Implementing NEF Neural Networks on Embedded FPGAs

Benjamin Morcos[1,2], Terrence C. Stewart[1], Chris Eliasmith[1], Nachiket Kapre[2]
[1]Applied Brain Research, Waterloo, Canada, {ben.morcos, terry.stewart, chris.eliasmith}@appliedbrainresearch.com
[2]University of Waterloo, Canada, {bmorcos, nachiket}@uwaterloo.ca

***Abstract*—** **Low-power, high-speed neural networks are critical for providing deployable embedded AI applications at the edge. We describe an FPGA implementation of Neural Engineering Framework (NEF) networks with online learning that outperforms mobile GPU implementations by an order of magnitude or more. Specifically, we provide an embedded Python-capable PYNQ FPGA implementation supported with a 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. 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 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 iterations of Vivado HLS before running the complete Vivado place-and-route phase on that subset. 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 (4× improvement), and low-latency direct I/O access (1000× improvement).**

I. Introduction

As the end of Moore's law 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 DSP and RAM blocks.

Machine learning (ML) developers commonly use Python as their development environment. Many popular packages such as Tensorflow, Keras, Nengo, 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 programming environments, and ease of integration with Python.

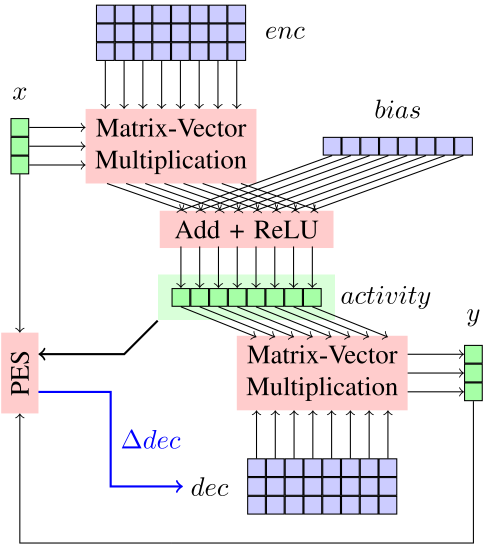


Fig. 1: High-Level picture of dataflow in the Neural Engineering Framework (NEF) network evaluation. We consume inputs x, perform Matrix-Vector multiplication with enc, add the $bias$ to the result and generate the $activity$ vector. The result y is generated through another Matrix-Vector multiplication with the dec matrix. The enc matrix is generated randomly, while the dec matrix is trained and updated by the online Prescribed Error Sensitivity (PES) learning rule.

FPGAs have been seen as exotic devices that have a steep learning curve for 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 boards. The PYNQ environment has a clean API for configuration of the FPGA, data movement using DMA, access to GPIO blocks, and more. A combination of HLS and PYNQ is a more attractive starting point for ML 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 past attempts at marrying ML and embedded FPGAs [7], [11], [13], [14], but most focus primarily on convolutional networks. The goal of this paper is to present an optimized FPGA backend that integrates HLS-generated hardware wrapped in PYNQ APIs with the Nengo [1] 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) [5]. 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 [6], was built using Nengo, demonstrating the rich range of capabilities of the framework. Nengo currently supports CPU, GPU, SpiNNaker [10], and other backends and is now also able to target PYNQ FPGA boards.

In summary, we develop an FPGA backend for Nengo to realize low-power, low-latency embedded systems that use neural network structures with online learning. We use an HLS description of the neural networks that is parametric and flexible enough to cover a range of implementation possibilities. In addition, we reformulate the parallelism in the description in order to overcome the limitations of Vivado HLS and expose dataflow and pipeline parallelism. Furthermore, we reduce the precision of the arithmetic operations using Hyperopt [3] to 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. Finally, to demonstrate fully embedded performance, we bypass the ARM host CPU and directly integrate sensors and actuators with the adaptive neural network architecture over GPIO.

II. Background

This section introduces the Neural Engineering Framework (NEF) [5] which is the basis of the implementation described in this paper. Figure 1 shows the architecture and data flow for one NEF population of neurons with online learning.

