NengoFPGA: an FPGA Backend for the Nengo Neural Simulator

by

Benjamin Morcos

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2019

© Benjamin Morcos 2019

Author's Declaration

This thesis consists of material all of which I authored or co-authored: see Statement of Contributions included in the thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Statement of Contributions

This work is an extension of my own work published in Morcos et al. (2018).

The initial interface between Nengo and the PYNQ board (Section 3.2.1) though functional, was re-implemented with help from Trevor Bekolay and Daniel Rasmussen to better integrate into the Nengo ecosystem.

The extended interface between the PC superhost and the ARM host on the PYNQ board (Section 3.2.2) was largely written by Xuan Choo.

Abstract

Low-power, high-speed neural networks are critical for providing deployable embedded AI applications at the edge. We describe a Xilinx FPGA implementation of Neural Engineering Framework (NEF) networks with online learning that outperforms mobile Nvidia GPU implementations by an order of magnitude or more. Specifically, we provide an embedded Python-capable PYNQ FPGA implementation supported with a Xilinx Vivado High-Level Synthesis (HLS) workflow that allows sub-millisecond implementation of adaptive neural networks with low-latency, direct I/O access to the physical world. The outcome of this work is NengoFPGA[1], a seamless and user-friendly extension to the neural compiler Python package Nengo. To reduce memory requirements and improve performance we tune the precision of the different intermediate variables in the code to achieve competitive absolute accuracy against slower and larger floating-point reference designs. The online learning component of the neural network exploits immediate feedback to adjust the network weights to best support a given arithmetic precision. As the space of possible design configurations of such quantized networks is vast and is subject to a target accuracy constraint, we use the Hyperopt hyper-parameter tuning tool instead of manual search to find Pareto optimal designs. Specifically, we are able to generate the optimized designs in under 500 short iterations of Vivado HLS C synthesis before running the complete Vivado place-and-route phase on that subset, a much longer process not conducive to rapid exploration. For neural network populations of 64–4096 neurons and 1–8 representational dimensions our optimized FPGA implementation generated by Hyperopt has a speedup of 10–484× over a competing cuBLAS implementation on the Jetson TX1 GPU while using 2.4–9.5× less power. Our speedups are a result of HLS-specific reformulation (15× improvement), precision adaptation (3× improvement), and low-latency direct I/O access (1000× improvement).

[1] https://www.nengo.ai/nengo-fpga/

Acknowledgements

I would first like to thank the team at Applied Brain Research whose vast knowledge and eagerness to teach and share is what started me on the path to further education. Once I began my thesis work, they continued to support me and assist me whenever possible while accommodating my studies as I continued to work part-time. Chris Eliasmith was always available for high-level discussion or guidance. Xuan Choo, Trevor Bekolay, Daniel Rasmussen, and Terry Stewart are all incredible resources for all software (or really any) questions and all directly contributed to the creation of NengoFPGA. A sincere thanks to the entire ABR team who have all led by example and inspired me to live a balanced life despite juggling work, school, and everything else.

Next I would like to thank everyone in the WatCAG lab: my friends and labmates Tushar Garg and Gurshaant Malik with whom I would discuss, commiserate, and laugh; and my supervisor Prof. Nachiket Kapre. Before beginning work with Nachiket I floundered about with my meagre FPGA experience and his mentorship, experience, and ambition was a welcome change of pace! I have no doubt that I could not have accomplished this work in this timeframe without his continued guidance and support. Through Nachiket I was also able to teach in China and present a conference paper in Japan, which were both incredible experiences, and for that too I am thankful.

Finally, as most do, I would like to thank my family and friends for their encouragement and support. Both family and friends were understanding when I was busy and stressed and reassuring when I finally made time for them. In particular, I would like to name Mr. Trevor Flynn who always inspires me to do my best and has an uncanny ability to remain in high spirits.

Dedication

This work is dedicated to the serendipity that bestows great opportunities upon those of us adrift in a sea of potential without a plan.

Table of Contents

List of Tables **xi**

List of Figures **xii**

List of Algorithms **xiv**

1 Introduction **1**

2 Background **4**

2.1 The Neural Engineering Framework 4

2.1.1 Principle 1: Representation 4

2.1.2 Principle 2: Transformation 6

2.1.3 Comparison to Deep Networks 7

2.2 Nengo 8

2.3 Neuromorphic Computing 10

2.4 High-Level Synthesis 12

2.5 PYNQ 13

3 NengoFPGA 15

3.1 FPGA Architecture . . . 15

3.1.1 Design Parametrization . . . 16

3.1.2 HLS Formulation . . . 18

3.1.3 Fixed-point Design . . . 28

3.1.4 Parameter Tuning . . . 29

3.1.5 Copy Protection . . . 33

3.2 Interface . . . 33

3.2.1 Direct ARM Host . . . 34

3.2.2 Remote PC Superhost . . . 35

3.2.3 Interface Considerations . . . 36

3.3 Using NengoFPGA . . . 38

3.3.1 Converting from Nengo to NengoFPGA . . . 38

3.3.2 Basic Simulation . . . 41

4 Results & Discussion 43

4.1 Impact of HLS Code Description . . . 43

4.2 Parameter Tuning with Hyperopt . . . 44

4.2.1 Initial Bounding . . . 45

4.2.2 Sensitivity of Precision to Design Parameters . . . 48

4.2.3 Hyperopt Convergence Rate . . . 49

4.2.4 Hyperopt Design Optimization 49
4.2.5 Pareto Optimal Designs 51
4.2.6 Floating-point vs. Fixed-Point 53
4.3 Comparison with Jetson TX1 GPU 54
4.3.1 Practical Application of NengoFPGA 55
4.4 System Profiling 58

5 Conclusion & Future Works **60**
5.1 Conclusion 60
5.2 Future Work 61
5.2.1 Larger FPGAs 61
5.2.2 More Functionality 62
5.2.3 Hardware Accelerators 62
5.2.4 Convolutions 63

References **64**

APPENDICES **75**

A Fixed-Point Precisions **76**
A.1 Hyperopt Search Space 76
A.2 Pareto Optimal Designs 77
A.3 Selected Fixed-Point Designs 79

B Power & Performance Data **80**

B.1 Jetson TX1 Power & Performance 80

B.2 FPGA Power & Performance 80

List of Tables

3.1 Resource–performance summary 27

A1 Search space bounds for Hyperopt runs 76

A2 Pareto optimal designs from err–resource minimization ($N = 200$, $D = 1$) . 77

A3 Pareto optimal designs from direct cycles minimization ($N = 200$, $D = 1$) . 78

A4 Pareto optimal designs from err–resource minimization ($N = 64$, $D = 1$) . 78

A5 Pareto optimal designs from err–resource minimization ($N = 4096$, $D = 1$) 78

A6 Selected fixed-point designs 79

B1 Jetson TX1 power & performance 81

B2 PYNQ-Z1 power & performance 81

List of Figures

2.1 Comparison of DNN and NEF topologies 8

2.2 Nengo ecosystem 9

2.3 Nengo GUI 10

3.1 Basic computation schematic 16

3.2 Simplified parallel computation schematic 19

3.3 Full parallel computation schematic 25

3.4 Hyperopt optimization flow 32

3.5 High-level NengoFPGA system diagram with ARM host 35

3.6 High-level NengoFPGA system diagram with PC superhost 36

3.7 High-level NengoFPGA system diagram with multiple FPGAs 37

4.1 Resource–performance improvement wiht HLS evolution 44

4.2 Accuracy with varied encoder precision 46

4.3 Accuracy with varied error precision 46

4.4 Error trend with varied error precision (fixed integer bits) 47

4.5 Error trend with varied error precision (fixed fractional bits) 48
4.6 Designs optimized for different values of N 49
4.7 Overall convergence of Hyperopt runs 50
4.8 Hyperopt trials showing Pareto optimal front 51
4.9 Evaluation of error–resource Pareto optimal designs 52
4.10 Performance of floating-point compared to fixed-point designs (1D) 53
4.11 Performance of floating-point compared to fixed-point designs (8D) 54
4.12 Performance comparison with Jetson TX1 55
4.13 Power comparison with Jetson TX1 56
4.14 FPGA performance for an adaptive controller 57
4.15 FPGA performance for a many body controller 57
4.16 FPGA performance for controlling inverted pendulum 58
4.17 High-level NengoFPGA system diagram with timing breakdown 59

List of Algorithms

3.1 High-level load–evaluate workflow 19

3.2 Basic NEF implementation 21

3.3 Restructured decode step 23

3.4 Explicit parallelism 24

3.5 High-level copy protection workflow 33

Chapter 1

Introduction

As the end of Moore's law (Moore, 1965) and Dennard scaling (Dennard et al., 1999) inevitably approaches, the semiconductor industry faces increasing technical and physical challenges in the manufacturing and fabrication of viable chips. Simultaneously, the machine learning revolution is creating a need for more powerful processing hardware that can train and evaluate sophisticated neural networks. FPGAs provide a configurable computing substrate that allow us to gracefully adapt to the end of Moore's law by configuring our hardware resources specifically for our computing tasks as needed. In particular, they are a great match for machine learning tasks as they provide parallel access to thousands of processing (DSP) and memory (RAM) blocks as well as direct connections to external I/O.

Machine learning (ML) developers commonly use Python as their development environment. Many popular packages such as Tensorflow (Abadi et al., 2015), Keras (Chollet et al., 2015), Nengo (Bekolay et al., 2014), and others are built around Python, and retain widespread use in the community. On the hardware side, ML developers have embraced GPUs for training and inference. This has been made possible by the availability of optimized GPU libraries, high-level CUDA (Nickolls et al., 2016) or OpenCL (Khronos OpenCL Working Group et al.) programming environments, and ease of integration with Python. FPGAs have high potential in terms of performance and efficiency; however, they have been seen as exotic devices that have a steep learning curve for software programming because they use low-level languages like VHDL or Verilog. To help address the productivity gap, FPGAs now allow programming using C/C++ with High-Level Synthesis (HLS) compilers. Furthermore, Xilinx has also introduced PYNQ, a Python environment for accessing

FPGA hardware on Zynq FPGA devices. The PYNQ environment has a clean Application Programming Interface (API) for configuration of the FPGA, data movement using Direct Memory Access (DMA), interfacing with GPIO, and more. An IEEE survey places Python and C++ as the top two programming languages (Cass, 2018) and a combination of HLS and PYNQ marries these two languages creating a more attractive starting point for software developers.

FPGAs offer significant advantages over GPUs in terms of latency, power use, and configurability. These features are particularly critical in power-limited, edge of the network deployments, for instance in IoT, mobile, or real-time applications. As a result, there have been many projects that aim to marry ML and embedded FPGAs. Most projects focus primarily on convolutional or deep networks (*e.g.* Hao and Quigley 2017; Nakahara et al. 2019; Noronha et al. 2018; Wang and van Schaik 2018; Aydonat et al. 2017). There are also several projects that try to create more general frameworks (*e.g.* Yinger et al. 2017; Abdelfattah et al. 2018; Umuroglu et al. 2017, 2018). There also exist projects that explore more detailed and biologically plausible implementations (*e.g.* Carlos Moctezuma et al. 2015) and some even target the Neural Engineering Framework (NEF) (*e.g.* Berzish et al. 2016; Corradi et al. 2014; Wang et al. 2014a) as this work does. The goal of this work is to merge these focuses into a single project: a flexible framework for implementing biologically plausible NEF-style networks on FPGA[1].

We present an optimized FPGA backend that integrates HLS-generated hardware wrapped in PYNQ APIs with the Nengo (Bekolay et al., 2014) neural network development framework. Nengo is a Python package for simulating spiking and non-spiking, large-scale neural networks with unique support for the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2003). Nengo includes a graphical interface to help visualize network topologies and inspect real-time represented values in the model. It is flexible and can implement traditional deep learning, vision, and motor control applications but goes beyond that to include working memory, hierarchical reinforcement learning, inductive reasoning, and planning. In fact, the world's largest functional brain model, Spaun (Eliasmith et al., 2012; Choo, 2018), was built using Nengo. This 6.5-million neuron model demonstrates the rich range of capabilities of the framework. Nengo currently supports CPU, GPU, specialized neuromorphic chips (Mundy et al., 2015; Voelker et al., 2017; GitHub:nengo-loihi, 2019), and other backends. With this work Nengo is now able to target PYNQ FPGA

[1]Currently we do not support convolutional or deep networks, but this is planned for future work (see Section 5.2)

boards using a new FPGA backend, NengoFPGA, integrated seamlessly into the traditional Nengo workflow (Morcos et al., 2019). We can run Nengo directly on the System on a Chip (SoC) device by leveraging the ARM processor adjacent to the Zynq FPGA but we go one step further and include infrastructure to run directly from a PC host. NengoFPGA will automatically handle connection and communication with the FPGA device on the same local network allowing users to explore and develop NengoFPGA in the same environment as they would use standard Nengo. This allows developers to enjoy a fully featured Nengo system with access to PC resources and subsystems while still allowing FPGA acceleration with only a $\approx 24\%$ overhead to NengoFPGA pipeline.

In summary:

- We develop a seamless FPGA backend for Nengo to realize low-power, low-latency embedded systems that use neural network structures with online learning. With direct I/O access we acheive sub-microsecond evaluation of small networks and with a system including a PC and an ARM CPU as hosts we still maintain real-time performance within the widely accepted 1 ms time envelop.
- We use an HLS description of the neural networks that is parametric and flexible enough to cover a range of implementation possibilities.
- We reformulate the parallelism in the software description in order to overcome the limitations of Vivado HLS and expose dataflow and pipeline parallelism. This reformulation leads to $15\times$ performance improvement.
- We reduce the precision of the arithmetic operations using Hyperopt (Bergstra et al., 2015) to automatically find optimal parameters for the design. Notably, the included online learning allows the network to continuously adjust the network weights to perform the desired function within the constraint of the chosen bit precision. This reduced precision implementation leads to an overall $50\times$ improvement over the original implementation.
- We demonstrate fully embedded performance by bypassing the PC and ARM hosts and directly integrate sensors and actuators with the adaptive neural network architecture over GPIO which erases data movement delays and improves performance by three orders of magnitude.

Chapter 2

Background

2.1 The Neural Engineering Framework

This section introduces the Neural Engineering Framework (NEF) (Eliasmith and Anderson, 2003). The NEF provides methods for implementing spiking or non-spiking, dynamic neural computations in arbitrary vector spaces. It allows for flexibility in low-level details such as the neuron model or method of adaptation and yet promotes higher-level abstract descriptions of algorithms and topologies at the same time. In addition to arbitrary feedforward networks, the NEF lends itself to biologically plausible cognitive architectures (Eliasmith, 2013) and the control and modelling of dynamical systems. This makes it uniquely well suited to online, real-time, and recurrent networks, which differs from the contemporary focus of backpropagation-trained networks and inference engines. The NEF is built on the principles of *representation*, *transformation*, and *dynamics*. For the purposes of this work, we focus on the first two principles, *representation* and *transformation*.

2.1.1 Principle 1: Representation

A key aspect of Principle 1 is akin to that of population coding wherein a population, or *ensemble* of neurons as we call it, collectively represent some real value based on the various distributions of activities given an input stimulus.

Given a D-dimensional time-varying signal, $\boldsymbol{x} \in \mathbb{R}^D$, defined in "state-space" as an input, we map it to an N-dimensional representation in "neuron-space". We call this mapping from state-space to neuron-space *encoding* and call the resultant neuronal representation the *activity*, $\boldsymbol{a} \in \mathbb{R}^N$. Each vector element in this neuronal activity representation corresponds to the activity, a_i, of a single neuron, each of which contributes to the collective representation of the real input value. This encoding step can be expressed as follows:

$$a_i = G[\alpha_i (\boldsymbol{e}_i \cdot \boldsymbol{x}) + b_i] \qquad \text{with } i \in \{1, \ldots, N\} \tag{2.1}$$

where

- a_i is the activity of the ith neuron in the ensemble;
- G is the non-linear transfer function of the neuron model, which in the case of this work is the Rectified Linear Unit (ReLU) simply defined as $G[v] = \max(0, v)$;
- $\alpha_i > 0$ is the gain term corresponding to the ith neuron in the ensemble;
- $\boldsymbol{e}_i \in \mathbb{R}^D$ is the ith row of the encoder matrix, $\boldsymbol{E} \in \mathbb{R}^{N \times D}$, that defines the distribution of activity given a stimulus for each neuron in the ensemble;
- $\boldsymbol{x} \in \mathbb{R}^D$ is the state-space input; and
- b_i is the bias term that accounts for background activity in the ith neuron in the ensemble.

The neuron activity, a_i, is then mapped back to state-space resulting in a D-dimensional output vector, $\boldsymbol{y} \in \mathbb{R}^D$. This mapping from neuron-space back to state-space is called *decoding* and can be expressed as follows:

$$y_j = \boldsymbol{d}_j \cdot \boldsymbol{a} \qquad \text{with } j \in \{1, \ldots, D\} \tag{2.2}$$

where

- y_j is the jth element of the state-space output, $\boldsymbol{y} \in \mathbb{R}^D$;
- $\boldsymbol{d}_j \in \mathbb{R}^N$ is the jth row of the decoder matrix, $\boldsymbol{D} \in \mathbb{R}^{D \times N}$, that linearly maps the given activity in neuron-space back to state-space; and

- $\boldsymbol{a} \in \mathbb{R}^N$ is the vector all neuron activities.

The encoders, $\boldsymbol{E}$, and bias, $\boldsymbol{b}$, are randomly generated and the decoders, $\boldsymbol{D}$, can be analytically obtained by least-squares optimization of the representational error, $||\boldsymbol{x} - \boldsymbol{y}||$, across the domain of $\boldsymbol{x}$ in state-space.

2.1.2 Principle 2: Transformation

The first principle of the NEF, representation, allows us to recreate inputs using a neuron-space representation but does not lend itself to any interesting computation in that basic form. Principle 2, on the other hand, describes methods with which to compute arbitrary functions. The encoding and decoding methodologies described in Equations 2.1 & 2.2 remain unchanged; however, the values of the decoders, $\boldsymbol{D}$, are modified and it is this updated decoder matrix that allows the computation. Instead of minimizing the representational error, $||\boldsymbol{x}-\boldsymbol{y}||$, more generally, we can find decoders that represent an arbitrary linear or non-linear function, $f(\boldsymbol{x})$, by minimizing $||f(\boldsymbol{x}) - \boldsymbol{y}||$. Furthermore, the output dimensionality of this arbitrary function need not match the dimensionality of the input. More concretely, given $\boldsymbol{x} \in \mathbb{R}^{D_{in}}$ and $\boldsymbol{y} \in \mathbb{R}^{D_{out}}$, D_{in} need not equal D_{out}.

The decoders may also be initialized to non-optimal, arbitrary values that are updated in real-time using online learning according to a particular learning rule. In the case of this work we employ the Prescribed Error Sensitivity (PES) (Bekolay et al., 2013) learning rule which updates the decoders each step by:

$$\Delta \boldsymbol{D} = -k\, \boldsymbol{Err_x} \cdot \boldsymbol{a} \tag{2.3}$$

where

- $\Delta \boldsymbol{D} \in \mathbb{R}^{D \times N}$ is the prescribed change to the decoders for this step through the network;
- k is the scalar learning rate that controls how quickly the system will adapt;
- $\boldsymbol{Err_x} \in \mathbb{R}^D$ is the error signal for the given system which can be calculated (*i.e.* $||f(\boldsymbol{x}) - \boldsymbol{y}||$), but can also be provided directly by an external source; and

- $\boldsymbol{a} \in \mathbb{R}^N$ is the vector all neuron activities.

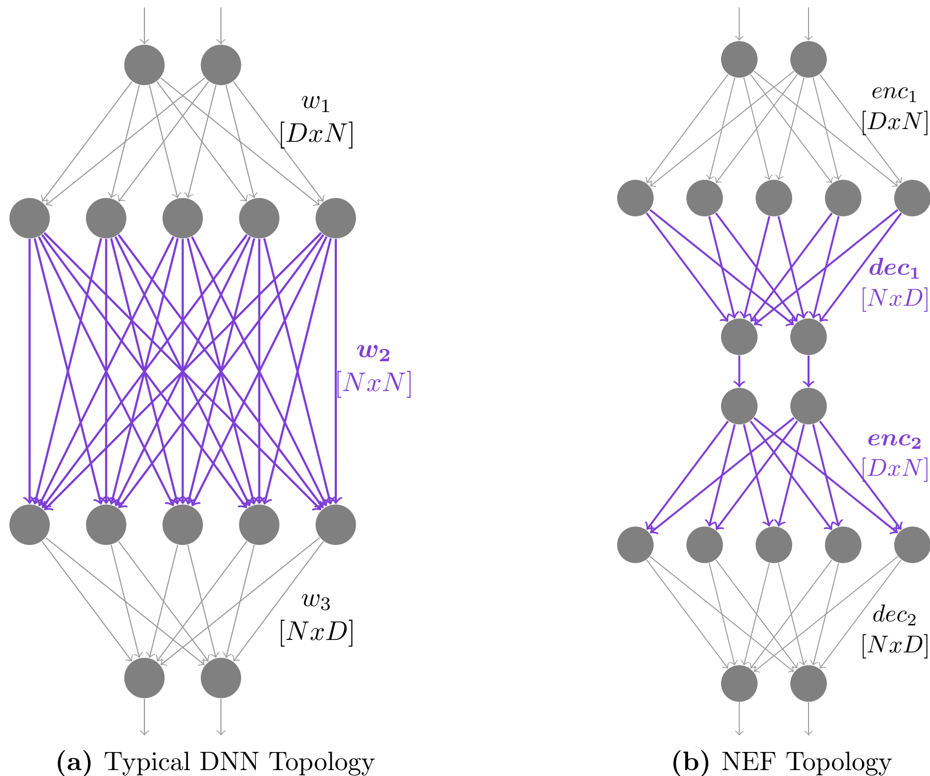
The neural network developer must choose a reasonable value for the learning rate, k, that 1) does not overshoot too far and oscillate, while 2) converging quickly enough to adapt to system changes in real-time. The neural activity, $\boldsymbol{a}$, is included in the decoder update in Equation 2.3 to allow decoders to be selectively updated. Only those neurons with a non-zero activity are contributing to the representation of the given input and consequently only those neurons contributing to the current representation need updating while neurons not involved are left unaffected.

