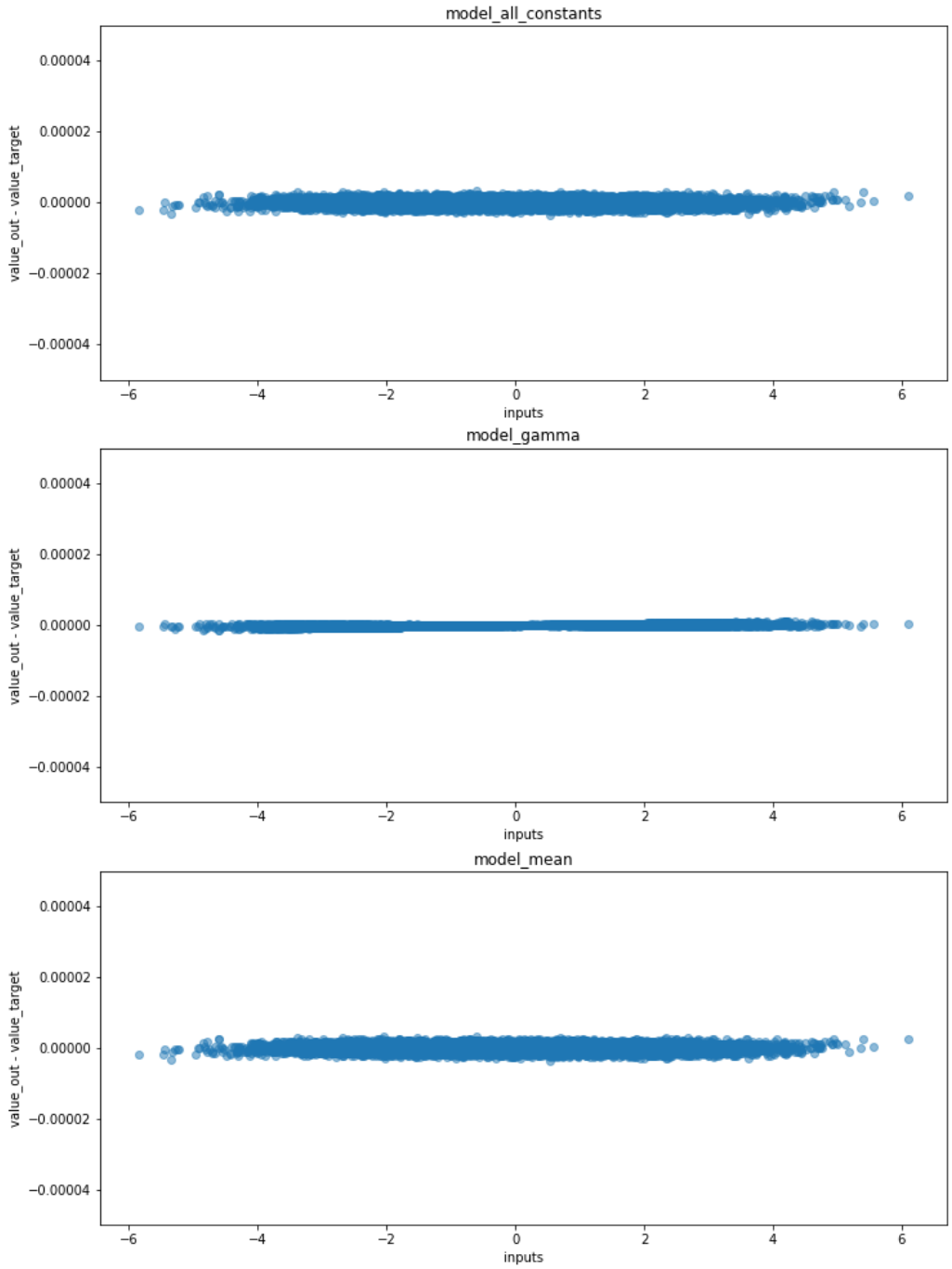


Figure 3.1: Six networks, ran in both Keras and Orpheus. Plots show the differences their outputs in Keras, and outputs on the same data in Orpheus. Expected difference is zero.



The Python bindings also enables Orpheus based systems to be used as part of larger workflows. Although network loading is slow, an Orpheus model object is persistent, and can be passed input again. NumPy and Matplotlib were used for manipulating data, and visualising errors. With further polish, users of Orpheus needed even be familiar with C++ to find it valuable in investigating their research question, if it can

be answered merely by choosing and tuning existing features.

3.2.6 Data Format

Different library backends generally require their own specific data type to store network weights. C++'s *std::vector*s, ARM Compute Library's *CLTensor*s, and OpenCL's buffers to name a few. Factoring in that many of these also have myriad of data layout choices one can make, for example NCHW versus NHWC, or the Morton order family [**rovder2019**]. Covering all data types and layouts becomes a difficult problem, and leads many frameworks to adopt a single choice.