A. The Neural Engineering Framework

The NEF provides methods for implementing spiking or non-spiking, dynamic neural computations in arbitrary vector spaces while allowing flexibility in details such as the neuron model or method of adaptation. In addition to arbitrary feedforward networks, the NEF lends itself to biologically plausible cognitive architectures and the control and modelling of dynamic systems. This makes it uniquely well-suited to online, real-time and recurrent networks, which differs from the contemporary focus of backpropagation-trained networks. The NEF is built on the principles of *representation*, *transformation*, and *dynamics*. For the purposes of this paper, we focus primarily on the *representation* and *transformation* aspects.

Representation: The NEF takes a D_{in}-dimensional time-varying vector, x, as an input defined in the "state-space" and maps it to an N-dimensional representation in "neuron-space". We call this mapping from state-space to neuron-space *encoding*. Each vector element in this neuronal representation corresponds to the activity, a_i, of a single neuron, i, and can be expressed as

$$a_i = G[enc_i \cdot x + bias_i] \tag{1}$$

where

- enc_i is the ith row of the $[N \times D_{in}]$ encoder matrix that defines the preferred stimulus of a neuron,
- $bias_i$ is a bias term that accounts for background activity in a neuron, and
- G is the non-linear transfer function of the neuron model, which in this case is the Rectified Linear Unit (ReLU) defined as $G[v] = \max(0, v)$.

The neuron activity, a_i, is then *decoded* to produce a D_{out}-dimensional vector, y, back in state-space according to

$$y_j = dec_j \cdot a \tag{2}$$

where

- dec_j is the jth row of the $[D_{out} \times N]$ decoder matrix that approximates the represented value in state-space given the activity of the neurons in neuron-space, and
- a is the vector all neuron activities.

The decoders can be obtained by least-squares optimization of the error $E_x = ||x - y||$. More generally, according to the second NEF principle (*transformation*) we can find decoders that represent an arbitrary linear or non-linear function, $f(x)$, by minimizing $E_x = ||f(x) - y||$ instead. In certain embedded PD control applications, the entire E_x may be supplied externally to approximate the inverse kinematics of the problem.

The decoders may also be initialized to non-optimal values that are adjusted in real-time using online learning according to a particular learning rule. In this case we employ the Prescribed Error Sensitivity (PES) [2] learning rule which updates the decoders each step by

$$\Delta \boldsymbol{dec} = -KE_x \cdot a \tag{3}$$

where

- K is the learning rate that controls how quickly the system will adapt, and
- E_x is the error signal for the given system which can be calculated (*i.e.* $E_x = f(x) - y$), but can also be provided directly by an external source.

The neural network developer must choose a reasonable value for K that 1) does not overshoot and oscillate, while 2) converging quickly enough to adapt to system changes in real-time. The neural activity, a, is included in the decoder update term so that only those neurons involved in representing the given stimulus in neuron-space are updated while neurons not involved are left unaffected.

We start with set values for enc and $bias$ elements which are selected at random. The x, $activity$, and y quantities vary with each timestep. The dec matrix is also updated once per timestep using the PES rule and replaces conventional backpropagation with an online learning approach. For D_{in} inputs, N neurons, and D_{out} outputs, this evaluation represents a total of $(D_{in} + D_{out}) \times N$ operations. When interfacing with sensors and actuators, the acquisition of x inputs and y outputs may present additional time penalties on various platforms. The evaluation time for one pass through the computation flow in Figure 1 should be minimized to allow the system to respond to real-time signal characteristics for a large number of neurons and dimensions.

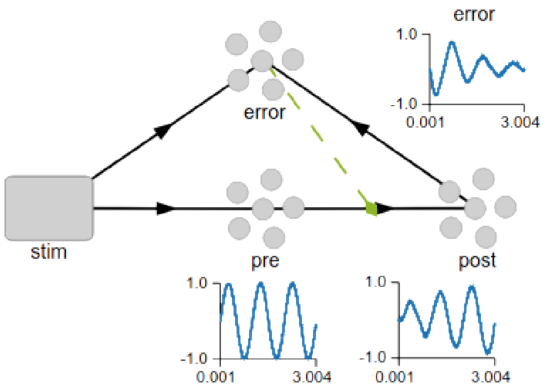


Fig. 2: The Nengo GUI displaying an adaptive model with neural populations *pre*, *post*, and *error* with input *stim*, which is a sine wave. Black solid lines show connections and the green dashed line represents the error signal used by the learning rule to updates the decoders. The plots show the represented value on the vertical axis and the simulation time on the horizontal axis.