2.1.3 Comparison to Deep Networks

In order to help understand NEF networks we will compare them to the more ubiquitous Deep Neural Networks (DNNs). Consider Figure 2.1 that compares typical DNN topologies to NEF topologies. Figure 2.1a shows a D-dimensional state-space input being projected to N neurons in neuron-space through weights $\boldsymbol{w_1}$ then fully connected to a second layer of N neurons by $\boldsymbol{w_2}$ and finally projected back to a D-dimensional state-space output by $\boldsymbol{w_3}$. The NEF topology in Figure 2.1b similarly takes a D-dimensional input and produces a D-dimensional output, both in state-space. However, instead of using a large weight matrix ($\boldsymbol{w_2} \in \mathbb{R}^{N \times N}$) to connect the two layers of neurons, the NEF splits this into two smaller matrices and moves back to a D-dimensional state-space representation between the layers. Typically we have $N \gg D$ so this method has two notable benefits:

1. The NEF saves memory since storing $\boldsymbol{enc}, \boldsymbol{dec}^T \in \mathbb{R}^{D \times N}$ is fewer elements than $\boldsymbol{w} \in \mathbb{R}^{N \times N}$.
2. The NEF saves on bandwidth since we are now only transmitting a D-dimensional vector between layers instead of a N-dimensional vector.

The NEF also allows the encoders matrix to be randomly generated in contrast to conventional networks in which weight initialization can have significant repercussions (Thimm and Fiesler, 1997). Of course, with random encoders the onus is placed on the decoder matrix to produce coherent values. The decoders can be solved by conventional methods, such as backpropagation, but can also be analytically solved or trained online during inference as mentioned in Section 2.1.2.



(a) Typical DNN Topology **(b)** NEF Topology

Figure 2.1: A visual comparison between typical DNN and NEF topologies. In this example, both topologies accept some input at the top of dimensionality D and pass through two layers of N neurons each. Then a D-dimensional output is produced at the bottom.

2.2 Nengo

Nengo (Bekolay et al., 2014) is a source available Python framework that enables high-level description, debugging, visualization, analysis, and deployment of neural networks. It is actively used and developed with 24 contributors, 423 stars, and 125 forks on github[1] at the time of writing[2]. While alternative frameworks exist for simulating large-scale neural models or cognitive phenomena with various levels of biological plausibility, Nengo has a proven track record modelling dynamic, real-time, and large-scale behaviours (Bekolay et al., 2014). By decoupling network description and simulation, Nengo is also agnostic to the

[1] https://github.com/nengo/nengo
[2] July 26, 2019

backend implementation and allows the user to target several hardware backends including CPUs and GPUs by leveraging PyOpenCL (Klöckner et al., 2012); various neuromorphic hardware with custom backends; and now FPGAs (with this work). Figure 2.2 illustrates the decoupled architecture of Nengo at a high level, albeit with some backends omitted to avoid unnecessary clutter.

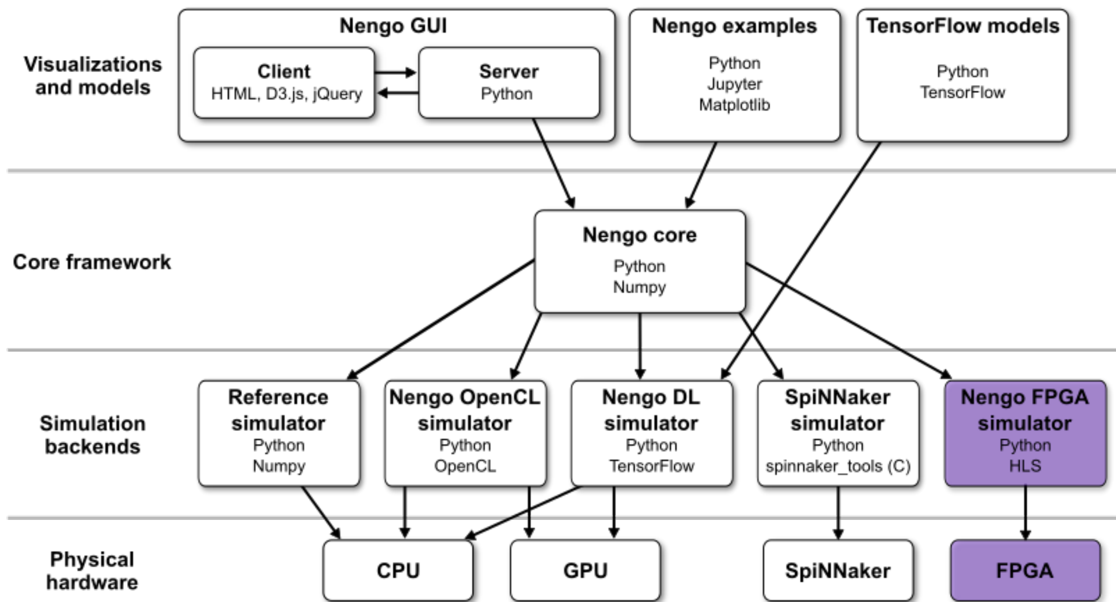


Figure 2.2: Architecture of the Nengo ecosystem showing the separation of the frontend description using Nengo core and multiple backends that execute on different hardware.
* Graphic adapted from Nengo Documentation (https://www.nengo.ai/documentation/)

In addition, this decoupling allows frontend add-ons. The deep learning extension of Nengo, NengoDL (Rasmussen, 2018), allows the integration of external network description from TensorFlow (Abadi et al., 2015) for example. The Nengo ecosystem also seamlessly integrates into the interactive Jupyter Notebook (Kluyver et al., 2016) system to facilitate explanation and exploration networks and topologies. In that vein, a key feature of Nengo is the optional integrated Graphical User Interface (GUI) that helps to develop and visualize computations. Figure 2.3 shows how the GUI displays a simple learning network including ensembles of neurons, connections, and real-time values from the simulation. The implementation described in this paper encompasses the *pre* ensemble seen in Figure 2.3 as well as the learning rule implemented on the pre–post connection. That is to say we accept an input (*stim*) that is represented internally (*pre*) then passed through the pre–post

connection weights to produce an output. The error signal that drives the decoder update for online learning is also consumed on the FPGA (green dashed line).

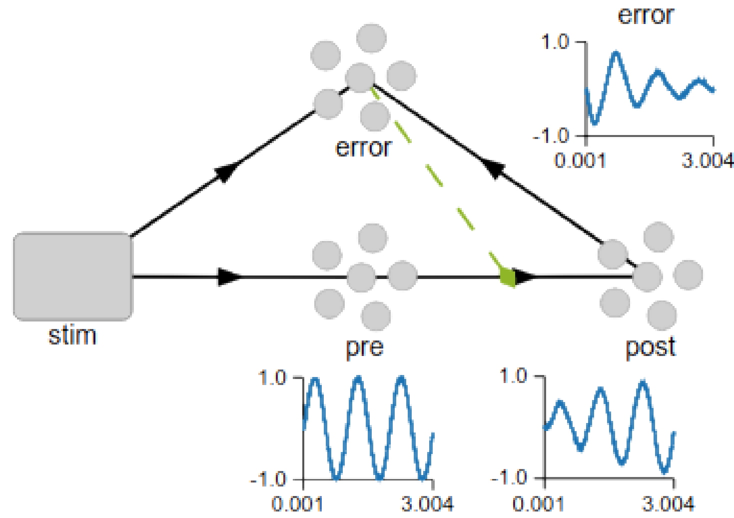


Figure 2.3: The Nengo GUI displaying an adaptive model with neural ensembles *pre*, *post*, and *error*; and with input *stim*, which is a sine wave. Black solid lines show connections and the green dashed line represents the error signal used by the learning rule to updates the decoders. The plots show the represented value on the vertical axis and the simulation time on the horizontal axis.

2.3 Neuromorphic Computing

Typically when we think of computers, and computing in general, we imagine the typical von Neumann architecture which is the basis of nearly all modern computing and a technology upon which we rely. Our world is becoming increasingly data-driven and emergent research into machine learning and neural networks increases the demand on compute resources. This increased demand coupled with the eventual demise of Dennard scaling (Dennard et al., 1999) and Moore's law (Moore, 1965) has left the traditional von Neumann architecture hard pressed to cope going forward. It is estimated that the human brain, in all it's computational majesty, consumes only ≈20 W of power compared to the 100's of watts required for modern CPUs and GPUs. Thus, researchers began development of compute platforms inspired by biology. This biologically inspired computing, to whatever extent it

may resemble biology, is dubbed *neuromorphic computing*. Beyond computational efficiency, there are several other driving factors for the development of neuromorphic hardware ranging from fault tolerance in computation to better understanding of neuroscience and biological computation (Schuman et al., 2017).

Modern, high-performance neuromorphic computing is still very much in it's infancy across both research and commercial sectors but the idea of these compute platforms is not new. In fact, the term *neuromorphic computing* was coined back in 1990 (Mead, 1990) in reference to a Very Large Scale Integration (VLSI) machine that used analog components to mimic a neural system. Today the term has broadened slightly to encompass both biologically inspired architectures as well as architectures driven by the demands of artificial neural networks (ANN) such as Google's Tensor Processing Unit (TPU) (Jouppi, 2016). Google has recently made their TPU technology available via the Coral Development board (Cass, 2019). Unfortunately, there was not enough time to evaluate this technology during this thesis work and similarly, it is not immediately obvious whether this ANN inspired processor is conducive to the more biologically plausible architecture of the NEF used in this work.

There are currently some more biologically inspired neuromorphic devices in existence leveraging various analog, digital, or mixed technologies, but none of these are readily available at the time of writing. There are some academically backed neuromorphic devices, such as SpiNNaker (Furber et al., 2014) from the University of Manchester or Braindrop (Neckar et al., 2019) from Stanford University, as well as some industry developed devices, such as IBM's TrueNorth (Merolla et al., 2014) and Intel's Loihi (Davies et al., 2018). Even if you are lucky enough to have access to one of these devices they are not easy to use. These devices, by definition, do not conform to the traditional von Neumann style of sequential instruction processing so it is not obvious how to program these devices with our current software paradigm. These devices all have their own unique API; however, many of them have also been connected to Nengo (Mundy et al., 2015; Voelker et al., 2017; GitHub:nengo-loihi, 2019; Fischl et al., 2018) as it provides separation between the frontend network description and the backend hardware implementation. These devices are not easily available for comparison as mentioned and moreover many of these systems employ different compute models which are not conducive to direct comparisons either. For example, Loihi and TrueNorth or both event-driven (asynchronous) systems that largely operate on synaptic events at the neuron granularity instead of the abstracted state-space ensemble granularity at which we operate.

The evolving nature of neuromorphic computing in conjunction with the multiple driving facets in the field has led us to explore implementation on the reconfigurable FPGA. An FPGA implementation is much more cost effective to implement than ASIC devices and furthermore, allows for reconfigurability whether it be minor tweaks for testing and optimization or entirely new implementations as new research develops. FPGAs also have the advantage of being readily available off-the-shelf devices and are (relatively) cost effective. As the Nengo software project grows in popularity and continues to support a wide array of different hardware backends, it seems like an obvious choice as a frontend connection to the FPGA architecture we develop in this work.

2.4 High-Level Synthesis

Custom hardware platforms, including FPGAs and ASICs, provide vast potential for high-performance and low-power computing; however, the development of these custom hardware platforms is time intensive and costly. Hardware designs are typically done using a Hardware Description Language (HDL), such as Verilog or VHDL, at the register transfer level (RTL). This approach offers the designer impeccable control over hardware resources and timing but is tedious in terms of code development and requires designers to have in-depth knowledge of hardware components. In order to mitigate the challenges associated with hardware development using HDL, researchers and industry alike began to develop what is known as High-Level Synthesis (HLS).

In much the same way software programming languages such as C and C++ offer a more user-friendly layer of abstraction on top of bare assembly language for CPU programming, HLS languages and compilers offer a more user-friendly layer on top of bare HDL for FPGA and/or ASIC programming. There are various approaches from academia, such as LegUp (Canis et al., 2011) and Chisel (Bachrach et al., 2012), as well as vendor developed systems, such as Xilinx's Vivado HLS (Feist, 2012) and Intel's (formerly Altera's) FPGA SDK for OpenCL (Czajkowski et al., 2012). These 4 examples all extend existing general purpose programming languages, namely C/C++ (LegUp and Vivado HLS), Scala (Chisel), and OpenCL (Intel's FPGA SDK); though many other approaches exist using domain specific languages or creating a new language altogether (Nane et al., 2016). By extending existing programming languages, these HLS frameworks begin to cater not to hardware programmers specifically, but to the overwhelmingly more numerous software

programmers (Bureau of Labor Statistics, 2016). Furthermore, these language extension frameworks are able to make use of existing libraries and tools.

The use of HLS for hardware development effectively relaxes the requirements of hardware developers: it is possible for software developers to foray into hardware more easily and even for experienced hardware developers this workflow can reduce development time. These benefits, of course, come with a cost. Trading-off the verbosity and complexity of the typical HDL design flow in favour of HLS inherently relinquishes some control to the compiler and therefore it is likely that HLS-built application will underperform as well as consume more hardware resources. This resource and performance cost can be lessened by optimizations and creative code structure specific to the particular framework, target hardware, and application (Huang et al., 2013). Unfortunately, to effectively optimize HLS designs, the developer once again requires knowledge of hardware components as well as framework specific experience and knowledge by which to convince the compiler to create certain hardware structures.

The use of HLS workflows as they are at the time of writing allow rapid development of *functional* hardware designs, though achieving good performance still requires some domain specific knowledge. It is important to note that HLS is very much in it's infancy and if we consider the development of higher level programming languages as precedent, the future is bright for HLS (Cong et al., 2011). Consider the effort and development that went into moving from assembly level programming to the C programming language. It took years of development and skepticism (Ritchie, 1993) and yet today trade-offs of using a C compiler instead of directly programming in assembly are rarely considered because the performance of modern day C compilers is tremendous.

2.5 PYNQ

The **Py**thon productivity for Zy**nq** (PYNQ) framework is Xilinx's foray into reducing the development effort needed for creating a usable FPGA accelerator and integrating it into various applications. PYNQ targets the Xilinx Zynq family of chips which are all System on a Chip (SoC) devices. These Zynq SoC devices pair FPGAs with multi-core ARM processors in order to provide a flexible, deployable platform. FPGA development is in and of itself an esoteric process with comparatively few accomplished developers (Bureau of Labor Statistics, 2016), but even if you are provided with a working hardware design there

is still much effort to be sunk into creating an interface. Integrating FPGA hardware into an application requires low-level embedded design which has been traditionally done using C or C++ languages. A 2018 survey of programming languages done by IEEE claims that Python has (marginally) surpassed C/C++ as the top programming language for the second year in a row (Cass, 2018). As such, it is fitting that Xilinx is marrying FPGA development with Python.

The overarching goal of the PYNQ project is to align hardware development with the more streamlined software development process. Instead of creating new designs for each new application, hardware developers can instead begin to create libraries. These hardware libraries, or *overlays*, can be loaded much like software libraries and then the PYNQ API (Xilinx, 2019c) can be used to easily interface with the FPGA via Python. Necessarily, these hardware libraries will still require development by an experienced hardware developer, but perhaps with an accessible Python API and attractive framework this new paradigm will be adopted more readily. The Python API provides pre-built functionality for many common use cases, including data movement and I/O interfacing, and has many example applications available (*e.g.* Corradi 2018; Xilinx 2019b). PYNQ is also designed to work with Jupyter Notebooks which may further entice developers as they can easily share implementations embedded within the descriptive and visual Jupyter environment.

Chapter 3

NengoFPGA

3.1 FPGA Architecture

All work done for this implementation used Vivado and Vivado HLS version 2016.1 and was evaluated on a PYNQ-Z1 device using the PYNQ API version 2.1.

In this section, we describe the design and engineering of NengoFPGA, a new FPGA backend for Nengo. This work presents a limited proof of concept implementation wherein a single ensemble of neurons with online learning is evaluated on the FPGA and the remainder of the network is evaluated on the host CPU as normal. As such, connectivity and synapses beyond the input and output of the single ensemble and the error signal for the contained learning rule are not implemented in hardware.

The foundations of the NEF approach to representation and transformation shown in Equations 2.1, 2.2, and 2.3 are embodied in the basic sketch shown in Figure 3.1. Note that there is a slight modification from Equation 2.1: to save memory and computation, instead of storing both the encoders and gain separately we instead pre-compute the *scaled encoders*, $\boldsymbol{E}'$, where each row of the encoder matrix, $\boldsymbol{e}_i$, is scaled by the corresponding gain term, α_i.

First we discuss at a high-level the parameterization of the design in Section 3.1.1 then we move to a lower level as we discuss the evolution of the architecture in Section 3.1.2.

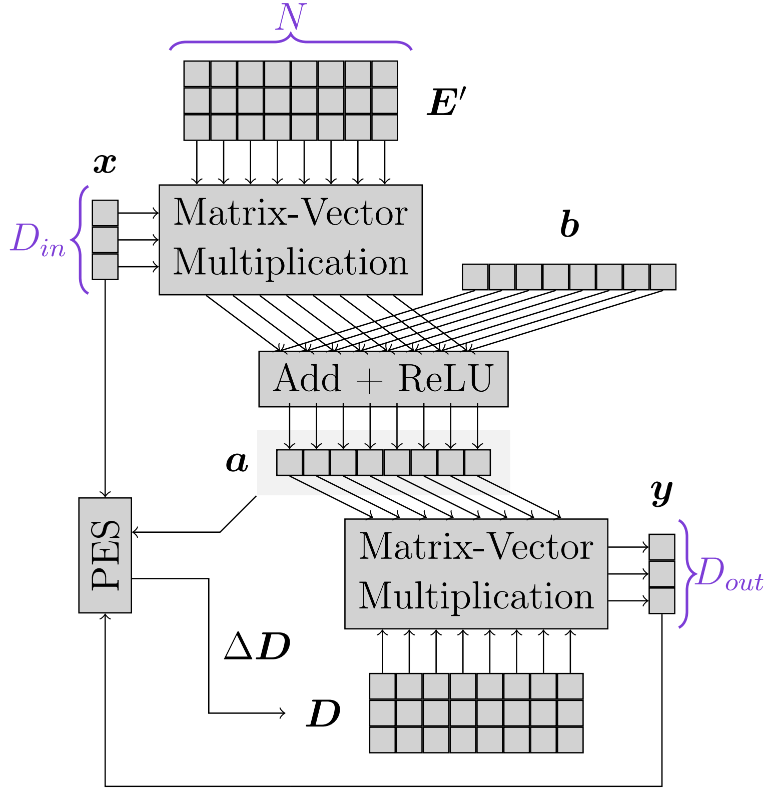


Figure 3.1: High-Level diagram of dataflow in the Neural Engineering Framework (NEF) network evaluation. We consume inputs ($\boldsymbol{x}$), perform matrix-vector multiplication with the scaled encoders ($\boldsymbol{E'}$), add the bias ($\boldsymbol{b}$) to the result and generate the activity ($\boldsymbol{a}$) vector. The result ($\boldsymbol{y}$) is generated through another matrix-vector multiplication with the decoders ($\boldsymbol{D}$). The scaled encoder matrix is generated randomly, while the decoder matrix is trained and updated by the online Prescribed Error Sensitivity (PES) learning rule.

Section 3.1.3 discusses the implementation of fixed-point data types and Section 3.1.4 discusses our method for automatically optimizing fixed-point hyper-parameters.

3.1.1 Design Parametrization

To support the use of NengoFPGA in various embedded scenarios, we first target run-time parametrization of the design. We identify the model size parameters as the primary factors that determine the neural network configuration. We leave the number of neurons, N,

and the input and output dimensionality, D_{in} and D_{out}, as parameters. We also leave the learning rate, k, as a parameter to tailor the hardware to specific applications, including those without learning (*i.e.* $k = 0$). Full on-chip storage is used in embedded contexts to eliminate external memory accesses during operation and to enable rapid response to real-time events from the outside world. Although the model size parameters we identified are easily implemented at run-time, the weight matrices, $\boldsymbol{E}'$ and $\boldsymbol{D}$, and the bias vector, $\boldsymbol{b}$, associated with those parameters must be stored on chip and require their memory sizes to be fixed at compile-time (not run-time). Although N, D_{in}, and D_{out} remain run-time parameters, we must choose maximum values for each of these at compile-time in order to allocate on-chip memory. Another compile-time parameter that dictates parallelism in the compute pipeline is extracted as well and is labelled $UFAC$, or *unroll factor*. Since these compile-time parameters dictate the architecture of the run-time parametrized hardware, we call these *hyper-parameters*.

This leaves us with a set of **run-time** parameters:

- number of neurons, N;
- input dimensionality, D_{in};
- output dimensionality, D_{out}; and
- learning rate, k.

As well as a set of **compile-time** hyper-parameters:

- maximum input or output dimensionality, D_{max}, used to allocate memory for the input and output vectors ($\boldsymbol{x}$ and $\boldsymbol{y}$ respectively);
- maximum number of neurons, N_{max}, used to allocate memory for the bias vector ($\boldsymbol{b}$);
- maximum weight matrix size, ND_{max}, used to allocate memory for the scaled encoders and decoder matrices ($\boldsymbol{E}'$ and $\boldsymbol{D}$ respectively); and
- the unroll factor, $UFAC$, used to determine the degree of parallelism in the compute pipeline.

Note that ND_{max} need not equal $N_{max} \times D_{max}$. This allows additional flexibility at run-time in cases where we choose large N and small D or vice versa. For example, say we set D_{max} = 1k and N = 16k. If we set $ND_{max} = N_{max} \times D_{max}$ = 16M we create

an unreasonable resource demand while underusing this memory in most cases. Now if we instead set $ND_{max} = 32$k for example, we could feasibly achieve this by setting $D_{max} = 8$ and $N = 4$k and imposing these conservative limits. Instead we choose to have $ND_{max} \neq N_{max} \times D_{max}$, say $ND_{max} = 32$k, $D_{max} = 1$k, and $N = 16$k, which allows flexibility to choose larger dimensionalities while still abiding to our memory budget. Furthermore, by choosing these values independently we make more efficient use of the memory allocated for weights as there are multiple (N, D) combinations that fill the memory instead of the single (N_{max}, D_{max}) pair.