However, for the purposes of research we want to be able to maintain flexibility in our choices, without having to rewrite large swathes of the codebase to support changes. We may wish to compose different backends together, that might have incompatible data formats.

Hence, I created the `orph::Tensor` class, a simple wrapper class for tensor data types. It has three types of data:

1. An enum parameter, which tells the program what the underlying type of the tensor data is.
2. Information about the shape of the tensor.
3. The data of the tensor. Represented as a variant type, which can be any of defined set of types, which are accessed using a type cast.

Layers in Orpheus have both their inputs and outputs as `orph::Tensor`. When initialised, the layer will check to see if the underlying type of the input is the same as the one it prefers to use. For example, ARMCL layers perform computation on data in the `CLTensor` format. If so, it can continue with its computation. Otherwise, the layer will prepare a conversion to the data to its preferred format. A layer will always return data in its preferred format.

Ideally, the conversion step should be composable, so if we have a conversion function from type $A \rightarrow B$, and from $B \rightarrow C$, we should be able to convert from $A \rightarrow C$, by running $A \rightarrow B \rightarrow C$. This feature is not yet implemented, but would be a way to reproduce some of the work done in [25], where networks can be faster by using alternative layer primitives (and associated layouts/formats) at different points of network inference.

A downside with the current implementation of `orph::Tensor` is that it uses a C++17 feature called `std::variant`. This goes against Requirement V, since not all platforms will have a C++17 compatible compiler. The functionality can be reworked using more classic language features, but due to time constraints this was not prioritised.

3.2.7 Build System

To manage the variety of build choices available, such as whether or not to include a given backend, to build unit tests, or Python bindings, as well as the associated finding and linking of libraries, I used the CMake build system.

This makes sense for a project of this scope, as personal experience has shown that Makefiles in large projects can become unwieldy to work with, and CMake supports a more modular design. I chose CMake over other build systems such as Scons or Bazel due to my previous experience with it, as well as relative maturity.

From within the build system, users can also set a "debug level", a numerical number representing the verbosity of log messages of Orpheus during its operations. By default, this is set to zero for maximum performance.

3.3 Unfinished Features

There are several features of Orpheus which are still in active development. None of them were essential for the completion of the project, however would have further demonstrated the extensibility of Orpheus. Development will continue on them through to the release of the system.

3.3.1 Additional Layers

Development was prioritised for layers which are used in MobileNetV1. As a result, some common layers are yet to be integrated. Below is a table of layers in Orpheus.

Layer	Baseline	ARMCL
Conv2D	✓	✓
Batch Normalisation	✓	✓
Depthwise Convolutional	✓	✓
Max Pooling	✓	✓
Average Pooling	✓	✓
Softmax	✓	✓
ReLU	✓	✓
Affine	✓	✗
Residual Blocks[1]	✗	✗
LSTM Cells[1]	✗	✗

Table 3.1: Orpheus Layer Table

Adding new layer types, as well as new layer implementations is a well defined workflow in Orpheus. Thus, in a future iteration more layers and thus more networks should be available to experiment with.

3.3.2 Quantisation

The use of generic programming in Orpheus improves its potential for support of using quantisation of weights. The preferred data format of weights can be specified when initialising the network. However, the quantisation functionality of ARM Compute Library is yet to be leveraged by Orpheus, and there is no performance difference using the baseline version with lower bit weights.

[1] Require slight alteration of interlayer data passing system.

3.3.3 CLBlast and OpenCL

The work carried out in Part 1 of this project used clBLAS, CLBlast, and hand crafted OpenCL kernels to accelerate inference. To compare performance, I tried to integrate these systems into Orpheus as options for layer backends. All approaches deal with managing the OpenCL context directly, thus required the same approach. Thus I designed and implemented a singleton class dubbed `CLController`, which manages access to the OpenCL devices, and context, and other details in a globally accessible way.

However, at time of writing, the process of instantiating and managing the command queue correctly, and enabling multiple forward passes of the network is still under development.

Chapter 4

System Evaluation

This chapter describes the experiments carried out using Orpheus. Our experiments find that Orpheus works as intended, and is capable as being used as a tool to pursue future research in the area of deep learning inference acceleration.

4.1 Framework Relative Performance

For Orpheus to be a valid research framework, it should have comparable performance to production frameworks, and not introduce overhead in the backends it uses.

This stipulation only need apply to parts of the Orpheus system that we benchmark. In this work, this means inference time across a whole network, and each layer. With its current network and weight loading modules Orpheus is embarrassingly slow compared to production frameworks. However, since this is not of research interest in this work we can discount it.