B. Nengo

Nengo [1] is a Python framework that enables high-level description, debugging, visualization, analysis, and deployment of neural networks. It is actively used and developed with 24 contributors, 333 stars, and 110 forks on github (https://github.com/nengo/nengo) at the time of writing. While alternative frameworks exist [1] 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. Nengo is also largely agnostic to the backend implementation which allows the user to target several hardware backends including CPUs, GPUs, SpiNNaker, and now FPGAs (with this paper). A key feature of Nengo is the optional integrated graphical user interface (GUI) that helps to develop and visualize computations. Figure 2 shows how the GUI displays a simple network including connections and real-time values from the simulation. The implementation described in this paper encompasses the *error*, *pre*, and *post* populations seen in the GUI. That is to say we accept an input (*stim*) that is represented internally (*pre*) and produce an output (*post*) that is learned based on an error signal.

III. NengoFPGA

In this section, we describe the design and engineering of NengoFPGA, a new FPGA backend for Nengo. The foundations of the NEF approach to representation and transformation shown in Equations 1, 2, and 3 are embodied in the basic sketch shown in Algorithm 1. This is the starting point for the High-Level Synthesis implementation and the basis of the following discussion of improvements, and refinements.

A. Design Parametrization and Weight Storage

To support the use of NengoFPGA in various embedded scenarios, we first target parametrization of the design. We identify the problem size model parameters N, D_{in}, and D_{out} as the primary factors that determine the neural network configuration. The weight matrices (enc, dec) and vector ($bias$) need to be loaded only once into the chip at startup, and require the memory sizes to be fixed based on the design parameters. 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. The size of the network that can be held on chip is up to 32k neurons but varies with design choice: having fewer bits of precision will allow more values to be stored, and similarly, having less parallelism requires less replication (*e.g.* for inputs) and can also allow larger networks to be stored but can slow down overall network response times. Our optimization tool, Hyperopt, allows the exploration of such implementation parameters (or hyper-parameters). The set of hyper-parameters can be automatically tuned to suit the resource, accuracy, and performance requirements of the application. For subsequent analyses we allocate memory for $N = 4096$ and $D_{in}, D_{out} = 8$ as the maximum size at compile time for all designs.

B. HLS Formulation

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:

- The NEF implementation has temporal dependency. Since consecutive evaluations of the neural model have a temporal dependency, inputs and outputs to this NEF implementation must be streamed sequentially. This prevents inputs from being batched and evaluated in parallel which, is commonly done with other models.
- 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. To maximize the extent of parallelization for both the encode and decode computations, we must 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 Eq. 2 requires a contribution from each neuron to create the output, 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, y_k, is generated in each parallel section. After the decode step, the partial results are then accumulated into the single output, $y = \sum_k y_k^{part}$, and streamed out sequentially.
- Each parallel section is allocated an independent portion of the encoder, enc, and decoder, dec, weights. However, the input, x, and error signal, E_x, must be shared between threads.

Inspection of data dependencies in the inner loops on lines 2 and 8 of Algorithm 1 suggest that they should each take 2 cycles to compute. The overall network should take $2 * N * (D_{in} + Dout) + 2 * N$ cycles to evaluate. For example, a model with $N = 200$ and $D_{in} = D_{out} = 2$ should require 2000 cycles. For the first-pass HLS implementation in Vivado HLS 2016.1, the decode loops are perfectly nested and are automatically flattened but has $II = 6$. The encode inner loop has $II = 5$ and the outer loop is not pipelined since it cannot be flattened and the inner loop cannot be unrolled. As a result, Algorithm 1: A basic sketch of the NEF implementation without consideration for hardware or HLS structure or limitations.

```
1  for $i < N$ do
2      $a_i = J_i^{bias}$
3      for $j < D_{in}$ do
4          $a_i \mathrel{+}= x_j * enc_{ij}$
5      end
6      $a_i = G_i(a_i)$
7  end
8  for $j < D_{out}$ do
9      for $i < N$ do
10         $y_j \mathrel{+}= a_i * dec_{ji}$
11         $dec_{ji} \mathrel{+}= -K * E_{x_j} * a_i$
12     end
13 end
```