The compile-time hyper-parametrization allows for rapid design-space exploration as well as easily tunable implementations that target particular applications. Typically we choose D_{max}, N_{max}, and ND_{max} to fill the entire chip, unless area or resources are reserved for auxiliary functions. However, these values are related to the last hyper-parameter, $UFAC$. Necessarily, having more parallel compute pipelines requires more resources for evaluation, but indirectly, as we increase the number of parallel pipelines ($UFAC$), it is necessary to replicate the input and output vectors by the same factor in order to service each pipeline. This overhead is visually displayed in Figure 3.2.

The real flexibility of the design stems from the run-time parameters. The ability to set model parameters, such as number of neurons or dimensionality, as well as neuron characteristics, such as encoders and bias, at run-time allows a single compiled design to be used in multiple applications. In order to efficiently load all of these values into on-chip memory, we employ a run-once initialization and then allow the hardware to run without host interaction henceforth[1]. The implementation is split into two distinct steps: 1) Loading parameters and 2) Evaluation. This is summarized at a high level in Algorithm 3.1.

3.1.2 HLS Formulation

A simple and naive implementation of the NEF is shown in Algorithm 3.2 and consists at its core of two matrix multiplies: 1) encoding the input as neural activity; and 2) decoding the neural activity to create the output.

Efficient, scalable parallelization of the NEF equations is the primary role of the HLS description. While the parallelism potential is abundant in the matrix-vector operations, it is not trivial to harness this parallelism for several reasons, including:

[1]Assuming I/O is connected directly to the FPGA via GPIO or another interface.

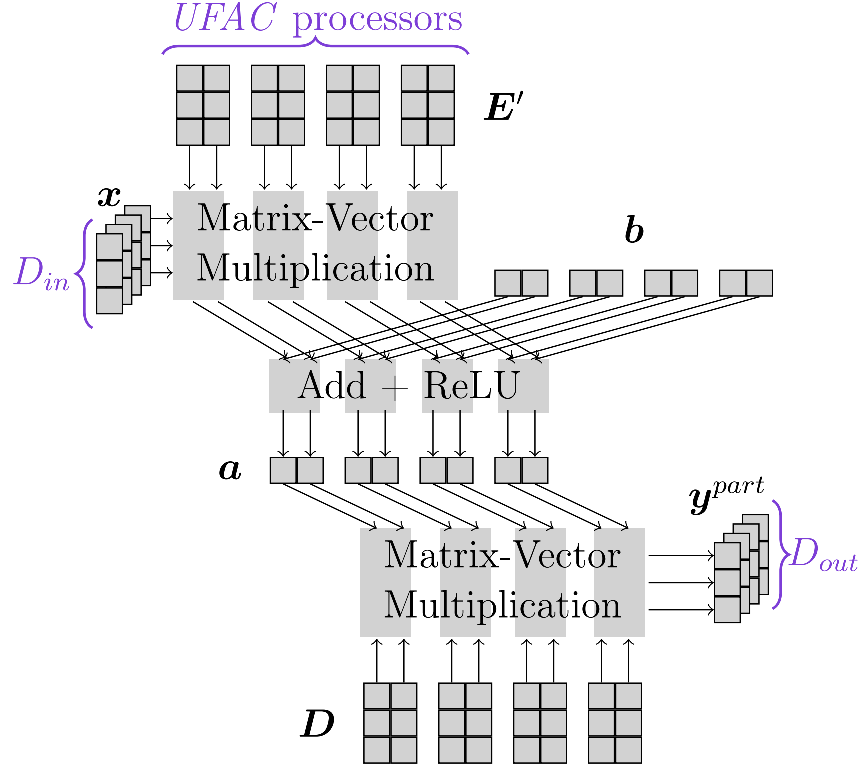


Figure 3.2: High-Level diagram of dataflow in the parallel network evaluation. The encoders ($\boldsymbol{E}'$), bias ($\boldsymbol{b}$), and decoders ($\boldsymbol{D}$) are partitioned across the $UFAC$ parallel compute pipelines. In order to service the parallel pipelines, the input ($\boldsymbol{x}$) is replicated for each parallel channel. Similarly, multiple output channels are required to read the partial results from each channel ($\boldsymbol{y}^{part}$).

Algorithm 3.1: Load–evaluate workflow of the FPGA implementation where `LOAD` is done once on initialization and the `else` condition is the evaluation that receives an input and an error and produces an output every step.

---

1 **if** *`LOAD`* **then**
2   Store parameters on chip
3 **else**
4   Read input and error
5   Encode input → neural activity
6   Decode neural activity → output
7   Write output

- A trick used to pack compute resources is batching. However, in our case batching presents a challenge due to the temporal dimension of the NEF. The NEF (and this implementation) is geared towards dynamic systems whose inputs vary and may not be known ahead of time (*i.e.* for batching). Similarly, this work employs online learning which has a temporal feedback dependency.
- The encode and decode loop topologies are inverted. Loops over N are followed by D_{in} for encode, while loops over D_{out} are followed by N for decode. Due to this structure, we cannot trivially partition the full encode–decode pipeline across the same dimension. To maximize the extent of parallelism for both the encode and decode computations, we aim to split the tasks across the N neurons. This is because in typical NEF formulations $N \gg D_{in}$ and $N \gg D_{out}$.
- The decode step from Equation 2.2 requires a contribution from each neuron to create the output, $\boldsymbol{y}$. Thus, in order to parallelize across N, we have to restructure our code. The nested loops of the decode step are inverted and a partial result, $\boldsymbol{y}_k$, is generated in each parallel section. After the decode step, the partial results are then accumulated into the single output, $\boldsymbol{y} = \sum_k \boldsymbol{y}_k^{part}$.
- Each parallel section is allocated an independent portion of the encoder, $\boldsymbol{E}'$, and decoder, $\boldsymbol{D}$, weights. However, the input, $\boldsymbol{x}$, and error signal, $\boldsymbol{Err_x}$, must be shared between threads.
- Having parametric loop bounds (*i.e.* N, D_{in}, and D_{out}) presents challenges regarding loop unrolling and partitioning since the problem structure is not static.

We start with our basic sketch of the implementation shown in Algorithm 3.2 and discuss incremental strategies used to address these parallelization challenges.

Though there is some overhead associated with the `LOAD` step in Algorithm 3.1, this is a run-once process so it is the evaluation time for one pass through the computation flow in Figure 3.1 that should be minimized to allow the system to respond to real-time signal characteristics for a large number of neurons and dimensions. Recall, we start with fixed values for scaled encoders ($\boldsymbol{E}'$) and bias ($\boldsymbol{b}$) elements which are selected at random. The input ($\boldsymbol{x}$), activity ($\boldsymbol{a}$), and output ($\boldsymbol{y}$) quantities vary with each timestep. The decoder matrix ($\boldsymbol{D}$) is also updated once per timestep using the PES rule and replaces conventional backpropagation with an online learning approach. When interfacing with sensors and

Algorithm 3.2: A basic sketch of the NEF implementation without consideration for hardware or HLS structure or limitations.

```
1 for i < N do    // Encode
2 |  a_i = b_i
3 |  for j < D_in do
4 |  |  a_i += x_j * e^s_ij
5 |  a_i = G(a_i)    // ReLU
6 for j < D_out do    // Decode
7 |  for i < N do
8 |  |  y_j += a_i * d_ji
9 |  |  d_ji += -k * Err_{x_j} * a_i    // Learning
```

actuators, the acquisition of inputs and projection of outputs may present additional time penalties on various platforms, but for this analysis we assume I/O presents zero latency and focus on the compute pipeline alone. Before exploring the evolution of our HLS description, we do a quick back-of-the-envelope calculation to estimate the required number of cycles given D_{in} inputs, N neurons, and D_{out} outputs.

Cycle estimations are referring to Algorithm 3.2. If we consider only the core encode and decode steps on lines 4 & 8 we require:

$$(D_{in} + D_{out}) \times N \tag{3.1}$$

We are using 32 bit floating-point numbers, so to be conservative we allow 3 cycles for both of these load-compute-store operations:

$$[(D_{in} + D_{out}) \times N] \times 3 \tag{3.2}$$

Now consider the addition of the bias term on line 2 and the evaluation of the Rectified Linear (ReLU) transfer function on line 5 for each neuron. We arrive at:

$$[(D_{in} + D_{out}) \times N] \times 3 + (2 \times N) \tag{3.3}$$

Finally we account for the online learning update using the PES rule on line 9. This is an additional load-compute-store operation inside our decode loop, so we increment our inner parentheses (alongside the original decode step) and conclude with:

$$[(D_{in} + 2 \times D_{out}) \times N] \times 3 + (2 \times N) \tag{3.4}$$

For the subsequent discussion regarding performance, we consistently compile designs with $N_{max} = 4\text{k}$, $D_{max} = 8$, and $ND_{max} = 32\text{k}$ for simplicity and we use $N = 200$ and $D_{in} = D_{out} = 2$ for all cycle count comparisons. Plugging these into Equation 3.4, we should expect the evaluation of one timestep to take ≈ 4000 cycles.

For the first-pass HLS implementation the two decode loops are perfectly nested and are automatically flattened and pipelined having an initiation interval (II) of 6. Due to the addition of bias and the application of the ReLU transfer function, the encode loops are not perfectly nested and therefore are not flattened and pipelined automatically. The inner encode loop over D_{in} is pipelined and has $II = 5$. The outer loop can only be pipelined if the inner loop is fully unrolled, but in our case we have a variable loop bound, and thus the execution of the inner loop is not known at compile-time and cannot be properly unrolled. Consequently, the outer encode loop over N cannot be pipelined as the internal workload is unknown and this contributes notably to the poor initial performance. As a result, this naive design requires 6026 cycles which is much worse than our estimate of only 4000 cycles.

To overcome the limits of the HLS compilation, we have to supply additional information to the compiler as explored by Huang et al. (2013). We perform the following optimizations guided by the need to describe HLS code with care:

1. Restructuring of the decode loops to match the encode structure which favours parallelization over N as it is the larger dimension and contains the most parallelism.
2. Unroll-friendly description of the design with an explicit third loop level that the compiler can recognize as parallelizable.
3. Dataflow concurrency directives (*i.e.* pragmas) attached to the proper loop bodies.

Restructuring

As observed earlier, maximum parallelism is available over the N neurons. This suggests the need to reformulate the decode loops accordingly. In the first restructuring step, the nested decode loops are inverted as seen in Algorithm 3.3. We do not fuse the outer encode and decode loops over N to ensure the inner loops are fully optimized in isolation. With the restructured loops, we also introduce a local activity value, a_{loc}, on line 6 to better exploit data reuse. Since we are using the simple Rectified Linear neuron model, we move G into the decode loop as a ternary operation (*i.e.* a_{loc} `> 0 ?` a_{loc}`:0`) on line 6 to simplify the encode loop. After these modifications the performance of the encode step remains unchanged, still unable to be fully pipelined, but the fully pipelined decode step improves two-fold, now boasting an $II = 3$, with the addition of data reuse over a_i. Overall, this improves the performance of the design which now requires 4829 cycles.

Algorithm 3.3: A sketch of the NEF implementation with the decode step restructured to more closely match the structure of the encode step. The decode loops are inverted and the ReLU transfer function is moved inside the decode loop.

```
1 for i < N do    // Encode
2     a_i = b_i
3     for j < D_in do
4         a_i += x_j * e^s_ij
5 for i < N do    // Restructured decode
6     a_loc = G(a_i)    // ReLU inline
7     for j < D_out do
8         y_j += a_loc * d_ji
9         d_ji += -k * Err_{x_j} * a_loc    // Learning
```

Explicit Parallelization

Once the outer loops have been harmonized and both the encode and decode steps begin with loops over N, it may seem tempting to simply apply the `UNROLL` pragma followed by `DATAFLOW` to evaluate the unrolled loop copies in parallel. However, Vivado HLS was not able to correctly identify the independence of the parallel sections and produced sequential hardware while still using the larger resource cost associated with the dataflow

framework. To overcome this limitation, we factor out an explicit outer loop for unrolling to make the parallelism more obvious to the compiler. This loop for unroll factor, $UFAC$, is introduced as shown in Algorithm 3.4 and explicitly partitions the encode and decode loops across their N-dimension. The use of the `UNROLL` pragma in this new loop now successfully creates multiple pipelines. This alone was not sufficient to improve performance though. Using the `ARRAY_PARTITION` pragma, the network weights, $\boldsymbol{E}'$ and $\boldsymbol{D}$, required by each parallel section were partitioned along their N-dimension. Unfortunately, even with the additional outer loop and this pragma, the compiler was still unable to recognize the independence of the partitioned data and did not allow concurrent memory access. The weights and biases instead had to be explicitly partitioned using an extra dimension in the array structure, much the same way the loops required an additional layer to be unrolled. For example, the $N \times D_{in}$ scaled encoder matrix, `enc[N][D]`, is reshaped and becomes `enc[UFAC][N/UFAC][D]`). Finally, this three-dimensional structure along with the `ARRAY_PARTITION` pragma *was* understood by the compiler for concurrent access. To keep each parallel section supplied with data, the input, $\boldsymbol{x}$, and error signal, $\boldsymbol{Err_x}$, must be replicated $UFAC$ times – one for each parallel section.

Algorithm 3.4: A sketch of the NEF implementation that has been restructured for parallelism by explicitly partitioning the work over the N neurons using an additional outer loop. Consequently, additional logic is required to accumulate the partial results of each partition.

1 **for** $k < UFAC$ **do** `// Explicit partitioning`
2   **for** $i < \lceil \frac{N}{UFAC} \rceil$ **do** `// Encode`
3         $a_i = b_{ki}$
4   **for** $j < D_{in}$ **do**
5             $a_i \mathrel{+}= x_{kj} * e^s_{ij}$
6   **for** $i < \lceil \frac{N}{UFAC} \rceil$ **do** `// Restructured decode`
7         $a_{loc} = G(a_i)$ `// ReLU inline`
8   **for** $j < D_{out}$ **do**
9             $y^{part}_{kj} \mathrel{+}= a_{loc} * d_{ji}$ `// Partial result`
10            $d_{ji} \mathrel{+}= -k * Err_{x_{kj}} * a_{loc}$ `// Learning`
11 **for** $j < D_{out}$ **do** `// Accumulate partial results`
12     $y_j = 0$
13   **for** $k < UFAC$ **do**
14         $y_j \mathrel{+}= y^{part}_{kj}$

Having explicitly described the parallelism manually, we must similarly handle edge cases explicitly as the compiler is not responsible for the parallel structure in this case. We add extra logic to handle the cases when $UFAC$ is not a factor of N by zero-padding arrays and computing loop bounds as the ceiling $i < \lceil \frac{N}{UFAC} \rceil$. The output, $\boldsymbol{y}$, requires a contribution from each neuron as seen in Equation 2.2 and therefore by partitioning along the N-dimension, each parallel section now creates a partial result for the output, $\boldsymbol{y}^{part}$. These partial results must then be handled explicitly and so an additional accumulator loop is added as seen in Algorithm 3.4. Luckily, the inner loop over $UFAC$ has a known bound at compile-time allowing the accumulator loop to be automatically pipelined. Figure 3.3 shows the computation schematic of this partitioned design.

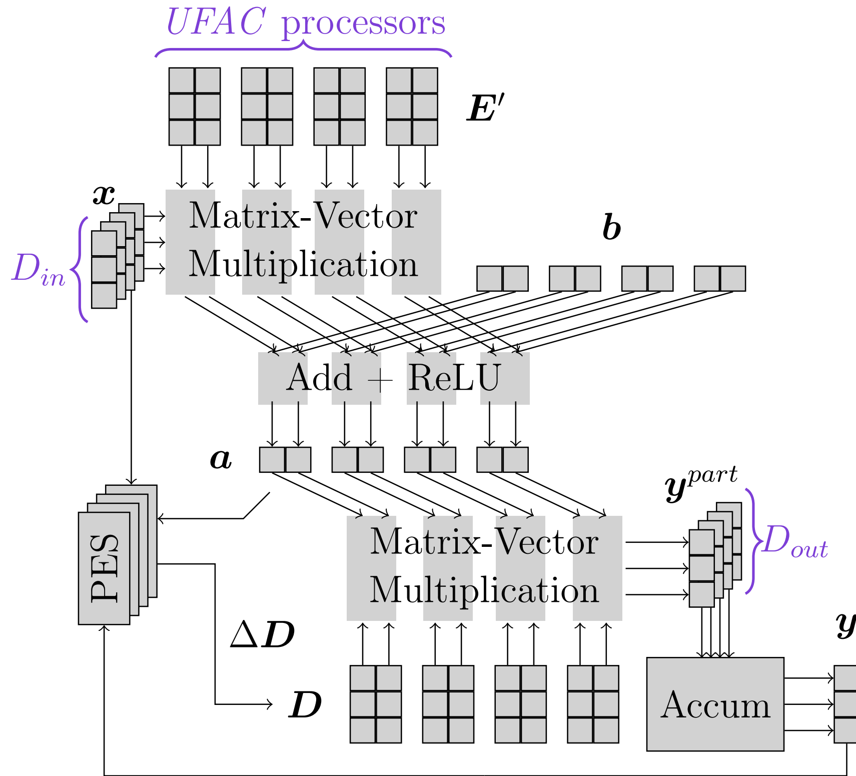


Figure 3.3: High-Level diagram of dataflow in the parallel network evaluation. The encoders ($\boldsymbol{E}'$), bias ($\boldsymbol{b}$), and decoders ($\boldsymbol{D}$) are partitioned across the $UFAC$ parallel compute pipelines. In order to service the parallel pipelines, the input ($\boldsymbol{x}$) is replicated for each parallel channel. Similarly, multiple output channels are required to read the partial results from each channel ($\boldsymbol{y}^{part}$) which are then accumulated into a single coherent output ($\boldsymbol{y}$).

When compiled with $UFAC = 12$[2] the performance of the encode and decode loops does not change, staying with $II = 5$ and $II = 3$ respectively. However, we do see 12 instances of each loop that handle a portion of the work, albeit sequentially. Without the strategic addition of the `DATAFLOW` pragma the compiler is unable to run the parallel sections concurrently. We are now poised to take advantage of the parallel problem description, however at this stage we maintain largely sequential performance while still paying the overhead of multiple processors, the accumulator loop, the input replication, and the additional control logic to handle the same. This leaves us still requiring a total of 4817 cycles at this stage.

Dataflow

We now have the layout for the parallel design ready and the challenge remains to convince the compiler to use concurrent dataflow execution. The `DATAFLOW` pragma optimization attempts to run all blocks within a module or loop concurrently using dataflow analysis to identify dependencies. Unfortunately, adding this pragma to the outer $UFAC$ loop was unsuccessful. It appears as though the scope of the `DATAFLOW` pragma was poorly enforced and the compiler encountered difficulties integrating dataflow within the main compute body with the surrounding support logic. To rectify this, the main compute body (*i.e.* the outer loop over $UFAC$) was moved into its own separate function. Although the code description had not changed beyond the addition of the function abstraction, this was enough to appease the compiler and dataflow was successfully applied within this new compute function. Variable scope also plays an important role it turns out. The final modification that successfully created parallel, concurrent hardware was to declare the activities vector, $\boldsymbol{a}$, within this new compute function as opposed to declaring it along with the other memory structures with broader file scope.

Upon examining the HLS reports we find that the compiler has generated $UFAC$ copies of separate encode and decode processors. The two encode loops have been successfully flattened and pipelined in each processor showing an $II = 6$. Previously the encode inner loop had an $II = 5$ but the outer loop was not pipelined. Successfully merging the two loops and fully pipelining the encode step is therefore worth the cost of a reduced II in this case. The decode processors maintain their pipelined execution with an $II = 3$.

[2]This value is chosen as $UFAC = 12$ is the largest factor used to successfully compile the final floating-point design.

The parallel hardware was successfully compiled with $UFAC = 12$. These improvements have progressed from the original 6026 cycles using a naive approach to the current design which shows an improvement of 15× requiring only 387 cycles! A comparison of resource usage and required cycles is summarized in Table 3.1.

Stage	Cycles	Resource Usage (%)		
		BRAM	DSP	LUT
Basic	6026	51	5	5
Restructured	4829	52	5	5
Explicit Parallelsim ($UFAC = 12$)	4817	90	22	35
Dataflow ($UFAC = 12$)	387	98	98	81

Table 3.1: Summary of performance and resource usage during the evolution of the HLS code description.

Overall, the HLS optimization process was a tedious journey. The tools are still in their infancy and so it is expected to face some challenges with the compiler, but some of this creative code description can and should be built into the compiler. The two biggest sticking points were the scope of the `DATAFLOW` pragma and the inability of the compiler to recognize independent data partitions. It was necessary to add a function level abstraction to get the dataflow optimization implemented on a subset of the code. I imagine the `DATAFLOW` pragma should be able to automatically figure out the largest possible scope to which it can be successfully applied, or at the very least should be able to properly recognize loop scope instead of function scope. Furthermore, it was necessary to add an explicit outer loop for parallelism as well as an explicit extra dimension for array partitioning. The HLS tools should be able to understand directives to unroll compute pipelines and partition data without explicitly adding extra dimensions to the problem.

Direct I/O Access

In addition to HLS optimizations, direct I/O access is achieved by adding ports of appropriate width to the module with no protocol. Physical pins from the board package are then connected directly to the module with an XDC file. NengoFPGA can use this strategy to connect to on-board peripherals or the physical world using GPIO pins and by doing so, forego any latency associated with data transfer through the host. This presents the opportunity for vast performance improvement in end-to-end applications as will be discussed in Section 3.2.1.