Framework	x86 Laptop (ms)	ODROID-XU4 (ms)
ARMCL	63	164
Orpheus+ARMCL	67	145
Orpheus+Baseline	5957	13638
Darknet (CPU)	2044	14494
Keras (CPU)	115	–

Table 4.1: Mean inference time of MobileNetV1

From table 4.1, we observe that ARM Compute Library gives the best performance on an x86 laptop CPU. The Eigen backend of Keras also performs well. Creating an Eigen integration for Orpheus will be a useful area of future research, since it is a mature and well optimised linear algebra backend which is likely to set a high standard to benchmark against. Note however that data on Keras inference performance on the test board could not be retrieved, as Keras has too many dependencies which makes installing and running on an embedded device such as the ODROID-XU4 non-trivial. This difficulty is a motivator for building Orpheus in the first place.

Most importantly, Orpheus does not incur an overhead when using the ARM Compute Library. This suggests that Orpheus itself does not introduce any significant costs during inference, making it valid tool for research. Although the performance is not identical, even averaged over multiple runs, this can be explained by the ARM Compute Library MobileNet model being different from the Orpheus model, and there being no guarantees that the ARM Compute Library API was called in precisely the same way.

As expected, the slower ODROID-XU4 board gives worse performance on than the x86 laptop. It is notable that despite being optimised for ARM Mali GPUs, the ARM Compute Library sees a 2x speedup on the x86 laptop CPU.

4.2 Per Layer Performance

Here, I present a comparison of per layer performance of MobileNetV1 between my baseline approach, and the ARM Compute Library. Figure 4.1 shows the relative performance.

The speedup is significant, with ARMCL being 94 times faster across the full network than the baseline system, and batch normalisation layers seeing the greatest improvement.

This finding is not a significant one, since the baseline system is poorly optimised. However, the workflows used to generate these results are reusable, and demonstrate the viability of Orpheus to explore answers to further research questions as more backends are added.

4.3 Answering Research Questions

Later into the development of Orpheus, some colleagues encountered a performance problem with a set of experiments in the PyTorch framework.

Attempting to build from research on model distillation in [8], they found that performance speedups were not as expected. Specifically, with networks containing grouped convolutions, varying the number of groups did not result in the linear speedup as expected. In fact, as can be seen in the graph of the original experiments, where grouped convolutions are shown in orange (Figure A.4)), the inverse occurred, with more groups increasing the inference time, despite there being fewer parameters and operations.

It was suspected that there was some inefficiency or bug in the PyTorch backend. However, due to the complexity of the PyTorch stack, debugging this problem proved to be difficult. This is where Orpheus could come in, to see if the problem can be reproduced in a different environment. After discussing the problem with the researchers, I was able to reproduce the problem in Orpheus with relative ease. Figure A.5 shows the average inference time of single layer networks, where only group size is varied. The same unexpected slowdown occurs, suggesting that the problem is more widespread than just PyTorch.