Algorithm 2: A sketch of the NEF implementation with the decode step restructured to more closely match the structure of the encode step.

```
1  for $i < N$ do
2      $a_i = J_i^{bias}$
3      for $j < D_{in}$ do
4          $a_i \mathrel{+}= x_j * enc_{ij}$
5      end
6  end
7  for $i < N$ do
8      $a_l = a_i$
9      for $j < D_{out}$ do
10         $y_j \mathrel{+}= G_i(a_l) * dec_{ji}$
11         $dec_{ji} \mathrel{+}= -K * E_{x_j} * G_i(a_l)$
12     end
13 end
```

Algorithm 3: (Right) A sketch of the NEF implementation that has been restructured for parallelism by a factor of UFAC across the number of neurons, N.

```
1  for $k < UFAC$ do
2      for $i < \lceil \frac{N}{UFAC} \rceil$ do
3          $a_i = J_{ki}^{bias}$
4          for $j < D_{in}$ do
5              $a_i \mathrel{+}= x_{kj} * enc_{ij}$
6          end
7      end
8      for $i < \lceil \frac{N}{UFAC} \rceil$ do
9          $a_l = a_i$
10         for $j < D_{out}$ do
11             $y_{kj}^{part} \mathrel{+}= G_i(a_l) * dec_{ji}$
12             $dec_{ji} \mathrel{+}= -K * E_{x_{kj}} * G_i(a_l)$
13         end
14     end
15 end
16 for $j < D_{out}$ do
17     for $k < UFAC$ do
18         $y_j \mathrel{+}= y_{kj}^{part}$
19     end
20 end
```

the design requires 6026 cycles for a model of this size which is much worse than our estimate.

To overcome the limits of the HLS compilation, we have to supply additional information [8] to the compiler. We perform the following optimizations guided by the need to describe HLS code with care:

1. Restructuring of the decode loops to prefer parallelization over N as it is larger and contains most parallelism;
2. Unroll-friendly description of the design with a third level of loop that the compiler can recognize as parallelizable; and
3. Pipeline and 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 decode process is inverted as seen in Algorithm 2. We do not fuse the outer loops over encode and decode operations to ensure the inner loops are fully optimized in isolation. Since we are using the simple Rectified Linear neuron model, we move G into the decode loop as a ternary operation to simplify the encode loop. With the restructured loops, we also introduce a_l on line 8 to exploit data reuse. As a result of these optimizations, we improved the decode loop two-fold with $II = 3$, now requiring 4829 cycles overall.

Parallelization: Once the outer loops have been harmonized between the encode and decode loops, 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 of unroll. 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. This alone was not sufficient to improve performance. The network weights, enc and dec, required by each parallel section were partitioned along their N-dimension but instead they had to be explicitly partitioned using an extra dimension in the array structure (*i.e.* `enc[N][D]` becomes `enc[UFAC][N/UFAC][D]`). This three-dimensional structure *was* understood by the compiler for concurrent access. We add extra logic to handle the cases when $UFAC$ is not a factor of N; zero-padding arrays and computing loop bounds as the ceiling $i < \lceil \frac{N}{UFAC} \rceil$. With this foundation for capturing parallelism, we finally turned to explicit pragma hints to expose the pipeline and dataflow parallelism in the problem.

Pipeline and Dataflow Pragmas: The use of flexible, parametrized code requires variable loop bounds that cannot be easily unrolled. In order to use the `PIPELINE` pragma on a nested loop, all sub-loops must be fully unrolled. As a result, nested loops with variable bounds are unable to be pipelined automatically. This leaves the large encode loop over N untouched. Variable scope also plays an important role, forcing us to declare the activities, a, within the **for** loop on line 1 of Algorithm 3. The `DATAFLOW` pragma optimization attempts to run all blocks within a module or loop concurrently using dataflow analysis to identify dependencies. After adding the `DATAFLOW` directive, the design is able to take advantage of parallelism.

These improvements have progressed from 6026 cycles using a naive approach to 4829 cycles with some restructuring and with an unroll factor of 12 ($UFAC = 12$), the latest design shows an improvement of 15× requiring only 399 cycles.

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.