3.1.3 Fixed-point Design

Floating-point numbers support representation of numerical quantities with high dynamic range, but neural network operations are often amenable to reduced fixed-point precision with little, if any, loss in accuracy (Nakahara et al., 2019; Umuroglu et al., 2017). For our NEF implementation we make use of the Xilinx `ap_fixed` library to define our fixed-point types. We use multiple independent fixed-point data types for different regions of the compute pipeline. This gives us more resolution to select the required precision efficiently as dictated by the different stages of computation. We identify 7 different fixed-point data types to be used in the design:

- `DATA_T_PORT` – used as the data type on the external interfaces. The AXI and DMA controllers on the interface require word sizes to be powers of 2, so this type was separated in order to allow more freedom when selecting the remaining precisions.
- `DATA_T_K` – used for the learning rate parameter, k. The learning rate is, for all intents and purposes, always less than 1 which allows us to easily restrict the number of integer bits thus reducing the search space.
- `DATA_T_IN` – used to store the input vector, $\boldsymbol{x}$.
- `DATA_T_ERR` – used to store the error vector, $\boldsymbol{Err_x}$.
- `DATA_T_DEC` – used to store the decoder weights, $\boldsymbol{D}$.
- `DATA_T_ENC` – used to store the scaled encoder weights, $\boldsymbol{E'}$, as well as the bias vector, $\boldsymbol{b}$.
- `DATA_T_RES` – used internally to represent intermediate results as well as the activity vector, $\boldsymbol{a}$.

Furthermore, to ensure accurate scaling of the quantities as run-time parameters change (N, D_{in}, D_{out}, and k), we introduce a new hyper-parameter, `K_SHIFT`, that normalizes the values depending on the learning rate, k. The learning rate that is stored on-chip is shifted left by `K_SHIFT` bits and this shifted value is used throughout the computation. The output produced from the timestep evaluation is then shifted back right to recover the original signal magnitude. The proper selection of this scaling parameter reduces the number of fractional bits required (Jacob et al., 2017b) and leads to a leaner, more flexible design. However some small values may still be unavoidably quantized to zero depending on the

selected precision.

The switch to fixed-point representation further reduces the cycle count by promoting a more efficient use of resources and allowing an unroll factor of 28 ($UFAC = 28$). This leads to over 3× improvement compared to the floating-point design: down to 114 cycles from 387. Notice that we see an improvement in hardware efficiency with the new fixed-point design beyond the linear speedup due to moving from $UFAC = 12$ to $UFAC = 28$. This result is achieved with the P4-C set of fixed-point hyper-parameters found in Table A6.

Choosing these new fixed-point hyper-parameters presents a challenge in and of itself as the design space is quite large. Each `ap_fixed` data type we define has four template arguments:

1. Total number of word bits.
2. Portion thereof that are integer bits.
3. Rounding strategy.
4. Overflow strategy.

Let's assume we've fixed the rounding and overflow strategies for all of our data types. This leaves us with two free precision hyper-parameters per data type. Both the word bit and integer bit options accept (somewhat) arbitrary integer arguments, for simplicity let's assume for each we give a range of 30 (*i.e.* 2–32 bits). Assuming we define 7 distinct data types, each with two free hyper-parameters with 30 possible values each, this gives us on the order of $30^{14} \approx 10^{20}$ different design possibilities which is intractable for a brute-force search. As a result, a more efficient strategy was developed and is discussed in Section 3.1.4.

3.1.4 Parameter Tuning

We use a hyper-parameter tuning package called Hyperopt developed by Bergstra et al. (2015) to automate the design space exploration process and discover the optimized hyper-parameter assignments quickly. Hyperopt determines an optimal design for the given constraints and also logs all progress, thereby: 1) showing convergence trends to determine how many trials are required; 2) making it possible to iterate on the search space, cost function, or simulation; and 3) making it possible to extract the Pareto optimal points for the given formulation.

Each fixed-point data type is defined by a number of integer and fraction bits, rounding approach, and overflow handling technique. The Vivado HLS `ap_fixed` type encapsulates all these specifications into a C++ template. We make an intuitive pre-selection of the rounding and overflow modes for our types to reduce the search space. For data storage (weights and bias) and transmission variables (inputs and outputs), we round towards $+\infty$, and saturate the overflow values. For internal arithmetic, we use truncation of small values below the least significant bit and wrapping on overflow to improve computation speed, albeit at the cost of accuracy and safety if we go beyond our dynamic range. This leaves us with the number of integer and word bits as free parameters and a design space size on the order of 10^{20}, as noted in Section 3.1.3. It may be possible to employ Hyperopt directly with the remaining vast design space; however, to improve convergence time we take a hierarchical approach which sets some initial bounds before applying Hyperopt to the problem.

Initial Bounding

In addition to fixing the rounding and overflow strategies we make two more simplifications before beginning the process of investigating the design space. First, we fix `DATA_T_K` using 32 word bits and 1 integer bit. The learning rate is a run-time parameter that is stored in a 32 bit register and so we select 32 word bits as they are available regardless. We also know the learning rate will typically be less than one, so we allocate nearly the entire representation as fractional bits. Next we fix the `DATA_T_PORT` type. We use the Xilinx provided DMA controller (Xilinx, 2019a) which operates on AXI Stream data types and restricts the data width to be a power of 2. This port data type is also used by multiple other data types in order to load parameters during the run-once initialization to exchange I/O during the evaluation of each timestep. Therefore this data type must be large enough to accommodate each type it services. We set 64 word bits and 16 integer bits. This reduce the design space by directly removing two more free hyper-parameters, but simultaneously this sets an upper bound to the number of word bits used by all data types serviced by the external interface, which is all types except the internal `DATA_T_RES` type.

To compute some rough bounds for the remaining search space, we initially evaluate the floating-point design as reference. Then we set each type to a large 64-bit fixed-point representation and select each fixed-point type individually for inspection to avoid the large combinatorial space. In each inspection, we sweep the candidate's fixed-point precision

(integer and fraction bits) from high (64 bits) to low (1 bit) in a grid search and observe when accuracy deviates from the reference floating-point design. We define accuracy in terms of the overall algorithm goal where we aim to minimize the absolute representational error $\boldsymbol{Err_x} = |\boldsymbol{x} - \boldsymbol{y}|$. We use the HLS implementation of an adaptive controller that learns to represent a sinusoid as the test case for evaluating error. The sinusoidal input effectively covers the domain of our input, $\boldsymbol{x} \in \{-1, \ldots, 1\}$. We also fix the learning rate, $k = 10^{-5}$, for all trials. Our floating-point reference design defines the reference error, $\boldsymbol{Err_x^{float}}$. We expect our fixed-point solution error, $\boldsymbol{Err_x^{fixed}}$, to be no worse than the reference floating-point solution given the same set of network weights. For each set of parameters, we identify the smallest precision where the floating-point error exceeds fixed-point error, $\boldsymbol{Err_x^{float}} \geq \boldsymbol{Err_x^{fixed}}$.

Optimization

Once our initial bounds are set, we perform the optimization using a given cost function. The HLS code that evaluates error is repurposed to also return the cost metrics (resources, and scheduled cycle counts) through a single step of the HLS C-synthesis compilation that generates RTL code. We do not run the expensive place-and-route phase at this point to speed up the search. The fastest design was found using the cost function that minimizes the error–cycles product (*i.e.* $\boldsymbol{Err_x} * \text{cycles}$), but other cost functions produced competitive results more quickly. We also use the cost function that minimizes the error–resource product (*i.e.* $\boldsymbol{Err_x} * \text{resources}$), where the resource cost is the maximum percentage of LUTs, DSPs, and BRAMs used. For this optimization we lock the unroll factor at 1 ($UFAC = 1$). The minimization of resources is proportional to overall cycle count since resource usage determines the degree of parallelism that is possible (*i.e.* $UFAC$). Thus this error–resource cost function produces designs balanced in accuracy and performance indirectly. In addition, using $UFAC = 1$ reduces the effort required by the HLS compiler for C-synthesis, which allows Hyperopt to run 3–4× faster than minimizing cycles directly while simultaneously reducing the size of the search space by omitting the $UFAC$ hyper-parameter. This improvement in convergence time only incurs a small performance penalty as will be discussed in Section 4.2.

Usage

The high-level Hyperopt configuration is show in Figure 3.4. The Hyperopt minimization algorithm used for this optimization was the default Tree-of-Parzen-Estimators (TPE) algorithm (Bergstra et al., 2011). Hyperopt exploration is configured in a Python script containing the structure of the problem in terms of search space and cost function. Hyperopt is then connected to a shell script that invokes the HLS C-synthesis and runs the adaptive controller accuracy test. The C-synthesis returns the estimated resource usage as well as estimated cycle counts for the given trial and the adaptive controller simulation returns our error metric. These values feedback to the Python script and the Hyperopt system iterates based on these metrics.

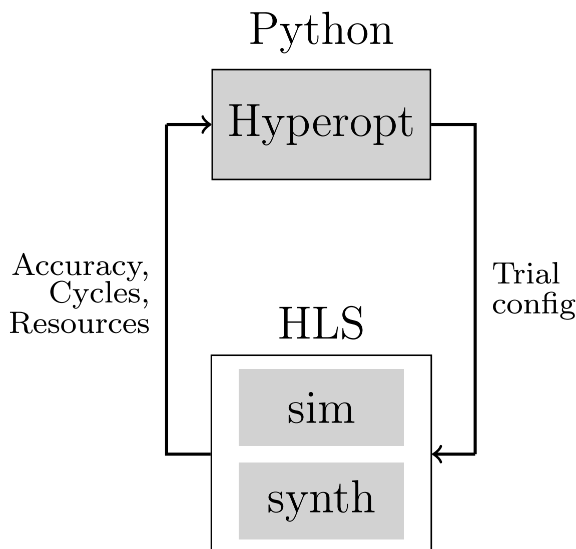


Figure 3.4: High-level workflow of the Hyperopt optimization process.

The most challenging part of configuring the Hyperopt system was choosing an appropriate cost function, though Hyperopt itself is robust and easily integrated with black box systems. In our case, optimizing for performance against resources was well defined but still offered a range of possible cost functions, it's possible for less well defined targets that the process of designing an effective cost function becomes a more involved process in and of itself. In any case, the Hyperopt system provides a high-level of abstraction in which it is easy to explore different cost functions and design spaces relatively quickly with ease.

3.1.5 Copy Protection

Since this work was developed in cooperation with Applied Brain Research[3], provisions were made for the commercialization process. Copy protection can be built into the compiled designs by locking execution to a particular device. Each Xilinx device has a unique read-only ID called the Device DNA. Each design is compiled with a reference value, whether it be an explicit Device DNA or a known function thereof, and during the run-once `LOAD` phase the Device DNA is compared to the reference value. Based on the result of this comparison, the `valid_id` flag is set which then gates the evaluation phase of the design as seen in Algorithm 3.5. This has minimal impact on performance as we only add a single bit check for the evaluation phase and have the fetching and comparison logic only run once on initialization. The Device DNA is retrieved using the publicly available IP core developed alongside the main body of this work (Morcos, 2019).

Algorithm 3.5: Load–evaluate workflow of the FPGA implementation with the additional copy protection step. The Device DNA is read once during the `LOAD` phase and is compared to a reference value in order to set the `valid_id` flag which gates execution.

```
1  if LOAD then
2  |   Read device DNA and set valid_id
3  |   Store parameters on chip
4  else if valid_id then
5  |   Read input and error
6  |   Encode input → neural activity
7  |   Decode neural activity → output
8  |   Write output
9  else
10 |   Do nothing
```

3.2 Interface

The core Nengo framework is directly integrated with the NengoFPGA Python package (Morcos et al., 2019) to seamlessly connect standard Nengo models directly to the FPGA backend. There are two modes by which to drive the FPGA implementation:

[3] https://appliedbrainresearch.com/

1. Via the on-chip ARM host.
2. Via a remote PC host.

In mode 1 we make use of the ARM processor packaged with the FPGA in the PYNQ SoC to drive the implementation and this will be discussed further in Section 3.2.1. For ease of use, we also add the ability to integrate the FPGA accelerator into Nengo models running remotely on a PC host. We call the PC the *superhost* in this context as we still rely on the ARM core as a local host and need to differentiate the two host systems. Mode 2 is discussed further in Section 3.2.2.

3.2.1 Direct ARM Host

As noted in the Statement of Contributions, this NengoFPGA software interface was refactored and cleaned up with assistance from Daniel Rasmussen and Trevor Bekolay.

In the most basic form, we implement the NengoFPGA backend split across the ARM CPU, and the FPGA fabric of the PYNQ-Z1 SoC. To use this environment, the neural network is first described using a typical Nengo network description. This model is run directly on the ARM host and the NengoFPGA extension uses the PYNQ API to load the network weight matrices to the FPGA's on-chip RAMs once at the start of a simulation. The PYNQ API makes use of the ubiquitous AXI protocol to stream data between the host and FPGA. This system-level description is diagrammed in Figure 3.5. Once initialized, the FPGA core can either receive inputs from the ARM CPU or directly from the environment over GPIO and similarly, can transmit outputs to the ARM or to the GPIO pins. Using the GPIO interface for both inputs and outputs effectively removes the ARM processor from the evaluation loop and therefore allows the FPGA to run independently for maximum performance. In this configuration, the ARM is only required to begin execution on the FPGA, but may have varied level of involvement depending on the application. If I/O is connected via the ARM each step, performance becomes limited by the AXI DMA transfer between the ARM and FPGA. Even small networks cannot improve beyond a step evaluation time of $715\,\mu$s compared to a fully independent design using GPIO that can operate over $1000\times$ faster at a sub-microsecond step time of $0.678\,\mu$s (for $N = 64$).

In this configuration we run Nengo directly on the PYNQ device creating a self-contained implementation, requiring only a terminal connection to initialize and begin the execution

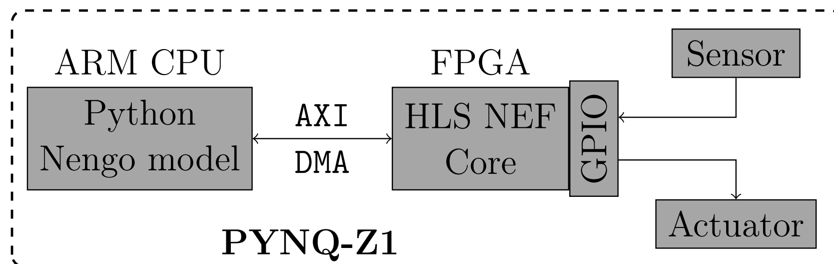


Figure 3.5: System level view of the NengoFPGA design using the ARM host. Nengo models are defined by Python code running on the ARM CPU and interact with the FPGA accelerator over AXI. The FPGA fabric interfaces directly with the external inputs and outputs over GPIO.

of a neural network. This self-contained mode is useful for certain embedded or edge applications as well as debugging and profiling the implementation. For a cleaner, more familiar experience, users are able to drive the FPGA implementation from their PC. Moreover, there are some additional consideration when running Nengo models directly on the ARM CPU which will be discussed in Section 3.2.3.

3.2.2 Remote PC Superhost

As noted in the Statement of Contributions, this extended superhost software interface was largely implemented by Xuan Choo.

In order to provide a more comfortable user experience, the NengoFPGA backend was extended to be run from a PC on the same network as the PYNQ device. Figure 3.6 shows the remote PC superhost interacting with the ARM on the PYNQ device. In this configuration, the Nengo model is run on the PC and the NengoFPGA backend communicates with the ARM host which in turn interfaces with the FPGA. Now that Nengo is run on the PC, we no longer require Nengo on the ARM processor and the ARM host interface is replaced by a simple script that makes use of local network connections to communicate with the PC superhost and uses the PYNQ API to communicate with the FPGA. On the superhost, NengoFPGA automatically opens an SSH connection with the PYNQ-Z1 and uses this connection to remotely send commands to begin execution. SSH is similarly leveraged to transfer model parameters to the ARM and subsequently to the FPGA. Once the device side has been initialized via SSH, the connection remains open and logs debugging information as well as monitoring for any errors or warnings that may arise.

In a separate thread, NengoFPGA opens a UDP socket connected to the ARM and it is this UDP socket that is used to transmit and receive data during the execution.

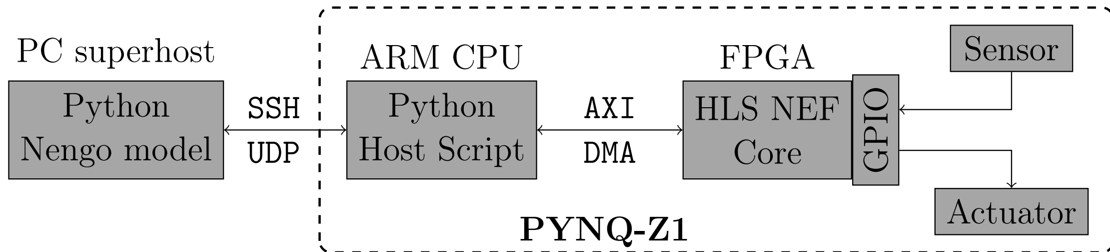


Figure 3.6: System level view of the NengoFPGA design using a PC superhost. Nengo models are defined by Python code running on a PC on the same network. The ARM CPU acts as an intermediary interacting with the FPGA accelerator over AXI and the PC over SSH and UDP socket. The FPGA fabric interfaces directly with the external inputs and outputs over GPIO.

Although the remote superhost is convenient, there is unavoidably a slight performance drop due to the extension in the execution loop. It is for this reason the lightweight UDP protocol was chosen, to minimize the performance drop. When the ARM processor is in the loop, the loop time sits around 715 μs, as noted in Section 3.2.1. When the superhost is brought into the loop, we get a loop time around 947 μs depending on the model size and the complexity of any additional Nengo code running on the PC. Although we see $\approx 20\%$ reduction in performance, we still meet our real-time goal of 1 ms step time in our Nengo simulation.

Another benefit of using the remote superhost is the multi-device capability. The interface is written in such a way that multiple PYNQ devices on the same local network can be easily integrated into a single Nengo network marshalled by the superhost. This is visualized in Figure 3.7 and allows multiple portions of the Nengo model to be accelerated in parallel with independent FPGA evaluations.

3.2.3 Interface Considerations

Beyond the aforementioned convenience–performance trade-off associated with using the superhost configuration, there are some other things to be aware of as well.

When initializing a Nengo network, the Nengo build system will analytically solve the decoder matrix given the randomly generated encoders and any desired function as

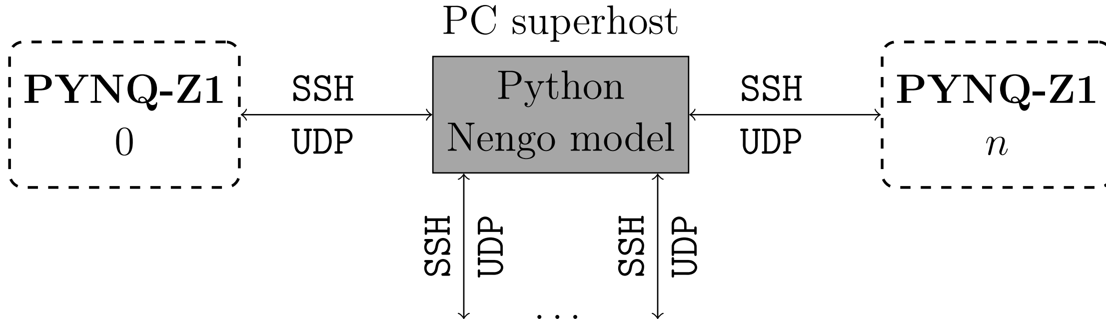


Figure 3.7: System level view of the NengoFPGA design using a PC superhost connected to multiple FPGA devices on the same network.

explained in Section 2.1. As the size of the weight matrices increases (*i.e.* N, D_{in}, and D_{out} increase), the complexity of solving the decoders grows super-linearly, as we are effectively doing a matrix inversion, and thus larger models increasingly tax available resources. In addition to having lower computational throughput, the embedded ARM processor also has limited memory available for computation (512MB in the case of the PYNQ-Z1) so solving for large decoder matrices on the ARM processor will, at the very least, be slower than solving the same on a full scale CPU and may, in fact, fail entirely if we run out of memory. For this reason, care should be taken when running large models directly from the ARM. On the other hand, if the decoders are manually set instead of calculated, then there is no issue as the remaining initialization steps are far less taxing computationally.

Currently the FPGA implementation supports a single ensemble of adaptive neurons and so it is likely there are some remaining portions of the Nengo model that are being run on the CPU, whether it be the ARM host or a PC superhost. Since the ARM has reduced performance compared to full scale CPUs, increasingly complex Nengo models run on the ARM may begin to impact performance and thus for large complex models it is recommended the superhost configuration be used such that the CPU component does not stall the evaluation. Furthermore, using the superhost configuration allows multiple FPGA accelerators to be integrated into a single model distributing the work further.

3.3 Using NengoFPGA

This section will introduce the basic Nengo integration, but full documentation for the NengoFPGA Python package[4] and the source code for the frontend interface[5] is available online. This section is heavily based on the usage documentation for NengoFPGA.

3.3.1 Converting from Nengo to NengoFPGA

Networks and models are described using the traditional Nengo workflow and a single ensemble, including PES learning, can be replaced with an FPGA ensemble using the `FpgaPesEnsembleNetwork` class. For example, consider the following example of a learned communication channel built with standard Nengo:

```
import nengo
import numpy as np

def input_func(t):
    return [np.sin(t * 2*np.pi), np.cos(t * 2*np.pi)]

with nengo.Network() as model:

    # Input stimulus
    input_node = nengo.Node(input_func)

    # "Pre" ensemble of neurons, and connection from the input
    pre = nengo.Ensemble(50, 2)
    nengo.Connection(input_node, pre)

    # "Post" ensemble of neurons, and connection from "Pre"
    post = nengo.Ensemble(50, 2)
    conn = nengo.Connection(pre, post)

    # Create an ensemble for the error signal
    # Error = actual - target = "post" - input
    error = nengo.Ensemble(50, 2)
    nengo.Connection(post, error)
```

[4] https://www.nengo.ai/nengo-fpga/
[5] https://github.com/nengo/nengo-fpga

```
    nengo.Connection(input_node, error, transform=-1)

    # Add the learning rule on the pre-post connection
    conn.learning_rule_type = nengo.PES(learning_rate=1e-4)

    # Connect the error into the learning rule
    nengo.Connection(error, conn.learning_rule)
```

The Nengo code above creates two neural ensembles, `pre` and `post`, and forms a PES-learning connection between these two ensembles. The weights of this connection are modulated by an error signal computed by a third neural ensemble (`error`). NengoFPGA can be used to replace the `pre` ensemble with an ensemble that will run on the FPGA. Converting the Nengo model above into a NengoFPGA model proceeds in three steps:

1. Replacing the desired neural ensemble with and FPGA ensemble.
2. Making the appropriate connections to and from the FPGA ensemble.
3. If desired (*i.e.* if learning is required), making the connections to and from an error-computing neural ensemble.