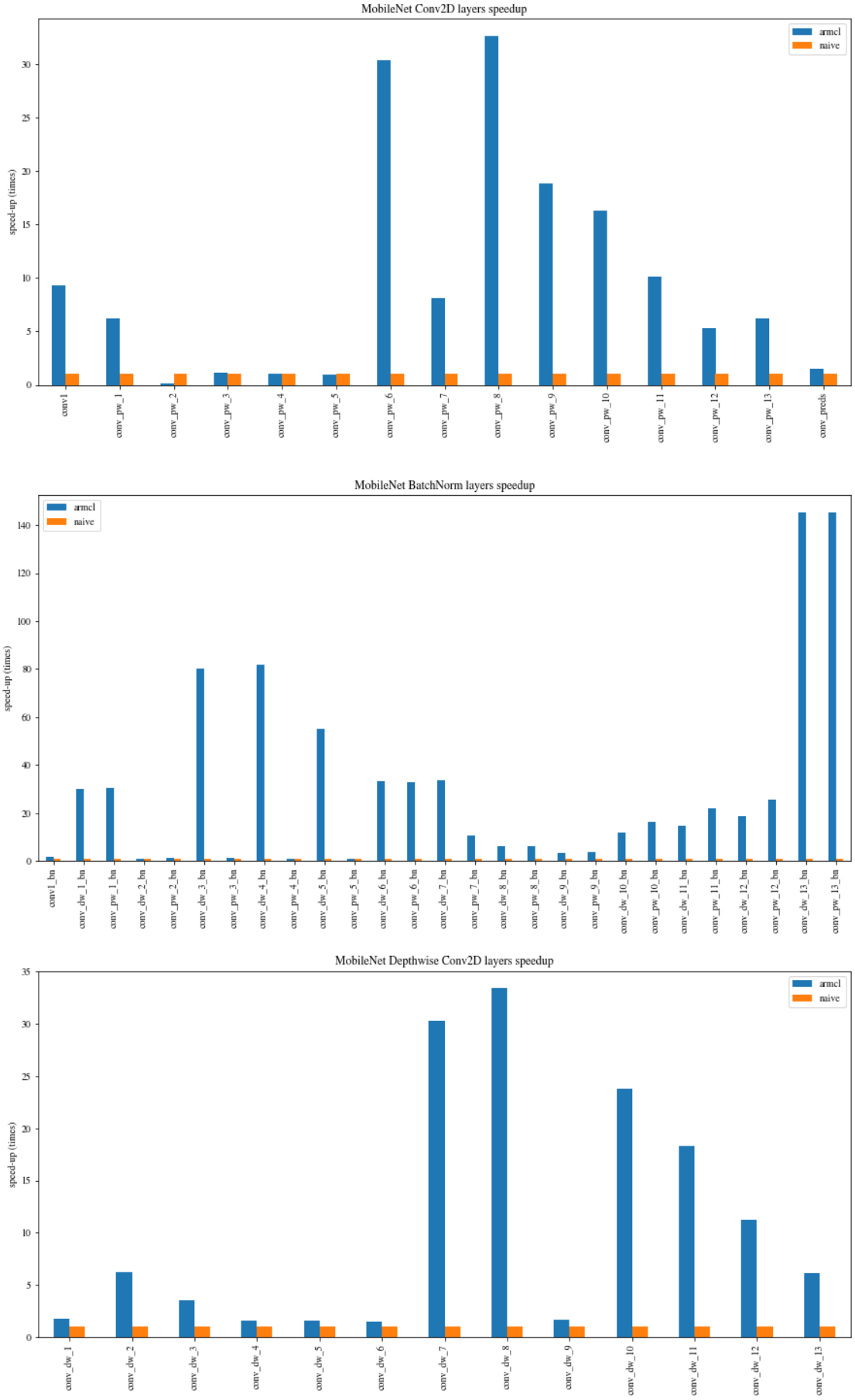


Figure 4.1: MobileNetV1 Per-Layer Performance in Orpheus

The research is ongoing, however the flexibility and transparency of Orpheus' runtime should make identifying and correcting the root cause of the problem an either to tackle problem.

Thus far, I have determined that in the ARM Compute Library, the implementation of grouped convolutions has not been fully developed. The latest version (at time of writing) only permits group size of greater than 1 when the data layout is NCHW. Fortunately this is the one used in the Orpheus bindings. Additionally, with a group size greater than 1, the only convolution primitive supported is the "GEMM" based one, with potentially more performant direct convolution and Winograd yet to be implemented.

Why performance drops in this way is still under investigation. The next step will be adding grouping to the naive Orpheus backend, to see if we can get our expected performance scaling. With that demonstrated, a more performant alternative will be developed.

4.4 Usability

I believe that the design of Orpheus makes it well suited for research, due to the properties discussed Chapter 3. However, as the developer of the framework from the ground up, I am significantly more familiar with the system's workings and quirks. This will mean that diagnosing problems and adding extensions will likely not be as straightforward to new users.

The usability of the framework in its current state has not been formally studied. Before it is ready for use by other researchers, it will require an improvement of documentation, and tutorial-article style demonstrations of common workflows such as adding a new layer implementation or supporting a new tensor data type. Once these steps have been completed, then Orpheus will be suitable for open source release, where I can elicit feedback and improvements from the community.

4.5 ImageNet Accuracy

The validity of the results can only be asserted if the outputs of operations carried out by each Orpheus backend are identical to those from production frameworks and the deep learning algorithms underpinning them. The unit tests go a long way to satisfying this requirement, by programmatic asserting that each component is correct in some known test cases. However, the most pertinent measure of validity is if the MobileNetV1 model in Orpheus achieves the same accuracy as the Keras.

Using 10,000 ImageNet images, and classifying them using MobileNetV1 in Keras, and MobileNetV1 in Orpheus (using both the Baseline and ARMCL backends). All three systems achieved the same accuracy across both the Top-1, and Top-5 metrics, demonstrating that there are no major bugs with the inference in Orpheus. The Python bindings of Orpheus made this experiment convenient to run, as ImageNet data could be loaded, passed, and inferred in a single Jupyter notebook.

Framework	Top-1	Top-5
Keras	68%	91%
Orpheus+Baseline	68%	91%
Orpheus+ARMCL	68%	91%

Table 4.2: Top-K Accuracy of MobileNetV1 on ImageNet

We can also verify that at each stage of computation in the network, the data is identical between Keras and Orpheus systems. This is done by accessing Orpheus layers via the Python bindings and comparing element-wise.

4.6 System Requirements

A way of assessing the success of the project is to compare against the requirements outlined in Chapter 3.

I **Model Portability** The model parsing script for Keras is successful in allowing relatively seamless exporting of models for use by Orpheus. It has a number of weaknesses. For example, very large models, on the order of dozens of millions of parameters (such as VGG 16), take a long time to export, typically over 20 minutes. Smaller models do not suffer from this problem, with MobileNetV1 being exported to file in under 30 seconds on average on my low-end laptop. The model export script in its current form only works with Keras models, however this was sufficient for a first version. The script can be adapted to other frameworks if needed. Alternatively, Orpheus can be adapted to accept a new format, such as the Protocol Buffers format explored earlier in the project.

II **Correctness** The unit tests, coupled with the suite of full network tests demonstrate that Orpheus produces valid, verifiable results.