C. Fixed-point Design

Floating-point numbers support representation of numerical quantities with high dynamic range, but neural network operations are often amenable to fixed-point precision with little, if any, loss in accuracy [11], [13] . For our NEF implementation, we identify a set of different variable groups for custom fixed-point representation including the input, x, output, y, encoder, enc, and decoder, dec, matrices, along with intermediate values. Furthermore, to ensure accurate scaling of the quantities, we introduce a hyper-parameter, K_SHIFT, that normalizes the values depending on the learning rate K. The proper selection of this scaling parameter reduces the number of fractional bits required [9]. The final result extracted from the evaluation is then shifted back to recover the signal.

The switch to fixed-point representation further reduces the cycle count by over 4× to 92 using and unroll factor of 28 ($UFAC = 28$) for our example with $N = 200$ and $D_{in} = D_{out} = 2$. The choice of variable precisions and scaling parameters to achieve this result defines a search space that is intractable for a brute-force search with $\approx 10^{20}$ possibilities. As a result, a more efficient strategy is necessary.

D. Parameter Tuning

We use a hyper-parameter tuning package called Hyperopt [3] 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 type. We made an intuitive pre-selection of the rounding and overflow modes for our types to reduce the search space. For data storage (weights) and transmission variables (input), we round towards +∞, and saturate the overflow values. For internal arithmetic, we use truncation of small values below one least significant bit and wrapping on overflow to improve computation speed.

Initial Bounding: To compute the bounds for the search, we initially set all types to floating-point as a reference design. Then we set each type to a large 64-bit fixed-point representation and select each fixed-point type individually for inspection. In each inspection, we sweep the candidate's fixed-point precision (integer and fraction bits) from high (64 bits) to low (1 bit) 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 $E_x = |x - y|$. We use the HLS implementation of an adaptive controller that learns to represent a sinusoid as the test case for evaluating error. As our reference design uses floating-point, there will be an inherent error, E_x^{float}. We expect our fixed-point solution error, E_x^{fixed}, to be marginally better than the reference floating-point solution. For each set of parameters, we identify the smallest precision where the floating-point error exceeds fixed-point error, $E_x^{float} \geq E_x^{fixed}$.

Optimization: Once our bounds are set up, we perform the optimization using a cost function. The HLS code that evaluates error is repurposed to also return the cost values (resources, and scheduled cycle counts) through a single step of the HLS 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 $E_x * cycles$, but other cost functions produced competitive results more quickly. We also use the cost function $E_x * resources$, where the resource cost is the maximum % 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 cycle count, since resource usage determines the degree of parallelism that is possible. Thus this cost function produces designs balanced in accuracy and performance. In addition, using $UFAC = 1$ reduces the effort required by the HLS compiler, which allows Hyperopt to run 3–4× faster than minimizing cycles directly, while only incurring a small performance penalty.

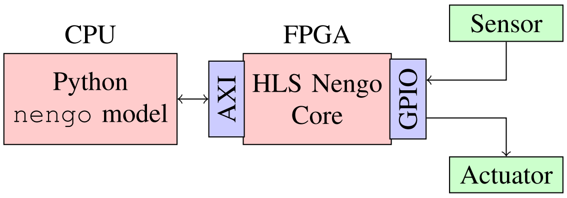


Fig. 3: System level view of the NengoFPGA design mapped to the PYNQ board. Nengo models are defined by Python code running on the CPU and interact with the FPGA accelerator over AXI. The FPGA fabric interfaces directly with the external inputs and outputs over GPIO.

Usage: Hyperopt exploration is configured in a Python script containing the structure of the problem. Hyperopt is then connected to a shell script that invokes the HLS compilation as well as the adaptive controller test.

IV. Methodology

We implement the NengoFPGA backend split across the ARM CPU, and the FPGA fabric of the PYNQ-Z1 Zynq board. The core *nengo* package is directly integrated with the *nengo-fpga* package to seamlessly connect standard Nengo models directly with the FPGA backend.

To use this environment, the neural model is first described using a typical Nengo network description. The *nengo-fpga* extension uses PYNQ drivers to load the network weight matrices to the FPGA on-chip RAMs once at the start of a simulation. This system-level description is diagrammed in Figure 3. We interface the FPGA HLS core directly with GPIO for getting inputs and driving outputs, to remove the ARM processor from the loop. This lets us run the adaptive neurons directly connected to GPIO pins. 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 DMA transfer between the ARM and FPGA. Even small networks cannot improve beyond a step time of 715μs compared to a direct I/O design that can operate over 1000× faster at a sub-microsecond step time of 0.678μs for $N = 64$. We use Vivado 2016.1 with PYNQ version 2.1 for our mapping.