To use the FPGA ensemble, first import the `FpgaPesEnsembleNetwork` class:

```
from nengo_fpga.networks import FpgaPesEnsembleNetwork
```

In the original Nengo code above, the `pre` ensemble is to be replaced by the FPGA ensemble. The standard Nengo code is replaced with the `FpgaPesEnsembleNetwork` class. Since learning is desired in the above model, the learning rule definition on the pre–post connection (`conn.learning_rule_type =nengo.PES(learning_rate=1e-4)`) has been removed and rolled into the `FpgaPesEnsembleNetwork` constructor.

```
# "Pre" ensemble & learning rule
ens_fpga = FpgaPesEnsembleNetwork('pynq', n_neurons=50,
                                  dimensions=2,
                                  learning_rate=1e-4)
```

Notice that the `ens_fpga` ensemble maintains the same arguments as the original `pre` ensemble and the learning rule which it encompasses – 50 neurons, 2 dimensions, and a

learning rate of 1e-4. The `ens_fpga` has an additional argument, in this case `'pynq'`, which specifies the desired FPGA device[6]. With the FPGA ensemble created, the connections to and from the original `pre` ensemble will have to be updated with the slightly modified FPGA versions:

```
# Connection from input to "pre" (FPGA) ensemble
nengo.Connection(input_node, ens_fpga.input)  # Note the added '.
                                               input'

# Connection from "pre" (FPGA) to "post" ensemble
nengo.Connection(ens_fpga.output, post)  # Note the added '.output
                                           '
```

The NengoFPGA connections are very similar to the original Nengo connections with the exception that they use the interfaces of the `FpgaPesEnsembleNetwork` object. The `ens_fpga.input` and `ens_fpga.output` replace the input and output of the original `pre` ensemble. In the original Nengo model, a neural ensemble was used to compute the error signal that drives the PES learning rule. Using NengoFPGA, this neural ensemble is still needed, and the only change required is to modify the connections from this error ensemble to the FPGA ensemble:

```
# Create an ensemble for the error signal
# Error = actual - target = "post" - input
error = nengo.Ensemble(50, 2)  # Remains unchanged
nengo.Connection(post, error)  # Remains unchanged
nengo.Connection(input_node, error, transform=-1)  # Remains
                                               unchanged

# Connect the error into the learning rule
nengo.Connection(error, ens_fpga.error)  # Note the added '.error'
```

Note that in the NengoFPGA code, the `learning_rule_type` definition of the pre–post connection has been removed as this is declared as part of the `FpgaPesEnsembleNetwork` object. Altogether the NengoFPGA version of the learned communication channel would look something like this:

[6] This is an arbitrary designation defined by the user to relate a set of device parameters (*e.g.* IP address) to a physical device available to the NengoFPGA system

```
import nengo
import numpy as np

from nengo_fpga.networks import FpgaPesEnsembleNetwork

def input_func(t):
   return [np.sin(t * 2*np.pi), np.cos(t * 2*np.pi)]

with nengo.Network() as model:

    # Input stimulus
    input_node = nengo.Node(input_func)

    # "Pre" ensemble of neurons, and connection from the input
    ens_fpga = FpgaPesEnsembleNetwork('pynq', n_neurons=50,
                                      dimensions=2,
                                      learning_rate=1e-4)
    nengo.Connection(input_node, ens_fpga.input)  # Note the added
                                                  '.input'

    # "Post" ensemble of neurons, and connection from "Pre"
    post = nengo.Ensemble(50, 2)
    conn = nengo.Connection(ens_fpga.output, post)  # Note the
                                                   added '.output'

    # Create an ensemble for the error signal
    # Error = actual - target = "post" - input
    error = nengo.Ensemble(50, 2)
    nengo.Connection(post, error)
    nengo.Connection(input_node, error, transform=-1)

    # Connect the error into the learning rule
    nengo.Connection(error, ens_fpga.error)  # Note the added '.
                                                   error'
```

3.3.2 Basic Simulation

When run via the superhost, NengoFPGA is designed to work with the integrated Nengo GUI. Choose a Python file to run that uses the `FpgaPesEnsembleNetwork` and simply indicate

to the GUI that we desire the NengoFPGA backend: `nengo <my_file> -b nengo_fpga`. If you do not wish to use the GUI, or are running directly from the ARM host, then we can use NengoFPGA in scripting mode. Simply replace the standard Nengo simulator with the NengoFPGA version:

```
import nengo
import nengo_fpga

with nengo.Network() as model:

  # Your network description...
  # Including an FpgaPesEnsembleNetwork

with nengo_fpga.Simulator(model) as sim:
  sim.run(1)
```

Chapter 4

Results & Discussion

In this section, we evaluate the performance, resource usage, and error behaviour of our NengoFPGA implementation. Section 4.1 highlights the evolution of the HLS code description then Section 4.2 takes a closer look at the hyper-parameter optimization process. Finally, Section 4.3 compares performance to the Jetson TX1 GPU and Section 4.4 explains the end-to-end system characteristics.

4.1 Impact of HLS Code Description

High-Level Synthesis tools like Vivado HLS offer a convenient way to express your design and communicate optimization intent. Through various parallelism-centric reformulations, described in Section 3.1.2, we took a sample design at $D_{in} = 2$, $D_{out} = 2$, and $N = 200$ and sped it up over 15× while only using 2× more resources moving from the initial naive approach ("Baseline") to the fully parallel design in floating-point ("Dataflow"). As shown in Figure 4.1, the bulk of the speed up comes from successfully parallelizing the computation across the N dimension and making use of dataflow pipelining. The simple restructuring alone managed to improve performance nearly 20% while maintaining the same resource usage. This is testament to the state of the HLS tools wherein their infancy does not allow them to intelligently extract problem structure automatically. The use of fixed-point types permits an over 2× improvement in unroll factor, from $UFAC = 12$ to $UFAC = 28$, and

reduces cycle count further by over 4× when given the same resource budget (the entire chip). This is an overall improvement of over 50× in the compute pipeline.

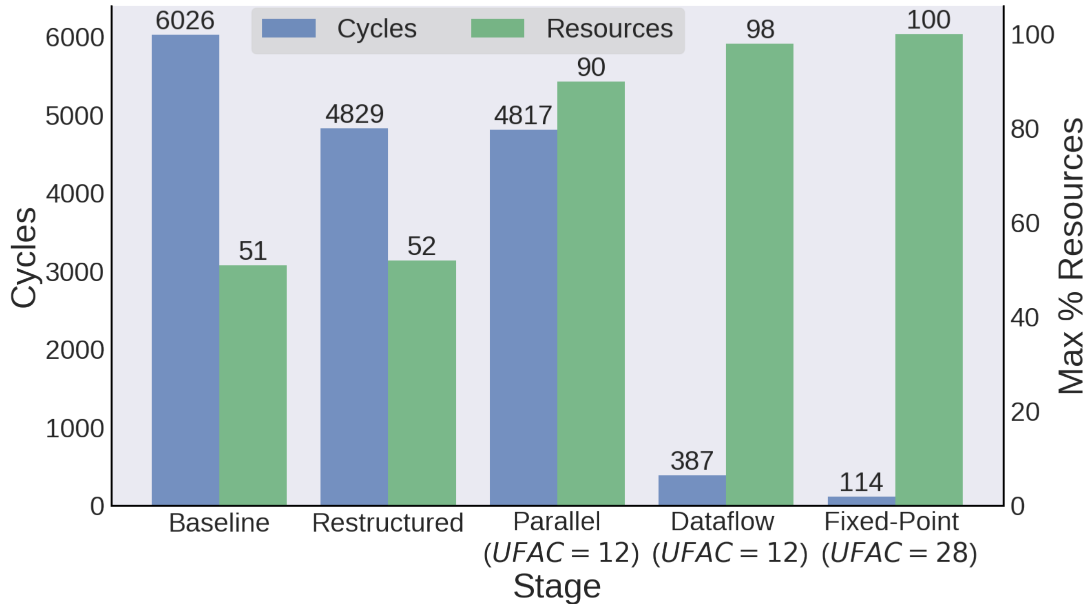


Figure 4.1: Performance improvements due to incremental, cumulative optimizations of HLS code. *Baseline* is the naive approach, *Restructured* has the decode loop inverted, *Dataflow* includes pipeline and dataflow pragmas, *Unroll* is unrolled with $UFAC = 12$, and *Fixed-Point* is the fixed-point design unrolled with $UFAC = 28$

4.2 Parameter Tuning with Hyperopt

As discussed earlier in Section 3.1.4, a brute-force exploration of the fixed-point precision design space is intractable. Hyperopt can help quickly search this design space if we first bound the possible range of parameter values. This section will explore the evaluation of the Pareto optimal designs discovered by Hyperopt as well as the convergence and performance of Hyperopt itself. More information about Hyperopt and the Pareto optimal designs can be found in Appendix A.

4.2.1 Initial Bounding

Two clear categories of fixed-point types emerged from the initial bounding experiments:

1. Those dependent on the number of integer bits.
2. Those dependent on the number of fractional bits.

Figures 4.2 & 4.3 show results from this isolated grid search for `DATA_T_ENC` and `DATA_T_ERR`; for clarity, the word bits axis is truncated to 38 bits instead of showing the full 64 bit range. Similar plots are generated for all candidate data types but not shown here. The error value displayed is the mean absolute error over the final 500 simulation timesteps. In Figure 4.2, we show the effect of varying the encoder type's (`DATA_T_ENC`) precision. The diagonal cut off indicates infeasible combinations where integer bits exceed total word bits. This is an example of a data type that depends on the number of integer bits, as illustrated by the horizontal gradation. We see that as we increase integer bits above 7 and, necessarily, word bits above 8, we are able to achieve very low error. For reference, floating-point error value for this design is $6.68e^{-3}$. Similarly, Figure 4.3 illustrates the effect of precision selection of the error signal data type (`DATA_T_ERR`). Here, the diagonal gradations show that most of the accuracy improvement is tied to having high precision representation of small quantities, *i.e.* at least 17–18 fractional bits.

To further explore the effects of varying precision on the overall system accuracy, we take a closer look at the time series error trend for the `DATA_T_ERR` type compared to the floating-point reference. Recall, a sinusoid is used as input to the simulation from which these error values were collected and the decoders are initialized as zeros (*i.e.* default output is zero for all stimuli). Figure 4.4 depicts the representational breakdown as we reduce the number of fractional bits in the data type. We start with a known good precision with 37 word bits and 4 integer bits (P10 in Table A6) and observe that this representation, being sufficient, exactly follows the floating-point reference. Next, we fix the number of integer bits at 4 and move horizontally across the accuracy transition diagonal observed in Figure 4.3. First we observe that the convergence *rate* is not effected by the varied precision as all curves settle after 2 input periods. However, the steady-state error displays notable differences as we reduce the number of available fractional bits. Second, we can observe that the quantization becomes apparent as the precision is reduced.

The curve with 23 word bits is closely follows the floating point and high-precision fixed-point trend and thus approaches the limit of useful bits for this applications. The

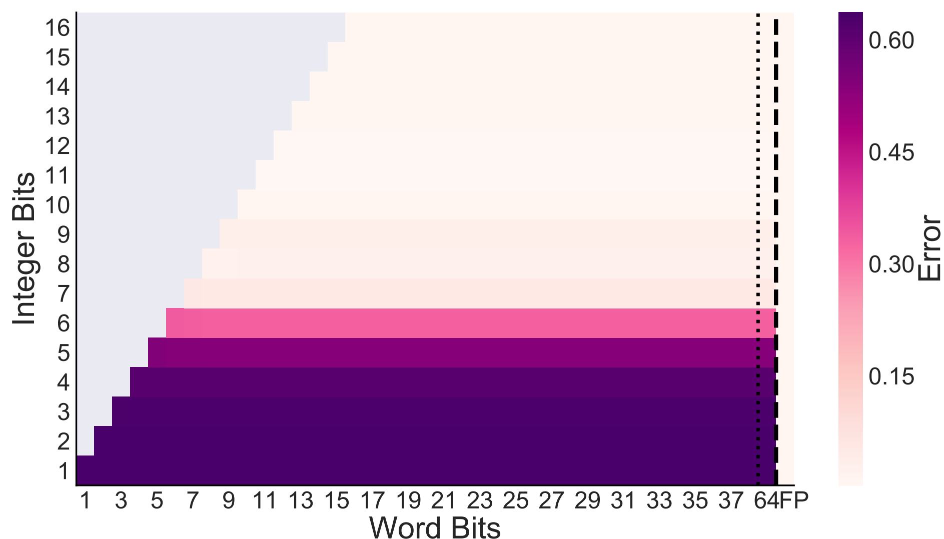


Figure 4.2: Grid search varying the precision of the `DATA_T_ENC` type as compared to the floating-point (FP) reference. Here we observe that the accuracy relies largely on the number of integer bits allocated to this data type.

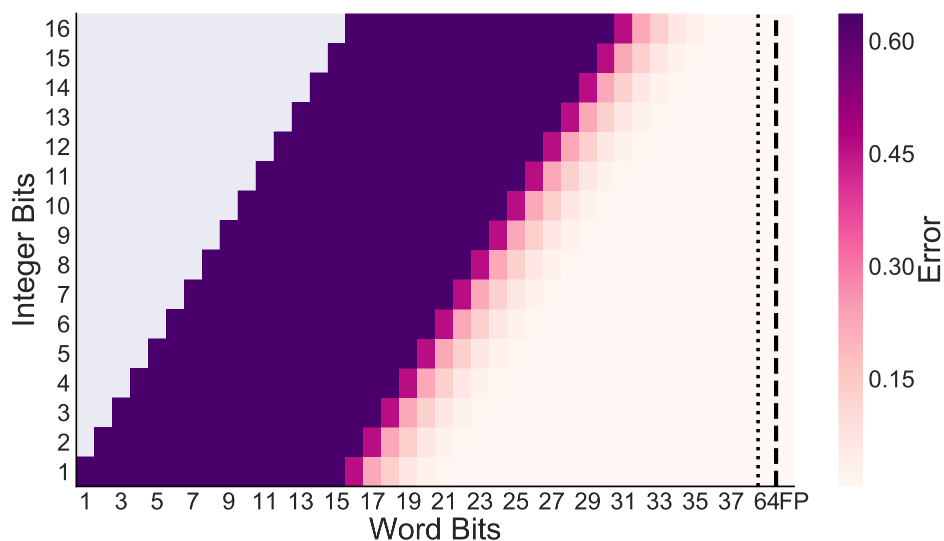


Figure 4.3: Grid search varying the precision of the `DATA_T_ERR` type as compared to the floating-point (FP) reference. Here we observe that the accuracy relies largely on the number of fractional bits allocated to this data type.

curves with 21 & 22 word bits happen to be coincident, which is an interesting artifact of this particular set of encoders and learning. these curves are almost able to represent the error to a useful degree: it begins by closely following the floating-point trend, but once the representation improves we notice that this precision is not able to represent the small error signal which drives the adaptation and therefore the system ceases to learn. The two curves with 19 and 20 word bits display similar issues, albeit with a worse steady-state gap as the precision is further reduced. These two curves also show some higher order ringing during the first two periods before settling at their respective steady-state errors. Finally, the curve with only 18 word bits is entirely incapable of adaptation at this precision as displayed by the full amplitude sinusoidal error.

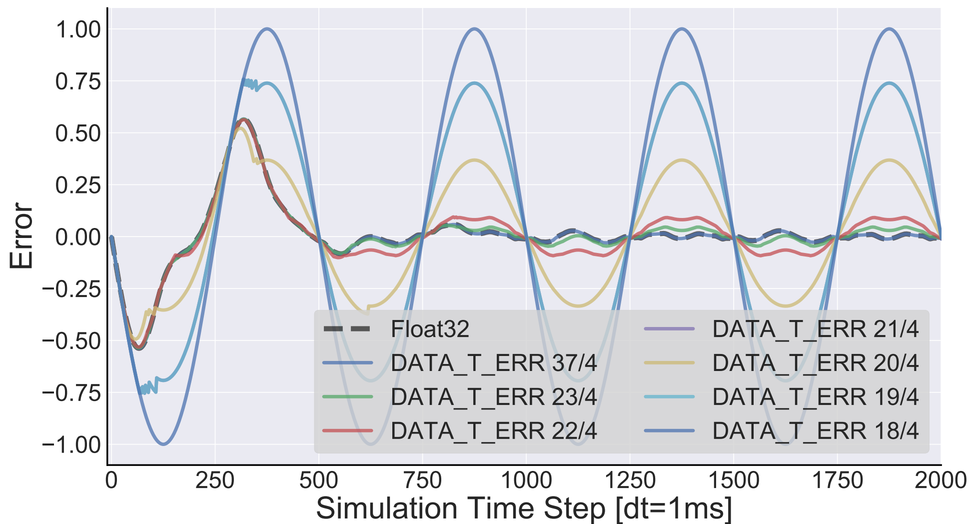


Figure 4.4: Overall error trends for `DATA_T_ERR` with 4 integer bits and varied fractional bits as the simulation learns a sinusoidal input. The curves are labelled as word_bits/integer_bits.

To further confirm that `DATA_T_ERR` is dependent on the number of fractional bits, we plot error trends for various precisions along the diagonal observed in Figure 4.3 (*i.e.* constant fractional bits and increased integer bits). Notice how all precisions plotted in Figure 4.5 adhere to the same trend therefore confirming that the additional integer bits are not useful for this particular data type.

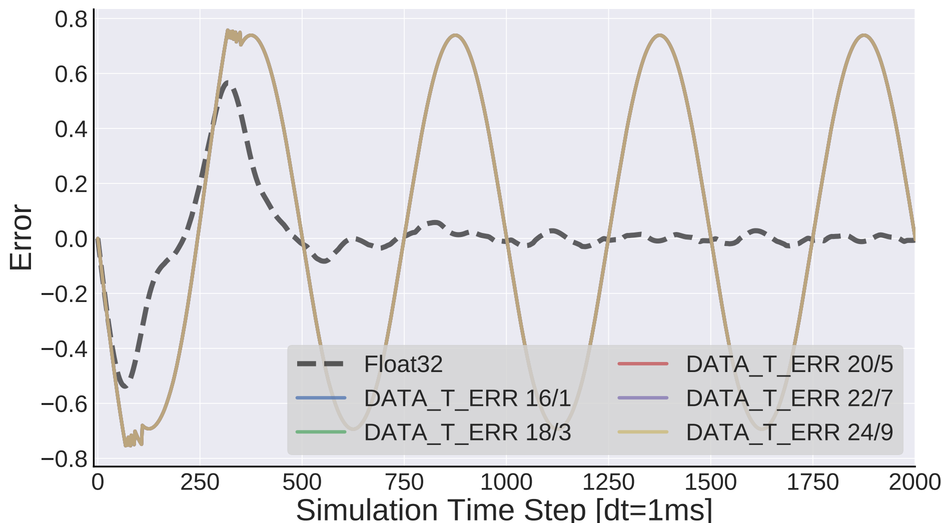


Figure 4.5: Overall error trends for `DATA_T_ERR` with 15 fractional bits and varied integer bits as the simulation learns a sinusoidal input. The curves are labelled as word_bits/integer_bits.

4.2.2 Sensitivity of Precision to Design Parameters

The error signal, $\boldsymbol{Err_x}$, is extremely sensitive to design parameters such as the number of neurons, N. This suggests that the Hyperopt tool will give different results for different neural network configurations. For example, Figure 4.6 shows the parameters optimized for one N cannot simply be transferred to another value of N. The distributed representation in neuron-space is such that as N increases, the average magnitude of the decoder decreases. This results in different optimal choices of precision for different N. Consequently, we define a general fixed-point design that uses the largest observed integer and fractional bits required over the design space, in this case from $N = 64$ to $N = 4096$. This design performs well over all values of N considered as seen in Figure 4.6.

In general, as the number of neurons increases we will require more fractional bits for the encoders and decoders. Since the neurons create a distributed representation, having more neurons leads to each neuron activity contributing less to the overall state-space representation and as such the encoders and decoders compensate by realizing smaller values. Consequently this also requires additional precision for the intermediate arithmetic as well.

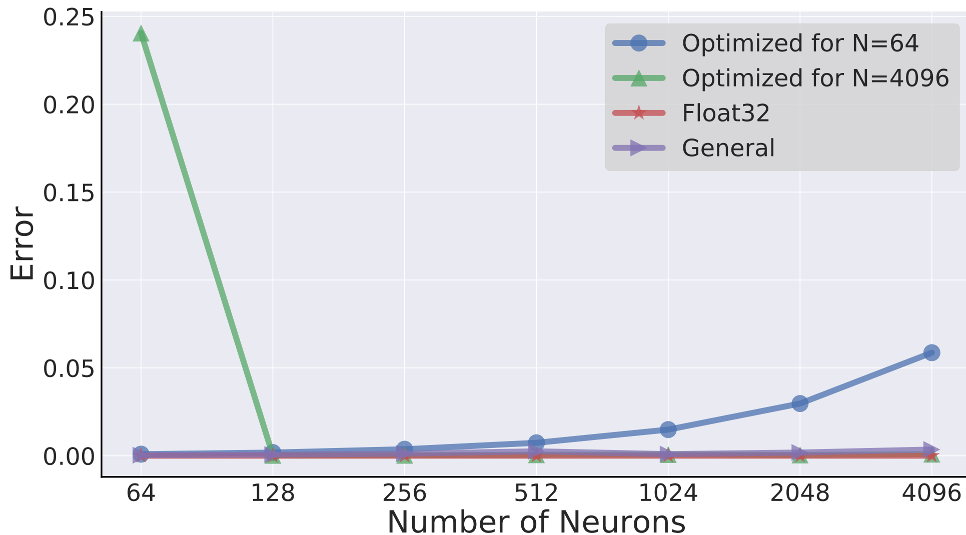


Figure 4.6: Representational error of designs optimized using different values of N in the Hyperopt simulation compared to a general fixed-point solution and a floating-point solution.