III **Modularity** This has been achieved in two ways. First, the build system gives fine-grained control of which libraries Orpheus should be built with. It is trivial to add new ones without requiring changes elsewhere. Secondly, the layer hierarchy and factory pattern defined by Orpheus allows a flexible approach to layer backend experimentation.

IV **Maintainability** Orpheus is still lacking a useful level of documentation and tutorials. However, it has working examples of each work. For outside libraries, it is the responsibility of developers to ensure that they are installed correctly on their system. Orpheus' build system will find them if present. Library management is outside the scope of Orpheus, however the README provides a list of libraries Orpheus can be built with, and how to install them on a Linux system.

V **Compatibility** Each outside library which Orpheus uses is an optional build option, and thus a barebones benchmark should work on any system which has a C++ compiler. However, the `orph::Tensor` feature has a dependency on a feature from the C++17 standard. Older compilers will not support this, and an alternative implementation should be written if Orpheus is deployed in an environment which

is unable to support it. However, since this is a relatively small feature change, Orpheus is ultimately successful at maintaining compatibility.

VI **Performance** When using the ARM Compute Library, the performance of Orpheus+ARMCL on MobileNetV1 is comparable to that of ARM Compute Library alone.

Chapter 5

Conclusions and Future Work

This project resulted in the creation of a tool, Orpheus, to investigate the inference time of neural networks, with a focus on the ODROID-XU4 development board.

The tool provides a way for researchers to export trained neural networks to a transparent inference runtime, where components can be altered to answer various research questions.

To aid debugging, Python bindings were created. Additionally, these bindings improve the easy of use for embedding in other experimental workflows.

As a research tool, the directions which Orpheus can be taken in are numerous. However, first we will be focused on writing supporting materials to onboard future researchers. This will include "Getting Started" guides, worked examples of how to add a new layer type and integrate with testing, and expansion of the in-source Doxygen documentation.

Future work will include continuing the work with grouped convolutions discussed in Section 4.3, adding new layer types, benchmarking other state-of-the-art models, completing the integration of OpenCL and CLBlast backends, and fully enabling quantisation.

I will validate the ease of use of both Orpheus and the drafted materials by contacting researchers and students working in similar areas, asking them to use Orpheus for a period of time, and elicit feedback.

I will extend Orpheus with further layer types, so that a wider range of networks can be tested. I am curious about improving the depth of Orpheus' profiling capabilities beyond time measurements. There is a rich depth of memory usage profiling tools, such as `perf` and `Valgrind`.

There is established work on measuring the power consumption of development boards under different workloads. Exposing this information will allow developers to express a weighted preference between power consumption and inference time. Approaches such as [33] use measured inference time on target hardware to guide knowledge distillation. Adding power measurements, and offering additional memory layout and

layer primitive choices opens new areas of research for embedded neural architecture search.

Once these extensions have been explored, I hope to submit a paper detailing Orpheus as a paper to a leading conference or workshop, and open source the codebase under the MIT License so that the research community can extract value from it.