4.2.3 Hyperopt Convergence Rate

Hyperopt is fundamentally limited by the speed of the HLS tools as the C-synthesis is the critical path in the optimization loop. The frontend HLS passes must be run to produce resource and cycle count estimates. As shown in Figure 4.7, the best observed configuration (lowest cost) is reached after $\approx$ 1500 trials, which takes $\approx$ 12 hours to run on an Intel i7-6700K CPU. However, it takes less than $\approx$ 500 trials, or about 4 hours, to get within 99% of the best design. Thus, Hyperopt is a tractable approach for optimizing design parameters with Vivado HLS in the loop, requiring only a few hours of trials.

4.2.4 Hyperopt Design Optimization

Starting from the optimized parallel dataflow HLS implementation, and the bounded parameter ranges, we begin the Hyperopt optimizations. We choose the error–cycles product as the cost function, subject to a resource threshold. Designs above the resource threshold are not discarded, but rather aggressively penalized so Hyperopt will learn to stay below the threshold. The thresholds used were 100% for BRAMs and LUTs and 120%

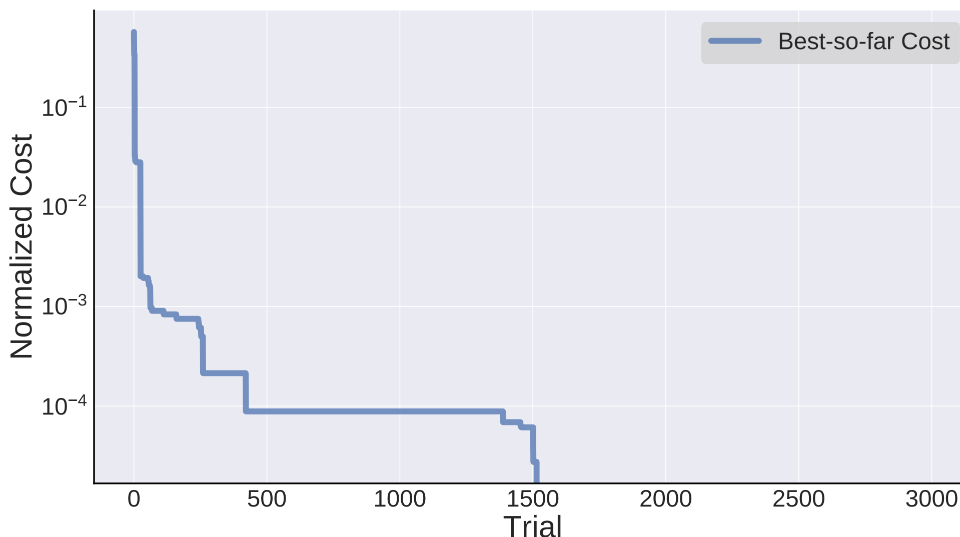


Figure 4.7: Normalized convergence of cost minimization for Hyperopt trials. This is a best-so-far trend and therefore will be monotonically decreasing.

for DSPs. The DSP threshold is higher since it is observed that Vivado can successfully compile designs in this range by making use of abundant LUTs to the handle additional fixed-point arithmetic requirement. This Hyperopt run produced 5 Pareto optimal designs (see Table A3), of which 1 is infeasible (too many resources) and 3 are slow but require few resources. The final design of the 5 (P4-C) is the fastest design discovered by Hyperopt across all runs.

The error–resource product cost function generated 11 Pareto optimal designs (see Table A2) 3–4× faster than the error–cycles minimization, and all of which are feasible and competitive in performance. The error–resources minimization is analyzed herein as it produces multiple good designs, better illustrating the possible solution space. Figure 4.8 shows the resulting error and resource cost combinations of each trial as a datapoint on the plot. As expected, smaller designs tend to have higher error than larger designs. However, a large number of design configurations are clearly dominated by those along the Pareto optimal curve, which are shown in Figure 4.9 in more detail. The final optimized designs for the various cost functions use precisions between 8–26 word bits for most data types with the exception of the error signal which uses between 37–48 word bits. These designs offer comparable accuracy against slower and larger floating-point designs and in some cases beat the floating-point solution in accuracy as some fixed-point types have more

precision than the 24 bit floating-point mantissa. In fact, there are a number of fixed-point designs highlighted in the lower left quadrant of Figure 4.8 that are strictly better than the floating-point reference.

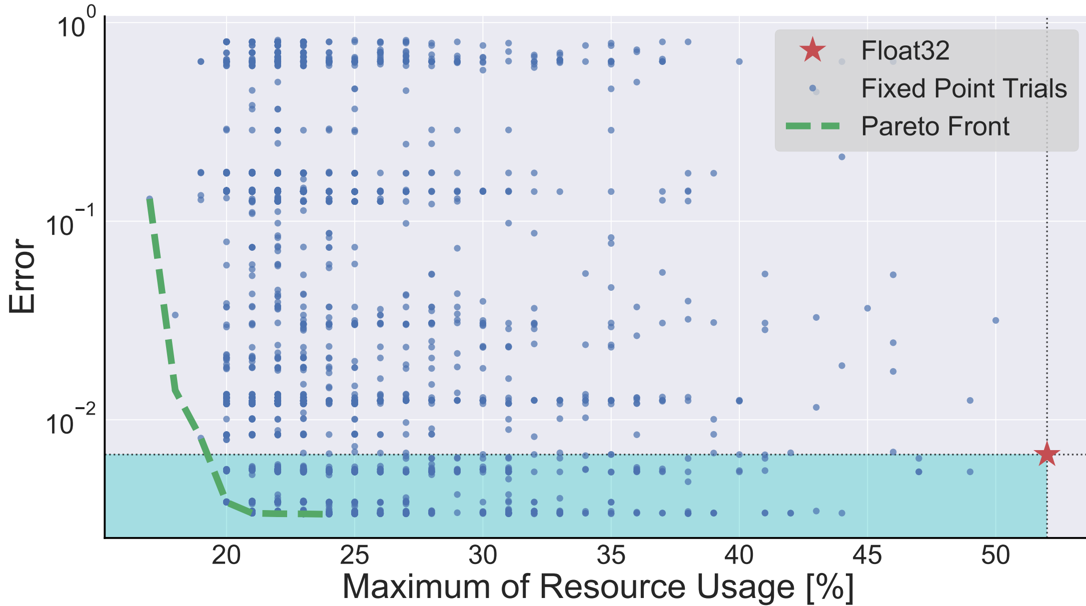


Figure 4.8: The Pareto optimal front resulting from Hyperopt trials minimizing the error–resource product. The shaded region in the lower left quadrant highlights designs that are strictly better than the floating-point baseline.

4.2.5 Pareto Optimal Designs

Referring to Figure 4.9, We observe that there is little difference in resource usage between designs of the error–resource optimization. These designs were discovered by minimizing resource usage with $UFAC = 1$ and are then manually unrolled to maximize replication and deliver high performance. All of these Pareto optimal designs are unrolled to a factor of 24 ($UFAC = 24$) and therefore have the same cycle count. The optimal design from the error–cycles minimization (labelled Cycles) is unrolled to a factor of 28 ($UFAC = 28$) resulting in slightly higher performance while the floating-point design is limited to an unroll factor of 12 ($UFAC = 12$). The general fixed-point solution using the largest integer and fraction bit configurations from the Pareto optimal designs can only be unrolled to a factor of 20 ($UFAC = 20$).

We now take a closer look at the results presented in Figure 4.9. The reduced BRAM usage in design P1 indicates reduced precision in the larger encoder and decoder memory structures that results in high error. The reduced DSP usage in design P2 indicates reduced precision throughout the arithmetic which also leads to reduced accuracy. Designs P3–P8 all have similar resource usage, however designs P3 and P4 appear to be using precision ineffectively as they boast larger error compared to designs P5–P8. Designs P9–P11 begin to show diminishing returns as they use far more resources without a significant improvement in accuracy. The P4-C cycles design has comparably high resource usage and similar error, but is justified in its resource usage with increased performance. The floating-point design also uses comparable resources but it has half the parallelism as the Pareto optimal designs and therefore much worse performance. The general fixed-point design is resource heavy and has an error slightly worse than the floating-point design, which illustrates the shortcomings of fixed-point arithmetic for flexible, general purpose applications having large dynamic ranges. Despite the slightly reduced accuracy, the general fixed-point design still vastly outperforms the floating-point design (as noted in Figure 4.10).

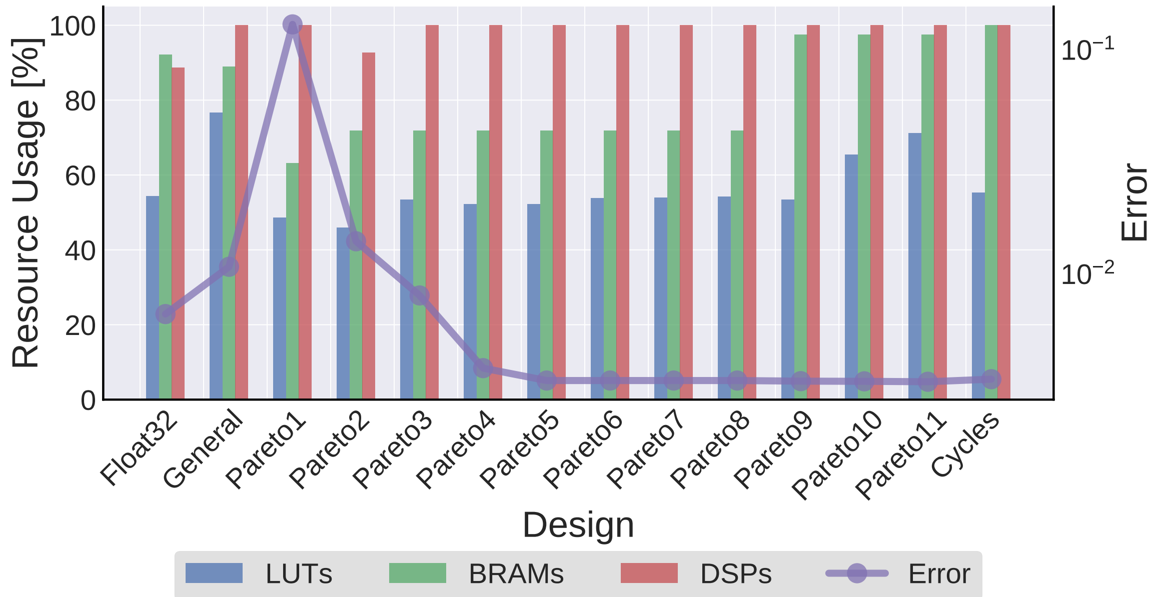


Figure 4.9: Resource usage and error trade-offs for the error–resource Pareto optimal designs ($UFAC = 24$) compared to the error–cycles optimal design ($UFAC = 28$), the general fixed-point ($UFAC = 20$), and the floating-point solution ($UFAC = 12$). $N = 200$, $D_{in} = D_{out} = 2$.

4.2.6 Floating-point vs. Fixed-Point

We now compare the performance of floating-point and fixed-point implementations on the FPGA after design optimization. Due to its large memory footprint, the floating-point design is limited to half the network size of the fixed-point design (*i.e.* $ND_{float} = 2 * ND_{fix}$). The complexity of the floating-point operations saturates the DSPs of the PYNQ board at an unroll factor of 12. In contrast, the fixed-point designs require fewer bits to store the network weights, and are able to effectively take advantage of both DSP and LUTs for arithmetic to support an unroll factor up to 28. The one dimensional trials in Figure 4.10 demonstrate that runtimes scale poorly for floating-point designs while the fixed-point solutions are faster by 4–5×. Furthermore, the general fixed-point design takes more resources (while staying within chip capacity), but it is only marginally slower than the fastest discovered design. Figure 4.11 shows that this scaling trend continues into higher dimensionality, in this case 8 dimensions.

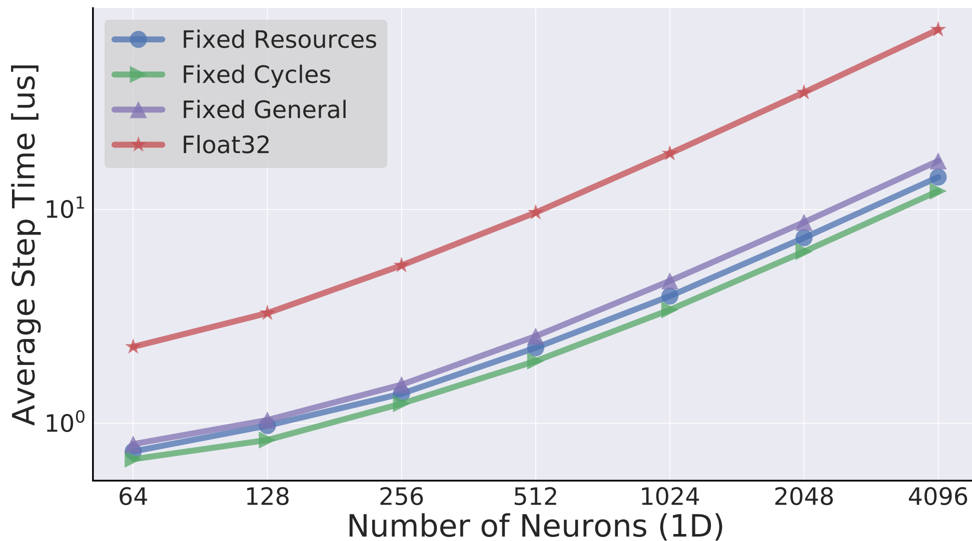


Figure 4.10: Comparing the scaling of execution times of a floating-point design, a general fixed-point design, and two differently optimized fixed-point designs with 1 representational dimension.

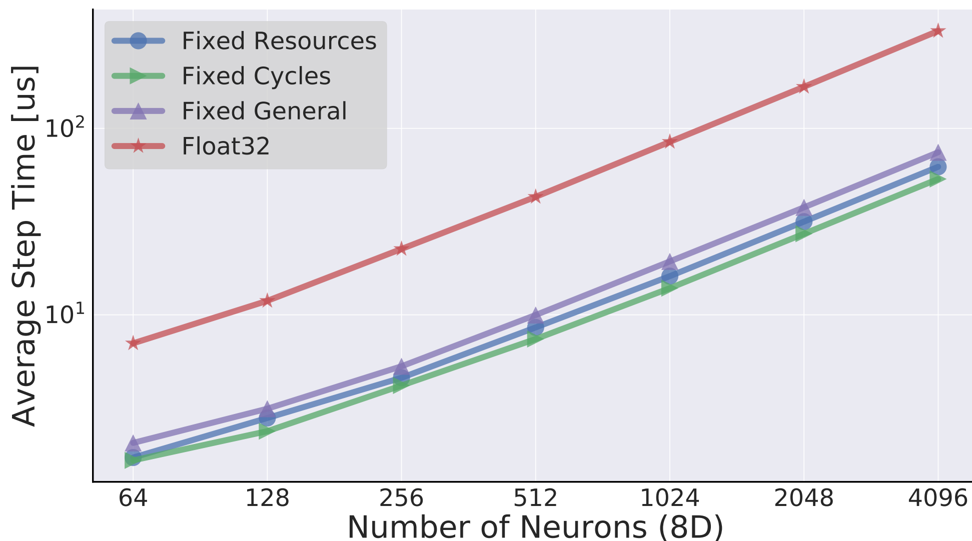


Figure 4.11: Comparing the scaling of execution times of a floating-point design, a general fixed-point design, and two differently optimized fixed-point designs with 8 representational dimension.

4.3 Comparison with Jetson TX1 GPU

The Jetson TX1 is not the latest embedded GPU, however, it uses 20 nm technology and the PYNQ-Z1 board uses an FPGA with 28 nm technology making the TX1 a reasonable comparison. We developed an optimized CUDA implementation of our NEF test network for the TX1 in order to compare performance with PYNQ. The NEF computations are composed of common matrix algebra primitives that map to highly optimized cuBLAS library calls. A small subset of the functions, including the neural model G and the feedback error calculation $\boldsymbol{Err_x} = |f(\boldsymbol{x}) - \boldsymbol{y}|$, are written in CUDA and optimized directly. Since Nengo operates on I/O signals that cannot directly interface to the GPU core, these signals must be copied over requiring frequent host-device transfers every timestep. Coupled with the relatively small size of the problem, the I/O heavy nature of the computation puts the GPU at a disadvantage compared to the FPGA.

The performance of the GPU compared to the FPGA is summarized in Figures 4.12 & 4.13. As expected, it shows that the FPGA is faster by 10–484×, and is simultaneously more power efficient using 2.4–9.5× less dynamic power than the GPU. As the problem size grows, the cost of I/O is amortized over the computation improving the efficiency of the GPU hardware. Thus, the FPGA speedups are highest ≈484× for the smaller neural networks,

but drop to ≈10× for the larger sizes. When scaling out to larger models accommodated by the fixed-point precision's small footprint, we see the FPGA continuing to beat the GPU with numbers in this range for models up to the tested 32 dimensions (1k neurons) and 32k neurons (1 dimension). We have $ND_{max} = 32\text{k}$, so models larger in both N and D are still infeasible at the highest performing unroll factor. The FPGA also consistently uses less dynamic power, using only 0.4–0.5 W of dynamic power while the GPU uses a wider range of 1.2–3.8 W. Both devices use ARM cores and therefore have similar static power draws of 2.6 W (for the GPU) and 2.5 W (for the FPGA).

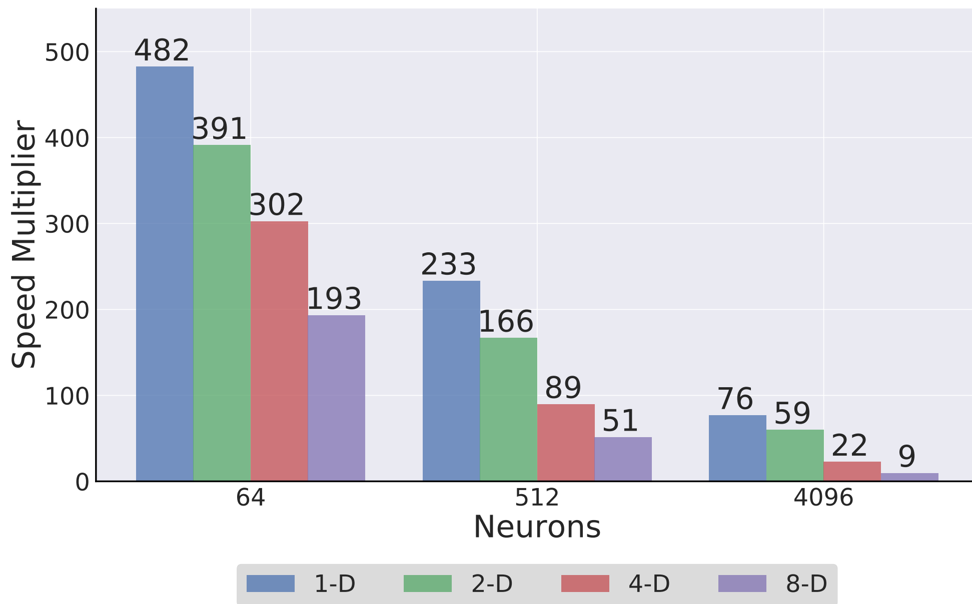


Figure 4.12: Jetson TX1 performance normalized against performance of the best P4-C FPGA design which is 10–484× faster depending on network size.

4.3.1 Practical Application of NengoFPGA

Although the size of networks that can be run with NengoFPGA is limited by on-chip memory, there are many useful applications well within our capacity. For example:

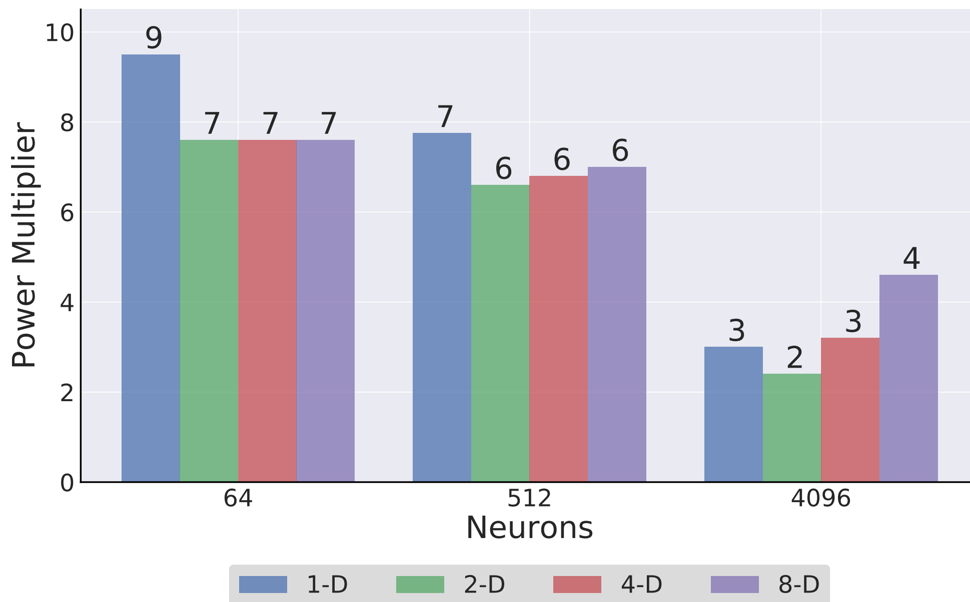


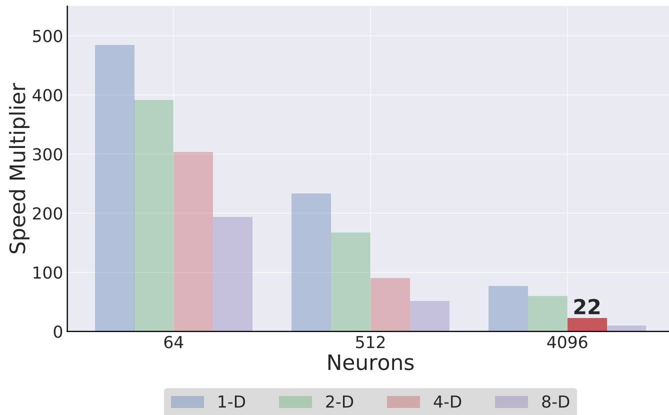
Figure 4.13: Jetson TX1 dynamic power use normalized against performance of the best P4-C FPGA design which is 2.4–9.5× more power efficient depending on network size.

Adaptive Motor Control

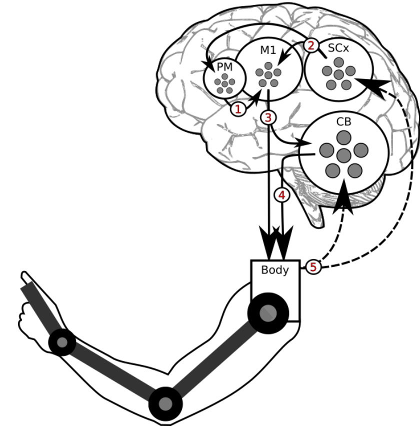
A population of 3000 neurons with 6 representational dimensions can form the basis of an efficient adaptive motor controller as shown by DeWolf et al. (2016). NengoFPGA can evaluate a model this size in under 31μs. Figure 4.14 highlights this performance against the Jetson GPU.

Many Body Control

As few as 500 neurons can adapt to a randomly generated 15-joint body simulation as shown by Stewart et al. (2015). NengoFPGA can evaluate 500 neurons with $D_{in} = 30$ (position and velocity for each of the 15 simple joints) in under 24μs. Figure 4.15 highlights this performance against the Jetson GPU.

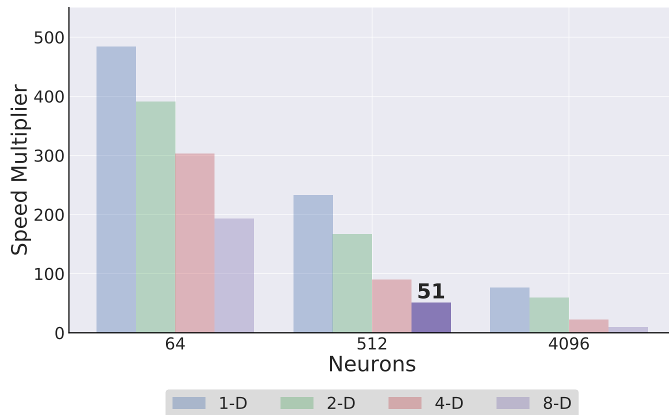


(a) FPGA performance for an adaptive controller

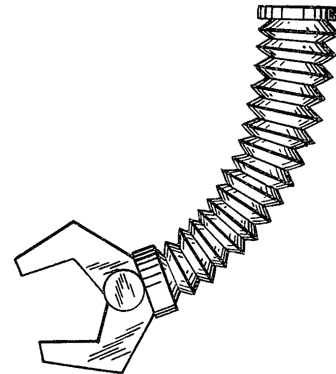


(b) Adaptive control graphic

Figure 4.14: Highlighting the speedup of the FPGA over the Jetson TX1 in a possible adaptive controller application. Adaptive control graphic from DeWolf et al. (2016)



(a) FPGA performance for a many body controller



(b) Many body graphic*

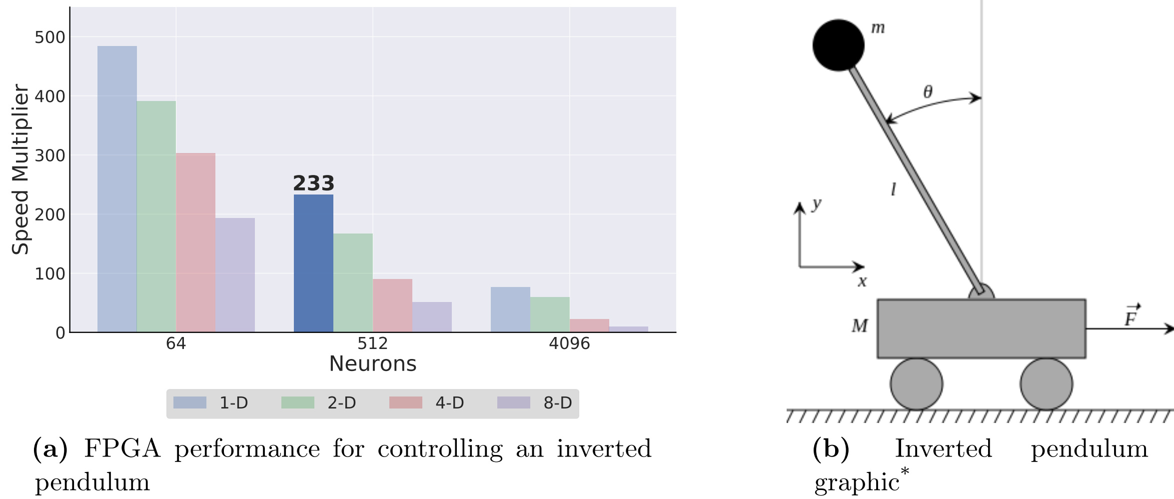
Figure 4.15: Highlighting the speedup of the FPGA over the Jetson TX1 in a possible many body control application.

***** Arm graphic from Google images (`https://www.kisspng.com/png-robotic-arm-clip-art-gears-art-629780/`)

Adaptive PID Controller

We also test NengoFPGA as an adaptive PID controller for an inverted pendulum in a simulated Python environment using 1000 neurons and 1 dimension. The fixed-point design

has an evaluation time under 4μs and has a competitive RMSE of $5.66e^{-3}$ compared to $5.45e^{-3}$ for the floating-point reference design. Figure 4.16 highlights this performance against the Jetson GPU.



(a) FPGA performance for controlling an inverted pendulum

(b) Inverted pendulum graphic*

Figure 4.16: Highlighting the speedup of the FPGA over the Jetson TX1 in a possible inverted pendulum control application.

* Pendulum graphic from Wikipedia (https://en.wikipedia.org/wiki/Inverted_pendulum#/media/File:Cart-pendulum.svg)

4.4 System Profiling

The standard Nengo timestep is 1 ms and therefore we must evaluate a full loop within this millisecond time budget in order to remain real-time. Figure 4.17 Show a diagram of the full system with timing data for each leg of the trip. First a test system is created that sends dummy UDP packets between the PC superhost and the ARM host. Three trials were done with this minimal test system and for each 10k UDP packets were transferred and the time per packet averaged over the run. This simple UDP transfer takes 379 μs according to this test. Note that this test was conducted using wired Ethernet connections across a local network via an off the shelf router (Asus RT-N56U) and this timing delay may vary greatly depending on the network configuration. Next the same dummy UDP packets

were transferred, but this time using a Nengo model to send and receive data. The UDP socket used for input and output communication between the PC superhost and the ARM host is built into a Nengo process and therefore the delay introduced by Nengo is negligible ($2\,\mu$s), unless the Nengo model running on the PC is large and computationally intensive, at which point this may become the system bottleneck. While exploring the speedup of direct I/O (*i.e.* GPIO) compared to using a DMA engine via the ARM host it was noted that the overhead for a DMA transfer is a constant $715\,\mu$s and similarly, we discover that for a network in which $N = 200$ and $D_{in} = D_{out} = 2$ the hardware computation alone takes a negligible amount of time ($\approx 1\,\mu$s).

The full end-to-end system test reported an elapsed time of $947\,\mu$s on average for our simple test network. Note that this elapsed time is not simply the sum of all parts as the components are pipelined and some delays are hidden and dominated by certain components. For example, the hardware execution time is entirely hidden by the overhead of the DMA transfer. and similarly a portion of the UDP transfer overlaps with the DMA transfer as data is pipelined between the host and superhost.

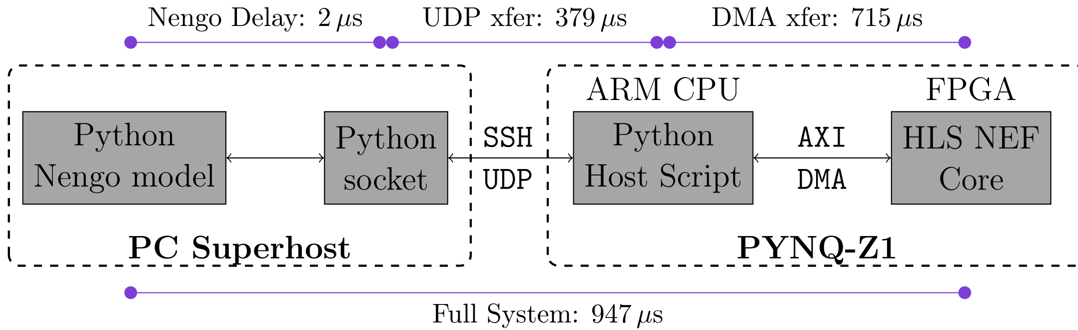


Figure 4.17: System level view of the full NengoFPGA implementation showing timing for each segment of the end-to-end system for the example network with $N = 200$ and $D_{in} = D_{out} = 2$.

Thus, the extension to the superhost configuration does add non-negligible time delay but still remains within our real-time budget of 1 ms. Similarly, if we consider that using the ARM host to furnish data in the direct host configuration can be no better than 715μs, the superhost configuration is only $\approx 24\%$ slower at $947\,\mu$s timing but allows for a full Nengo development environment with which to incorporate multiple FPGAs or subsystems.

Chapter 5

Conclusion & Future Works

5.1 Conclusion

The use of embedded Python-capable PYNQ FPGA boards offers a promising solution to the heavy workload computing required by the machine learning revolution. Using High-Level Synthesis in conjunction with the PYNQ Python API helps make FPGA programming more accessible. Moreover, NengoFPGA makes this flexible hardware accelerator accessible through the familiar Nengo ecosystem directly from a PC with no esoteric hardware knowledge required from the user.

We have shown how a structured HLS approach tailored to the given hardware can be used to exploit the parallelism of the problem reducing the cycles required to evaluate a neural network by 15×. Furthermore, we showed that the fixed-point hyper-parameters can be optimized automatically using Hyperopt for cost functions comprising of resource usage, accuracy, and cycle count to further reduce required evaluation cycles by an additional 3× — an overall improvement of over 50×. In addition, the reduced precision of the fixed-point representation allows larger models, with up to 32k neurons, to be stored on chip while still outperforming the floating-point counterpart, which can only support 16k neurons. Using direct I/O access improves performance 1000× compared to a design limited by the ARM DMA engine, from a step time of 715 μs to 0.678 μs. We also demonstrated that our FPGA implementation outperforms the Jetson TX1 GPU by 10–484× for neural network populations of 64–4096 neurons and 1–8 representational dimensions while using 2.4–9.5×

less power. In fact, the FPGA continues to beat the GPU with numbers in this range when scaling to models of 32 dimensions (1k neurons) and 32k neurons (1 dimension)[1].

5.2 Future Work

This project is the first step in creating a fully featured FPGA backend for the Nengo ecosystem. Beyond low-power, low-cost embedded scenarios, we plan to extend this work to larger FPGAs to address larger cloud and edge computing workloads as well. Some possible directions for future development include:

- scaling to larger FPGAs;
- implementing more Nengo classes such as neuron models and synapses;
- adding dedicated hardware accelerators; and
- supporting convolutions.

5.2.1 Larger FPGAs

The current design on the small PYNQ-Z1 device implements a single adaptive population of neurons. In order to create a fully featured and independent Nengo backend we will need to scale this up to include multiple neural ensembles as well as other objects. Some preliminary work has been done which tiles the current single population design multiple times and adds a simple ring communication network between the ensemble processors and the I/O interfaces. This has been functionally implemented at reduced scale on the PYNQ-Z1, but much work is needed to bring the design into an *accurate* state and scale up to larger Xilinx devices.

In another venture, this time using an Intel Aria10 device, a proof of concept design has been developed including an ensemble processor, as well as a generic matrix multiply block and a scalar multiply block. This preliminary design is able to accept an entire Nengo model which is then partitioned into these 3 compute categories. The system then constructs the

[1] Recall, ND_{max} = 32k and so models cannot be larger in both N and D

dataflow graph and evaluates the Nengo model across these 3 processor types. Currently, the implementation is quite inefficient having one of each processor type, each of which is naively implemented and requires optimization.

There is much work to be done, but this work and these two forays into scaled hardware are promising starts.

5.2.2 More Functionality

The current implementation only supports Rectified Linear Units (ReLU) as the neuron model and the Prescribed Error Sensitivity (PES) learning rule. However there are a plethora different neuron models and learning rules of varying complexity and detail that would be interesting to accelerate in hardware. In addition, there are many operations beyond the ensemble processor presented in this thesis. There is ample opportunity to implement various support blocks, such as direct matrix-matrix multiplies or scalar multiplies, and other neural blocks such as different synaptic filters. As the design is scaled to larger devices, adding more specialized blocks such as these should help improve performance and avoid adding complexity to generic processors. Similarly, having on-chip communication between blocks opens up possible computation during communication and interesting fanin and fanout considerations.

Another exciting opportunity would be to create a generic, run-time programmable GPIO interface. Currently any GPIO interaction require a custom compiled design to appropriately handle specific I/O.

5.2.3 Hardware Accelerators

The FPGA implementation is in and of itself a hardware accelerator; however there are opportunities to optimize specific functions even further. Currently the design is implemented using High-Level Synthesis (HLS) which is great for flexibility and design effort, but does sacrifice *some* performance. There could be dedicated, hand optimized RTL blocks dropped into the critical path (*e.g.* matrix multipliers or neuron models).

There is also work underway to embed a hardware Random Number Generator (RNG) based of the work of Thomas and Luk (2013). Since NEF encoders can be randomly

generated, the idea is to use a repeatable RNG to generate the encoders on-the-fly each cycle and save the large $N \times D$ memory footprint. In such an implementation, a random sequence of encoders is repeatably generated for each timestep which places the onus on the decoders to produce a sensible output value. The decoders could be initialized randomly as well and we could rely on the online learning to converge to a solution. Conversely, a software simulated RNG could be implemented as part of the build process in order to generate would be encoders and analytically solve decoders from there.

5.2.4 Convolutions

Finally, it would be great to also support Convolutional Neural Networks (CNN) in order to deliver a more well rounded platform. The NEF implementation is exciting and well suited to dynamic, real-time applications, but there is significant research and industrial interest in around CNNs.

References

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

Abdelfattah, M. S., D. Han, A. Bitar, R. Dicecco, S. O'Connell, N. Shanker, J. Chu, I. Prins, J. Fender, A. C. Ling, and G. R. Chiu
2018. DLA: Compiler and FPGA overlay for neural network Inference acceleration. In *Proceedings - 2018 International Conference on Field-Programmable Logic and Applications, FPL 2018*, Pp. 411–418.

Ando, K., K. Ueyoshi, Y. Oba, K. Hirose, R. Uematsu, T. Kudo, M. Ikebe, T. Asai, S. Takamaeda-Yamazaki, and M. Motomura
2018. Dither NN: An Accurate Neural Network with Dithering for Low Bit-Precision Hardware. In *International Conference on Field-Programmable Technology (FPT)*.

Aydonat, U., S. O'Connell, D. Capalija, A. C. Ling, and G. R. Chiu
2017. An OpenCL(TM) Deep Learning Accelerator on Arria 10.

Bachrach, J., H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović
2012. Chisel: Constructing Hardware in a Scala Embedded Language. In *Proceedings of the 49th Annual Design Automation Conference on - DAC '12*, P. 1216, New York, New York, USA. ACM Press.

Bekolay, T., J. Bergstra, E. Hunsberger, T. DeWolf, T. C. Stewart, D. Rasmussen, X. Choo, A. R. Voelker, and C. Eliasmith
2014. Nengo: A python tool for building large-scale functional brain models. *Frontiers in Neuroinformatics*, 7(48).

Bekolay, T., C. Kolbeck, and C. Eliasmith
2013. Simultaneous unsupervised and supervised learning of cognitive functions in biologically plausible spiking neural networks. In *35th Annual Conference of the Cognitive Science Society*, Pp. 169–174. Cognitive Science Society.

Bergstra, J., R. Bardenet, Y. Bengio, and B. Kégl
2011. Algorithms for hyper-parameter optimization. In *NIPS*.

Bergstra, J., B. Komer, C. Eliasmith, D. Yamins, and D. D. Cox
2015. Hyperopt: a python library for model selection and hyperparameter optimization. *Computational Science & Discovery*, 8(1):014008.

Berzish, M., C. Eliasmith, and B. Tripp
2016. Real-Time FPGA Simulation of Surrogate Models of Large Spiking Networks. In *International Conference on Artificial Neural Networks*, Pp. 349–356. Springer, Cham.

Blouw, P., X. Choo, E. Hunsberger, and C. Eliasmith
2018. Benchmarking keyword spotting efficiency on neuromorphic hardware. *CoRR*, abs/1812.01739.

Bureau of Labor Statistics, U. S.
2016. Occupational outlook handbook. https://www.bls.gov/ooh/.

Canis, A., J. Choi, M. Aldham, V. Zhang, A. Kammoona, J. Anderson, S. Brown, and T. Czajkowski
2011. LegUp: High-level synthesis for FPGA-based processor/accelerator systems.

Carlos Moctezuma, J., J. P. Mcgeehan, and J. Luis Nunez-Yanez
2015. Biologically compatible neural networks with reconfigurable hardware. *Microprocessors and Microsystems*, 39:693–703.

Cass, S.
2018. The 2018 top programming languages. Technical report.

Cass, S.
2019. Taking AI to the edge: Google's TPU now comes in a maker-friendly package. *IEEE Spectrum*, 56(5):16–17.

Cerda, J. C., C. D. Martinez, J. M. Comer, and D. H. Hoe
2012. An efficient FPGA random number generator using LFSRs and cellular automata. In *Midwest Symposium on Circuits and Systems*, Pp. 912–915. IEEE.

Chollet, F. et al.
2015. Keras. https://keras.io.

Choo, X.
2018. *Spaun 2.0: Extending the World's Largest Functional Brain Model*. Phd thesis, University of Waterloo.

Choudhary, S., S. Sloan, S. Fok, A. Neckar, E. Trautmann, P. Gao, T. Stewart, C. Eliasmith, and K. Boahen
2012. Silicon neurons that compute. In *International Conference on Artificial Neural Networks*, volume 7552 LNCS, Pp. 121–128.

Cong, J., B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang
2011. High-level synthesis for FPGAs: From prototyping to deployment. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, volume 30, Pp. 473–491.

Corradi, F., C. Eliasmith, and G. Indiveri
2014. Mapping arbitrary mathematical functions and dynamical systems to neuromorphic VLSI circuits for spike-based neural computation. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, Pp. 269–272. IEEE.

Corradi, G.
2018. The Value of Python Productivity : Extreme Edge Analytics on Xilinx Zynq Portfolio. Technical report.

Czajkowski, T. S., U. Aydonat, D. Denisenko, J. Freeman, M. Kinsner, D. Neto, J. Wong, P. Yiannacouras, and D. P. Singh
2012. From opencl to high-performance hardware on FPGAS. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Pp. 531–534. IEEE.

Davies, M., N. Srinivasa, T.-H. Lin, G. Chinya, Y. Cao, S. H. Choday, G. Dimou, P. Joshi, N. Imam, S. Jain, Y. Liao, C.-K. Lin, A. Lines, R. Liu, D. Mathaikutty, S. McCoy, A. Paul, J. Tse, G. Venkataramanan, Y.-H. Weng, A. Wild, Y. Yang, and H. Wang
2018. Loihi: A Neuromorphic Manycore Processor with On-Chip Learning. *IEEE Micro*, 38(1):82–99.

Dennard, R. H., F. H. Gaensslen, Y. U. Hwa-Nien, V. Leo Rideout, E. Bassous, and A. R. Leblanc
1999. Design of Ion-Implanted MOSFETs with Very Small Physical Dimensions. *Proceedings of the IEEE*, 87(4):668–678.

DeWolf, T., T. C. Stewart, J.-J. Slotine, and C. Eliasmith
2016. A spiking neural model of adaptive arm control. volume 283.

Diamantopoulos, D. and C. Hagleitner
2018. A System-level Transprecision FPGA Accelerator for BLSTM Using On-chip Memory Reshaping. In *International Conference on Field-Programmable Technology (FPT)*.

Eliasmith, C.
2013. *How to build a brain: A neural architecture for biological cognition*. New York, NY: Oxford University Press.

Eliasmith, C. and C. H. Anderson
2003. *Neural engineering: Computation, representation, and dynamics in neurobiological systems*. Cambridge, MA: MIT Press.

Eliasmith, C., T. C. Stewart, X. Choo, T. Bekolay, T. DeWolf, Y. Tang, and D. Rasmussen
2012. A large-scale model of the functioning brain. *Science*, 338:1202–1205.

Feist, T.
2012. Vivado design suite. *White Paper*, 5:30.

Fischl, K. D., A. G. Andreou, T. C. Stewart, and K. Fair
2018. Implementation of the Neural Engineering Framework on the TrueNorth Neurosynaptic System. In *2018 IEEE Biomedical Circuits and Systems Conference (BioCAS)*, Pp. 1–4. IEEE.

Frenkel, C., M. Lefebvre, J. D. Legat, and D. Bol
2019. A 0.086-mm2 12.7-pJ/SOP 64k-Synapse 256-Neuron Online-Learning Digital Spiking Neuromorphic Processor in 28-nm CMOS. *IEEE Transactions on Biomedical Circuits and Systems*, 13:145–158.

Furber, S. B., F. Galluppi, S. Temple, and L. A. Plana
2014. The SpiNNaker Project. *Proceedings of the IEEE*, 102(5):652–665.

GitHub:nengo-loihi
2019. nengo/nengo-loihi: Run Nengo models on Intel's Loihi chip. https://github.com/nengo/nengo-loihi.

Greff, K., R. K. Srivastava, J. Koutník, B. R. Steunebrink, and J. Schmidhuber
2015. LSTM: A Search Space Odyssey. *Transactions On Neural Networks And Learning Systems*.

Grossman, J., B. Towles, B. Greskamp, and D. E. Shaw
2015. Filtering, Reductions and Synchronization in the Anton 2 Network. In *2015 IEEE International Parallel and Distributed Processing Symposium*, Pp. 860–870. IEEE.