Appendices

Appendix A

Appendix

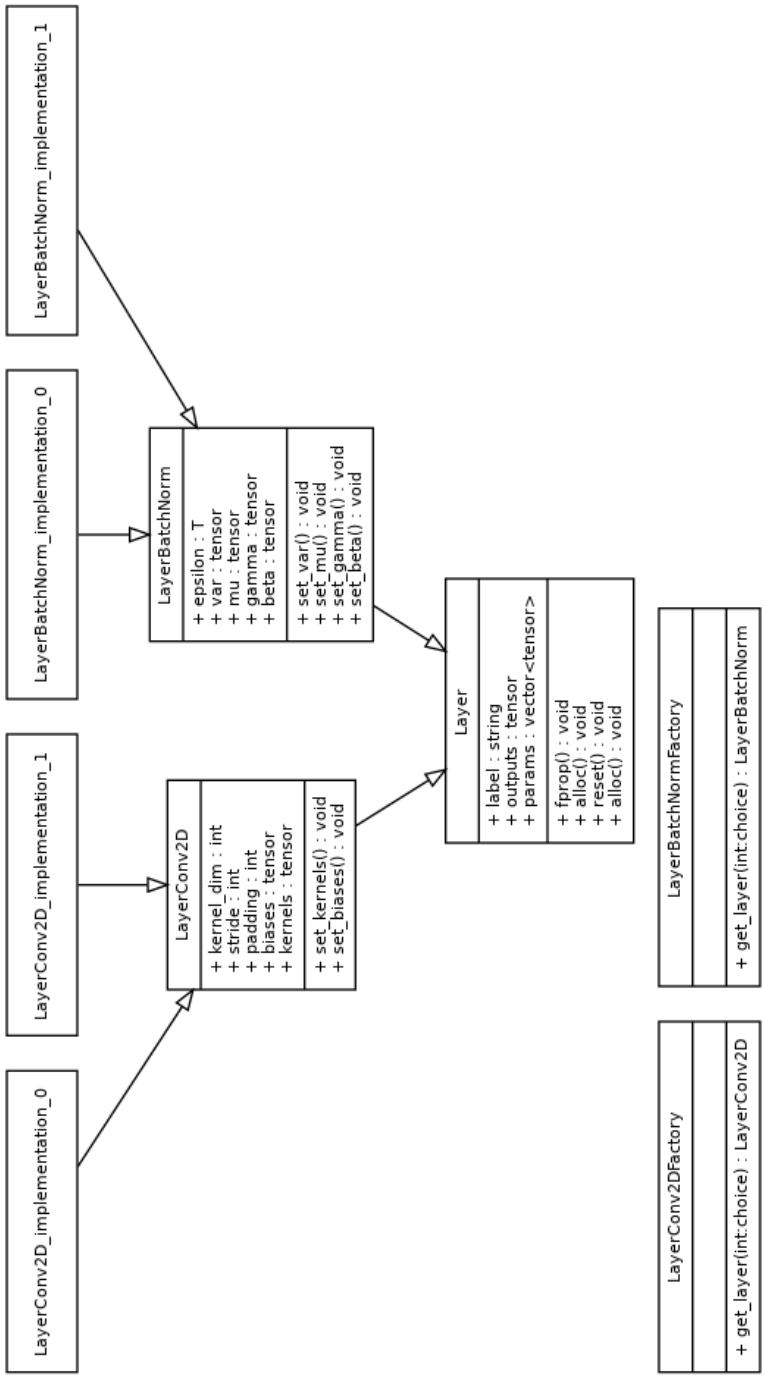
A.1 Example network configuration

```
[net]
height=27
width=27
channels=1

#conv1
[convolutional]
label=conv1
filters=3
size=3
stride=2
pad_hl=0
pad_hr=1
pad_wl=0
pad_wr=1
activation=linear
use_bias=0

#conv1_bn
[batch_normalize]
label=conv1_bn
epsilon=0.001

[relu]
label=conv1_relu
```

A.2 Figures

A.2.1 Sketch of Layer Design

Figure A.1: Sketch of Layer class hierarchy



A.2.2 Moonshine Performance

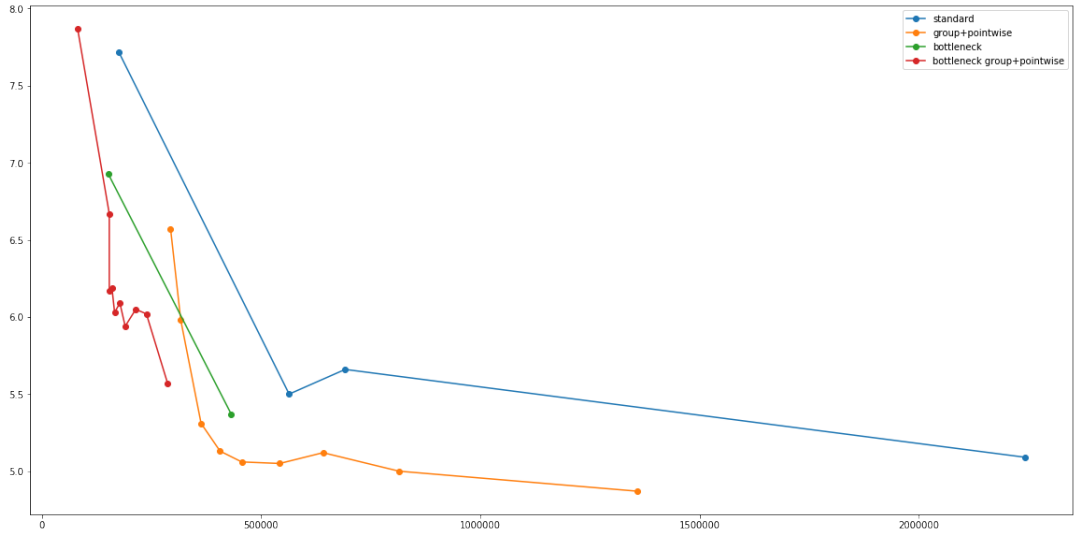


Figure A.2: Moonshine compression factor against number of parameters.