Hao, Y. and S. Quigley
2017. The implementation of a Deep Recurrent Neural Network Language Model on a Xilinx FPGA.

Hoppner, S., Y. Yan, B. Vogginger, A. Dixius, J. Partzsch, F. Neumarker, S. Hartmann, S. Schiefer, S. Scholze, G. Ellguth, L. Cederstroem, M. Eberlein, C. Mayr, S. Temple, L. Plana, J. Garside, S. Davison, D. R. Lester, and S. Furber
2017. Dynamic voltage and frequency scaling for neuromorphic many-core systems. In *Proceedings - IEEE International Symposium on Circuits and Systems*.

Huang, Q., R. Lian, A. Canis, J. Choi, R. Xi, S. Brown, and J. Anderson
2013. The effect of compiler optimizations on high-level synthesis for FPGAs. In *Proceedings - 21st Annual International IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM 2013*, Pp. 89–96.

Jacob, B., S. Kligys, B. Chen, M. Zhu, M. Tang, A. Howard, H. Adam, and D. Kalenichenko
2017a. Quantization and Training of Neural Networks for Efficient Integer-Arithmetic-Only Inference. Technical report.

Jacob, B., S. Kligys, B. Chen, M. Zhu, M. Tang, A. G. Howard, H. Adam, and D. Kalenichenko
2017b. Quantization and training of neural networks for efficient integer-arithmetic-only inference. *CoRR*, abs/1712.05877.

Jain, A. K., S. A. Fahmy, and D. L. Maskell
2015. Efficient Overlay Architecture Based on DSP Blocks. In *2015 IEEE 23rd Annual International Symposium on Field- Programmable Custom Computing Machines*, Pp. 25–28. IEEE.

Jouppi, N.
2016. Google supercharges machine learning tasks with tpu custom chip.

Khronos OpenCL Working Group et al.
. The opencl specification, version 1.2.

Klöckner, A., N. Pinto, Y. Lee, B. Catanzaro, P. Ivanov, and A. Fasih
2012. PyCUDA and PyOpenCL: A scripting-based approach to GPU run-time code generation. *Parallel Computing*, 38(3):157–174.

Kluyver, T., B. Ragan-Kelley, F. Pérez, B. Granger, M. Bussonnier, J. Frederic, K. Kelley, J. Hamrick, J. Grout, S. Corlay, P. Ivanov, D. Avila, S. Abdalla, and C. Willing
2016. Jupyter notebooks – a publishing format for reproducible computational workflows. In *Positioning and Power in Academic Publishing: Players, Agents and Agendas*, F. Loizides and B. Schmidt, eds., Pp. 87 – 90. IOS Press.

Koromilas, E., I. Stamelos, C. Kachris, and D. Soudris
2017. Spark acceleration on FPGAs: A use case on machine learning in Pynq. In *2017 6th International Conference on Modern Circuits and Systems Technologies (MOCAST)*, Pp. 1–4. IEEE.

Le, Q. V., N. Jaitly, and G. E. Hinton
2015. A Simple Way to Initialize Recurrent Networks of Rectified Linear Units. Technical report.

Lee, J. H., M. J. Jeon, and S. C. Kim
2012. Uniform random number generator using leap-ahead LFSR architecture. In *Communications in Computer and Information Science*, volume 340 CCIS, Pp. 264–271. Springer, Berlin, Heidelberg.

Mead, C.
1990. Neuromorphic electronic systems. *Proceedings of the IEEE*, 78(10):1629–1636.

Merolla, P. A., J. V. Arthur, R. Alvarez-Icaza, A. S. Cassidy, J. Sawada, F. Akopyan, B. L. Jackson, N. Imam, C. Guo, Y. Nakamura, B. Brezzo, I. Vo, S. K. Esser, R. Appuswamy, B. Taba, A. Amir, M. D. Flickner, W. P. Risk, R. Manohar, and D. S. Modha
2014. A million spiking-neuron integrated circuit with a scalable communication network and interface. *Science*, 345(6197):668–673.

Moore, G. E.
1965. Cramming more components onto integrated circuits. *Electronics*, 38(8).

Morcos, B.
2019. Zynq axi dna reader ip. `https://github.com/abr/zynq-axi-dna`.

Morcos, B., X. Choo, T. Bekolay, and D. Rasmussen
2019. nengo/nengo-fpga: Nengo extension to connect to FPGAs. `https://github.com/nengo/nengo-fpga`.

Morcos, B., T. C. Stewart, C. Eliasmith, and N. Kapre
2018. Implementing nef neural networks on embedded fpgas. In *International Conference on Field-Programmable Technology (FPT)*, Pp. 25–32. IEEE.

Mundy, A., J. Knight, T. C. Stewart, and S. Furber
2015. An efficient SpiNNaker implementation of the Neural Engineering Framework. In *2015 International Joint Conference on Neural Networks (IJCNN)*, Pp. 1–8. IEEE.

Nakahara, H., T. Fujii, and S. Sato
2017. A fully connected layer elimination for a binarized convolutional neural network on an FPGA. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, Pp. 1–4. IEEE.

Nakahara, H., H. Yonekawa, T. Fujii, M. Shimoda, and S. Sato
2019. GUINNESS: A GUI Based Binarized Deep Neural Network Framework for Software Programmers. *IEICE Transactions on Information and Systems*, E102.D(5):1003–1011.

Nane, R., V.-M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y. T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, and K. Bertels
2016. A Survey and Evaluation of FPGA High-Level Synthesis Tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(10):1591–1604.

Navaridas, J., M. Luján, J. Miguel-Alonso, L. A. Plana, and S. Furber
2009. Understanding the interconnection network of SpiNNaker. In *Proceedings of the 23rd international conference on Conference on Supercomputing - ICS '09*, P. 286, New York, New York, USA. ACM Press.

Navaridas, J., M. Luján, L. A. Plana, S. Temple, and S. B. Furber
2015. SpiNNaker: Enhanced multicast routing. *Parallel Computing*, 45:49–66.

Neckar, A., S. Fok, B. V. Benjamin, T. C. Stewart, N. N. Oza, A. R. Voelker, C. Eliasmith, R. Manohar, and K. Boahen
2019. Braindrop: A Mixed-Signal Neuromorphic Architecture With a Dynamical Systems-Based Programming Model. *Proceedings of the IEEE*, 107(1):144–164.

Neil, D. and S.-C. Liu
2014. Minitaur, an Event-Driven FPGA-Based Spiking Network Accelerator. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(12):2621–2628.

Nickolls, J., I. Buck, M. Garland, and K. Skadron
2016. Scalable parallel programming. In *2008 IEEE Hot Chips 20 Symposium, HCS 2008*, Pp. 40–53. IEEE.

Noronha, D. H., K. Gibson, B. Salehpour, and S. J. E. Wilton
2018. LeFlow: Automatic Compilation of TensorFlow Machine Learning Applications to FPGAs. In *International Conference on Field-Programmable Technology (FPT)*.

Pande, S., F. Morgan, S. Cawley, T. Bruintjes, G. Smit, B. McGinley, S. Carrillo, J. Harkin, and L. McDaid
2013. Modular Neural Tile Architecture for Compact Embedded Hardware Spiking Neural Network. *Neural Processing Letters*, 38(2):131–153.

Podobas, A. and S. Matsuoka
2017. Designing and accelerating spiking neural networks using OpenCL for FPGAs. In *2017 International Conference on Field Programmable Technology (ICFPT)*, Pp. 255–258. IEEE.

Rasmussen, D.
2018. Nengodl: Combining deep learning and neuromorphic modelling methods. *CoRR*, abs/1805.11144.

Ritchie, D. M.
1993. The Development of the C Language. Technical report.

Sadri, M., C. Weis, N. Wehn, and L. Benini
2013. Energy and performance exploration of accelerator coherency port using Xilinx ZYNQ. In *Proceedings of the 10th FPGAworld Conference on - FPGAworld '13*, Pp. 1–8, New York, New York, USA. ACM Press.

Schmidhuber, J., F. Cummins, and F. A. Gers
2000. Learning to forget: Continual prediction with LSTM. *Neural Computation*.

Schmidt, A. G., G. Weisz, and M. French
2017. Evaluating Rapid Application Development with Python for Heterogeneous Processor-Based FPGAs. In *2017 IEEE 25th Annual International Symposium on Field- Programmable Custom Computing Machines (FCCM)*, Pp. 121–124. IEEE.

Schuman, C. D., T. E. Potok, R. M. Patton, J. Douglas Birdwell, M. E. Dean, G. S. Rose, and J. S. Plank
2017. A Survey of Neuromorphic Computing and Neural Networks in Hardware. Technical report.

Siddhartha, S. J. E. Wilton, D. Boland, B. Flower, P. Blackmore, P. H. W. Leong, and A. Backpropagation
2018. Simultaneous Inference and Training using On-FPGA Weight Perturbation Techniques. In *International Conference on Field-Programmable Technology (FPT)*, number 3.

Stewart, T. C., T. DeWolf, A. Kleinhans, and C. Eliasmith
2015. Closed-loop neuromorphic benchmarks. *Frontiers in neuroscience*, 9.

Thimm, G. and E. Fiesler
1997. High-order and multilayer perceptron initialization. *IEEE Transactions on Neural Networks*, 8(2):349–359.

Thomas, D. and W. Luk
2009. FPGA Accelerated Simulation of Biologically Plausible Spiking Neural Networks. In *2009 17th IEEE Symposium on Field Programmable Custom Computing Machines*, Pp. 45–52. IEEE.

Thomas, D. B. and W. Luk
2013. The LUT-SR family of uniform random number generators for FPGA architectures. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(4):761–770.

Timotheou, S.
2009. A novel weight initialization method for the random neural network. *Neurocomputing*, 73(1-3):160–168.

Trimberger, S. M.
2015. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proceedings of the IEEE*, 103(3):318–331.

Umuroglu, Y., N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers
2017. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays - FPGA '17*, Pp. 65–74, New York, New York, USA. ACM Press.

Umuroglu, Y., L. Rasnayake, and M. Sjalander
2018. BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing. In *2018 28th International Conference on Field Programmable Logic and Applications (FPL)*, Pp. 307–3077. IEEE.

Voelker, A. R., B. V. Benjamin, T. C. Stewart, K. Boahen, and C. Eliasmith
2017. Extending the neural engineering framework for nonideal silicon synapses. In *2017 IEEE International Symposium on Circuits and Systems (ISCAS)*, Pp. 1–4. IEEE.

Voelker, A. R. and C. Eliasmith
2018. Improving Spiking Dynamical Networks: Accurate Delays, Higher-Order Synapses, and Time Cells. *Neural Computation*, 30(3):569–609.

Wang, R., T. J. Hamilton, J. Tapson, and A. van Schaik
2014a. A compact neural core for digital implementation of the Neural Engineering Framework. In *2014 IEEE Biomedical Circuits and Systems Conference (BioCAS) Proceedings*, Pp. 548–551. IEEE.

Wang, R., T. J. Hamilton, J. Tapson, and A. van Schaik
2014b. An FPGA design framework for large-scale spiking neural networks. In *2014 IEEE International Symposium on Circuits and Systems (ISCAS)*, Pp. 457–460. IEEE.

Wang, R., C. S. Thakur, G. Cohen, T. J. Hamilton, J. Tapson, and A. van Schaik
2017. Neuromorphic Hardware Architecture Using the Neural Engineering Framework for Pattern Recognition. *IEEE Transactions on Biomedical Circuits and Systems*, 11(3):574–584.

Wang, R. and A. van Schaik
2018. Breaking Liebig's Law: An Advanced Multipurpose Neuromorphic Engine. *Frontiers in Neuroscience*, 12:593.

Wijesinghe, W., M. Jayananda, and D. Sonnadara
2006. Hardware Implementation of Random Number Generators. *Proceedings of the Technical Sessions*, 22:28–38.

Xilinx
2019a. Axi dma v7.1 logicore ip product guide. Technical report.

Xilinx
2019b. Pynq community projects. http://www.pynq.io/community.html.

Xilinx
2019c. Pynq: Python productivity for zynq. https://pynq.readthedocs.io.

Yan, Y., D. Kappel, F. Neumaerker, J. Partzsch, B. Vogginger, S. Hoeppner, S. Furber, W. Maass, R. Legenstein, and C. Mayr
2019. Efficient Reward-Based Structural Plasticity on a SpiNNaker 2 Prototype. *IEEE Transactions on Biomedical Circuits and Systems*.

Yinger, J., E. Nurvitadhi, D. Capalija, A. Ling, D. Marr, S. Krishnan, D. Moss, and S. Subhaschandra
2017. Customizable FPGA OpenCL matrix multiply design template for deep neural networks. In *2017 International Conference on Field Programmable Technology (ICFPT)*, Pp. 259–262. IEEE.

APPENDICES

Appendix A

Fixed-Point Precisions

A.1 Hyperopt Search Space

Table A1 outlines the search space bounds used in the hyper-parameter optimization.

Hyper-Parameter		Low	High
DATA_T_IN	**Word Bits**	4	28
	Integer Bits	1	8
DATA_T_ERR	**Word Bits**	19	48
	Integer Bits	1	16
DATA_T_DEC	**Word Bits**	12	32
	Integer Bits	1	16
DATA_T_ENC	**Word Bits**	7	32
	Integer Bits	7	16
DATA_T_RES	**Word Bits**	10	48
	Integer Bits	10	18
K_SHIFT		8	14
$UFAC$		1	28

Table A1: Search space bounds for Hyperopt runs. All parameters are integers and bounds are shown for a closed interval.

A.2 Pareto Optimal Designs

This section will list the Pareto optimal results discovered by Hyperopt as discussed in Sections 3.1.4 & 4.2.

- Table A2 shows results for the cost function that minimizes the error–resource product and uses $UFAC = 1$, $N = 200$ and $D_{in} = D_{out} = 1$ for the simulations.
- Table A3 shows results for the cost function that minimizes the error–cycles product, subject to a resource constraint, and uses $N = 200$ and $D_{in} = D_{out} = 2$ for the simulations leaving $UFAC$ as a free parameter.
- Table A4 shows results for the cost function that minimizes the error–resource product and uses $UFAC = 1$, $N = 64$ and $D_{in} = D_{out} = 1$ for the simulations.
- Table A5 shows results for the cost function that minimizes the error–resource product and uses $UFAC = 1$, $N = 4096$ and $D_{in} = D_{out} = 1$ for the simulations.

Label	**Data Type Precisions (word_bits, integer_bits)**						**Error**	$UFAC$	**Resource %**
	`DATA_T_IN`	`DATA_T_ERR`	`DATA_T_DEC`	`DATA_T_ENC`	`DATA_T_RES`	`K_SHIFT`			
P1	19, 8	42, 11	13, 3	8, 7	16, 16	14	0.129511	1	17
P2	13, 8	40, 10	12, 2	10, 10	17, 16	14	0.014077	1	18
P3	22, 8	47, 12	12, 2	11, 11	16, 16	14	0.008085	1	19
P4	19, 8	43, 11	14, 2	11, 11	16, 16	14	0.003843	1	20
P5	20, 8	44, 9	15, 2	11, 11	18, 16	14	0.003385	1	21
P6	20, 8	46, 11	15, 2	11, 11	18, 16	14	0.003385	1	21
P7	20, 8	47, 12	15, 2	11, 11	18, 16	14	0.003385	1	21
P8	20, 8	48, 13	15, 2	11, 11	18, 16	14	0.003385	1	21
P9	20, 8	46, 13	16, 2	11, 11	20, 16	14	0.003364	1	22
P10	17, 6	37, 4	16, 2	11, 11	27, 7	14	0.003359	1	23
P11	25, 5	37, 15	15, 2	13, 11	26, 16	14	0.003344	1	24

Table A2: List of Pareto optimal fixed-point precisions extracted by Hyperopt by minimizing the error–resource product. We use $UFAC = 1$ and set $N = 200$ and $D_{in} = D_{out} = 1$ for the simulations.

Label	Data Type Precisions (word_bits, integer_bits)						Error	*UFAC*	Resource %
	DATA_T_IN	DATA_T_ERR	DATA_T_DEC	DATA_T_ENC	DATA_T_RES	K_SHIFT			
P1-C	19, 17	23, 1	25, 12	12, 11	22, 17	14	0.003336	20	85
P2-C	16, 6	24, 2	23, 10	16, 11	22, 17	14	0.003315	2	32
P3-C	16, 6	23, 1	25, 12	13, 11	19, 17	14	0.003346	20	85
P4-C	11, 2	24, 2	25, 9	23, 11	17, 17	14	0.003433	28	100
P5-C	13, 1	23, 1	20, 7	24, 11	19, 16	14	0.003433	24	130

Table A3: List of Pareto optimal fixed-point precisions extracted by Hyperopt by minimizing the required cycles subject to a resource constraint. We set $N = 200$ and $D_{in} = D_{out} = 1$ for the simulations and leav $UFAC$ as a free parameter.

Label	Data Type Precisions (word_bits, integer_bits)						Error	*UFAC*	Resource %
	DATA_T_IN	DATA_T_ERR	DATA_T_DEC	DATA_T_ENC	DATA_T_RES	K_SHIFT			
P1-64	27, 2	40, 16	17, 2	10, 10	19, 16	10	0.006600	1	22
P2-64	22, 4	38, 16	18, 4	23, 10	22, 18	11	0.006599	1	34
P3-64	21, 1	46, 8	13, 1	10, 10	15, 13	12	0.007035	1	19
P4-64	21, 4	46, 7	12, 1	10, 10	15, 16	12	0.009562	1	18
P5-64	12, 2	43, 3	14, 1	10, 10	17, 14	12	0.006608	1	20
P6-64	16, 2	45, 3	12, 1	9, 9	17, 14	14	0.394523	1	17
P7-64	15, 2	23, 2	15, 2	10, 10	17, 14	12	0.006570	1	55

Table A4: List of Pareto optimal fixed-point precisions extracted by Hyperopt by minimizing the error–resource product. We use $UFAC = 1$ and set $N = 64$ and $D_{in} = D_{out} = 1$ for the simulations.

Label	Data Type Precisions (word_bits, integer_bits)						Error	*UFAC*	Resource %
	DATA_T_IN	DATA_T_ERR	DATA_T_DEC	DATA_T_ENC	DATA_T_RES	K_SHIFT			
P1-4k	19, 3	40, 14	31, 12	11, 10	26, 16	14	0.004306	1	33
P2-4k	20, 3	42, 14	15, 13	10, 10	25, 16	14	0.636612	1	21
P3-4k	17, 4	40, 13	12, 12	9, 9	25, 14	14	0.636612	1	18
P4-4k	28, 7	48, 1	21, 5	9, 9	19, 17	14	0.022840	1	24
P5-4k	25, 8	46, 2	20, 3	12, 10	24, 17	14	0.004463	1	26
P6-4k	23, 8	31, 1	16, 1	14, 10	27, 17	14	0.007443	1	25
P7-4k	22, 8	29, 1	19, 1	15, 10	26, 17	14	0.004353	1	28
P8-4k	21, 8	29, 1	19, 1	16, 10	29, 17	14	0.004345	1	30
P9-4k	22, 8	30, 2	20, 2	13, 10	26, 17	14	0.004353	1	27

Table A5: List of Pareto optimal fixed-point precisions extracted by Hyperopt by minimizing the error–resource product. We use $UFAC = 1$ and set $N = 4096$ and $D_{in} = D_{out} = 1$ for the simulations.

A.3 Selected Fixed-Point Designs

Table A6 shows selected best designs for our application from the sets of Pareto optimal designs.

Label	Data Type Precisions (word_bits, integer_bits)						Error	*UFAC*	Resource %
	DATA_T_IN	DATA_T_ERR	DATA_T_DEC	DATA_T_ENC	DATA_T_RES	K_SHIFT			
P10	17, 6	37, 4	16, 2	11, 11	27, 7	14	0.003359	24	98
P4-C	11, 2	24, 2	25, 9	23, 11	17, 17	14	0.003433	28	100
P1-64	27, 2	40, 16	17, 2	10, 10	19, 16	10	0.006600	24	94
P5-4k	25, 8	46, 2	20, 3	12, 10	24, 17	14	0.004463	24	100
General	25, 8	48, 4	20, 3	13, 11	27, 17	13	0.010830	20	85

Table A6: Selected fixed-point designs from the Pareto optimal sets.

Appendix B

Power & Performance Data

For both the FPGA and the GPU power measurements were taken using the P3 P4455 plug-in Power Monitor. Devices were warmed up by running several networks before beginning measurements. The static power was noted before and after evaluation runs, then averaged and subtracted from the total power to arrive at dynamic power numbers for the entire device.

B.1 Jetson TX1 Power & Performance

The static power of the Jetson TX1 board was discovered to be 2.6 W. The values in Table B1 were collected by observing power and step time values over 1 million evaluation steps.

B.2 FPGA Power & Performance

The static power of the PYNQ-Z1 board was measured to be 2.5 W. The values in Table B2 were collected by observing power and step time values over 1 million evaluation steps.

Dimensions	Neurons	Mean Step Time (us)	Mean Total Power (W)	Mean Dynamic Power (W)
	64	328.304	6.4	3.8
1	512	456.582	5.7	3.1
	4096	938.075	3.8	1.2
	64	319.022	6.4	3.8
2	512	456.625	5.9	3.3
	4096	1085.20	3.8	1.2
	64	331.768	6.4	3.8
4	512	385.472	6.0	3.4
	4096	676.593	4.2	1.6
	64	320.431	6.4	3.8
8	512	381.833	6.1	3.5
	4096	517.866	4.9	2.3

Table B1: Power and performance data collected from the Jetson TX1 GPU device.

Dimensions	Neurons	Mean Step Time (us)	Mean Total Power (W)	Mean Dynamic Power (W)
	64	0.68	2.9	0.4
1	512	1.96	2.9	0.4
	4096	12.20	2.9	0.4
	64	0.82	3.0	0.5
2	512	2.74	3.0	0.5
	4096	18.10	3.0	0.5
	64	1.10	3.0	0.5
4	512	4.30	3.0	0.5
	4096	29.90	3.0	0.5
	64	1.66	3.0	0.5
8	512	7.42	3.0	0.5
	4096	53.50	3.0	0.5

Table B2: Power and performance data collected from the PYNQ-Z1 FPGA device.