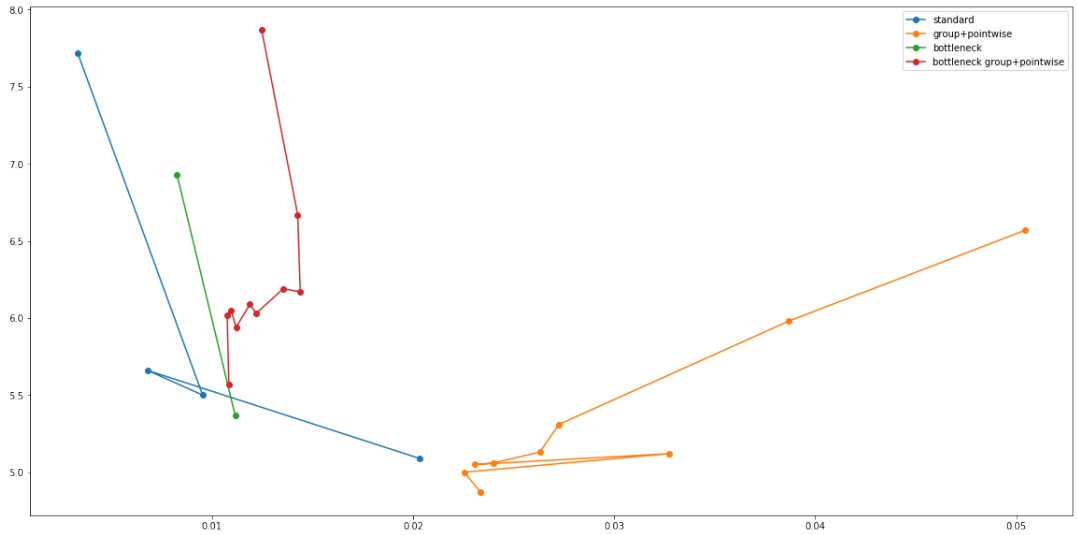
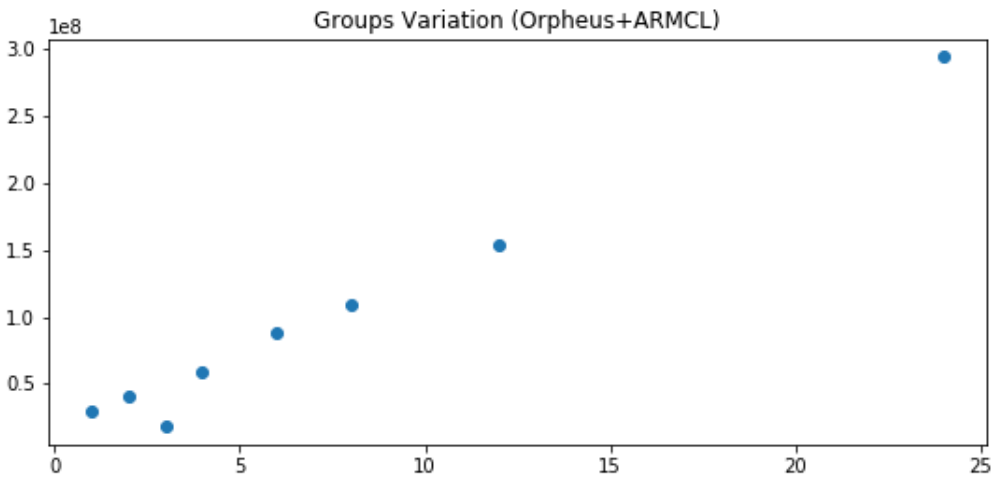


Figure A.3: Moonshine compression factor against inference time.

Figure A.4: Results from Moonshine research group. A higher Moonshine compression factor means more groups.

Figure A.5: Effect of increasing number of groups on inference time in Orpheus+ARMCL



References

[1] Laurent Sifre. "Rigid-Motion Scattering for Image Classification". PhD thesis. Ecole Polytechnique, CMAP, 2014.

[2] Oleksii Kuchaiev and Boris Ginsburg. "Factorization Tricks for LSTM Networks". In: *arXiv:1703.10722 [cs, stat]* (Mar. 2017). arXiv: 1703.10722 [cs, stat].

[3] J. Turner et al. "Characterising Across-Stack Optimisations for Deep Convolutional Neural Networks". In: *2018 IEEE International Symposium on Workload Characterization (IISWC)*. Sept. 2018, pp. 101–110. DOI: 10.1109/IISWC.2018.8573503.

[4] Dawei Sun, Shaoshan Liu, and Jean-Luc Gaudiot. "Enabling Embedded Inference Engine with ARM Compute Library: A Case Study". In: *arXiv:1704.03751 [cs]* (Apr. 2017). arXiv: 1704.03751 [cs].

[5] Joseph Redmon. *Darknet: Open Source Neural Networks in C*. 2013.

[6] Andrew G. Howard et al. "MobileNets: Efficient Convolutional Neural Networks for Mobile Vision Applications". In: *arXiv:1704.04861 [cs]* (Apr. 2017). arXiv: 1704.04861 [cs].

[7] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". en. In: *Communications of the ACM* 60.6 (2012), pp. 84–90. ISSN: 00010782. DOI: 10.1145/3065386.

[8] Elliot J. Crowley, Gavin Gray, and Amos J Storkey. "Moonshine: Distilling with Cheap Convolutions". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 2888–2898.

[9] *SYCL - C++ Single-Source Heterogeneous Programming for OpenCL*. Jan. 2014.

[10] Roberto Mijat. *Compute Library for Computer Vision and Machine Learning Available - Graphics & Multimedia Blog - Graphics & Multimedia - Arm Community*. en. https://community.arm.com/graphics/b/blog/posts/arm-compute-library-for-computer-vision-and-machine-learning-now-publicly-available.

[11] Lanmin Zheng and Tianqi Chen. "Optimizing Deep Learning Workloads on ARM GPU with TVM". In: *Proceedings of the 1st on Reproducible Quality-Efficient Systems Tournament on Co-Designing Pareto-Efficient Deep Learning*. ACM, June 2018, p. 3. ISBN: 978-1-4503-5923-8. DOI: 10.1145/3229762.3229764.

[12] M. Papamichail, T. Diamantopoulos, and A. Symeonidis. "User-Perceived Source Code Quality Estimation Based on Static Analysis Metrics". In: *2016 IEEE International Conference on Software Quality, Reliability and Security (QRS)*. Aug. 2016, pp. 100–107. DOI: 10.1109/QRS.2016.22.

[13] Amirali Sanatinia and Guevara Noubir. "On GitHub's Programming Languages". In: *arXiv:1603.00431 [cs]* (Mar. 2016). arXiv: 1603.00431 [cs].

[14] Tianqi Chen et al. "TVM: An Automated End-to-End Optimizing Compiler for Deep Learning". In: *arXiv:1802.04799 [cs]* (Feb. 2018). arXiv: 1802.04799 [cs].

[15] Martín Abadi et al. "TensorFlow: Large-Scale Machine Learning on Heterogeneous Distributed Systems". In: *arXiv:1603.04467 [cs]* (Mar. 2016). arXiv: 1603.04467 [cs].

[16] Adam Paszke et al. "Automatic Differentiation in PyTorch". In: (Oct. 2017).

[17] *Halide*. http://halide-lang.org/.

[18] *Loo.Py*. https://mathema.tician.de/software/loopy/.

[19] Tianqi Chen et al. "Learning to Optimize Tensor Programs". In: *Advances in Neural Information Processing Systems 31*. Ed. by S. Bengio et al. Curran Associates, Inc., 2018, pp. 3393–3404.

[20] Tien-Ju Yang et al. "NetAdapt: Platform-Aware Neural Network Adaptation for Mobile Applications". en. In: *Computer Vision – ECCV 2018*. Ed. by Vittorio Ferrari et al. Lecture Notes in Computer Science. Springer International Publishing, 2018, pp. 289–304. ISBN: 978-3-030-01249-6.

[21] *TensorFlow Lite*. en. https://www.tensorflow.org/lite/.

[22] Miguel de Prado, Nuria Pazos, and Luca Benini. "Learning to Infer: RL-Based Search for DNN Primitive Selection on Heterogeneous Embedded Systems". In: *arXiv:1811.07315 [cs]* (Nov. 2018). arXiv: 1811.07315 [cs].

[23] Tim Llewellynn et al. "BONSEYES: Platform for Open Development of Systems of Artificial Intelligence: Invited Paper". In: *Proceedings of the Computing Frontiers Conference*. CF'17. New York, NY, USA: ACM, 2017, pp. 299–304. ISBN: 978-1-4503-4487-6. DOI: 10.1145/3075564.3076259.

[24] Christopher John Cornish Hellaby Watkins. "Learning from Delayed Rewards". PhD thesis. Cambridge, UK: King's College, May 1989.

[25] Andrew Anderson and David Gregg. "Optimal DNN Primitive Selection with Partitioned Boolean Quadratic Programming". In: *Proceedings of the 2018 International Symposium on Code Generation and Optimization*. CGO 2018. New York, NY, USA: ACM, 2018, pp. 340–351. ISBN: 978-1-4503-5617-6. DOI: 10.1145/3168805.

[26] Miguel de Prado et al. "AI Pipeline - Bringing AI to You. End-to-End Integration of Data, Algorithms and Deployment Tools". In: *arXiv:1901.05049 [cs, eess, stat]* (Jan. 2019). arXiv: 1901.05049 [cs, eess, stat].

[27] *Protocol Buffers: Google's Data Interchange Format - Protocolbuffers/Protobuf*. Protocol Buffers. Apr. 2019.

[28] François Chollet. *Keras*. 2015.

[29] *Googletest: Google Testing and Mocking Framework*. Google. Apr. 2019.

[30] David Abrahams and Ralf W Grosse-Kunstleve. "Building Hybrid Systems with Boost.Python". en. In: *2003/03/20* (), p. 13.

[31] Wenzel Jakob. *Seamless Operability between C++11 and Python*. pybind. Mar. 2019.

[32] *SWIG Is a Software Development Tool That Connects Programs Written in C and C++ with a Variety of High-Level Programming Languages.: Swig/Swig*. SWIG. Mar. 2019.

[33] Jack Turner et al. "Distilling with Performance Enhanced Students". In: *arXiv:1810.10460 [cs, stat]* (Oct. 2018). arXiv: `1810.10460 [cs, stat]`.