V. Results

In this section, we evaluate the performance, resource usage, and error behaviour of our NengoFPGA implementation.

A. 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 III-B, 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. As shown in Figure 4, the bulk of the speed up comes from unrolling, but the restructuring along with pragmas (`DATAFLOW+PIPELINE`) lay the groundwork to deliver this performance. The use of fixed-point types permits an extra 2× improvement in unroll factor and reduces cycle count further by over 4× when given the same resource budget (the entire chip).

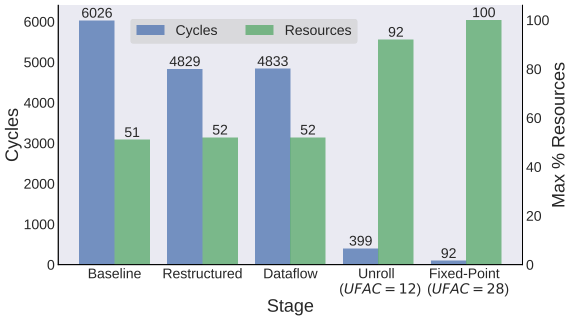


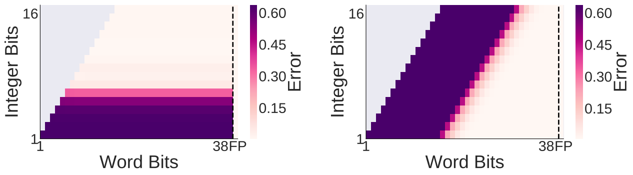
Fig. 4: Performance improvements due to incremental, cumulative optimizations of HLS code. *Baseline* is the naive approach, *Restructured* has the decode loop inverted, *Dataflow* includes pipeline and dataflow pragmas, *Unroll* is unrolled with $UFAC = 12$, and *Fixed-Point* is the fixed-point design unrolled with $UFAC = 28$

B. Impact of Parameter Tuning with Hyperopt

As discussed earlier, a brute-force exploration of the design space is intractable. Hyperopt can help search this design space if we first bound the possible range of parameter values.

Initial Bounding: In Figure 5a, we show the effect of varying the enc word bits from 1–38 and varying integer bits from 1–16. The diagonal cut off indicates infeasible combinations where integer bits exceed total word bits. We see that as we increase integer bits above 7 and 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 5b illustrates the effect of precision selection of the error signal, E_x, on overall accuracy. Here, an increase in the number of word bits above 17–18 yields low error. Thus, most of the accuracy improvement is tied to having high precision representation of small quantities.

Sensitivity of Precision to Design Parameters: The error signal, E_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 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 space of design parameters, in this case from $N = 64$ to $N = 4096$. This design performs well over all values of N considered (see Figure 6).



(a) Accuracy of ENC type

(b) Accuracy of ERR type

Fig. 5: Effect of varying precision of single parameters (rest locked at 64b fixed-point). Encoder bits should have a sufficient number of integer bits, while error signals being small need a larger number of fraction bits.

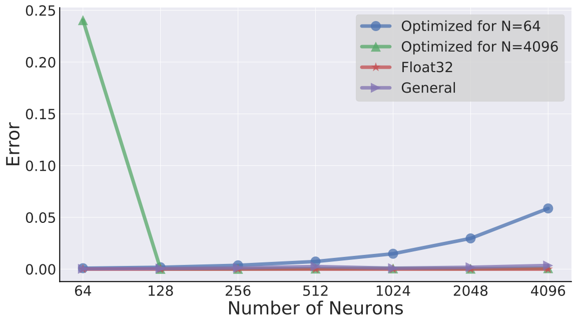


Fig. 6: Representational error of designs optimized using different values of N in the Hyperopt simulation compared to a general fixed-point solution and a floating-point solution.

Convergence Rate: Hyperopt is fundamentally limited by the speed of the HLS tools. The frontend HLS passes must be run to produce resource and cycle count estimates. As shown in Figure 7, the best observed configuration (lowest cost) is reached after ≈ 1500 trials, which takes ≈ 12 hours to run on an Intel i7-6700K CPU. However, it takes ≈ 500 trials, or about 4 hours, to get within 99% of the best design (lowest cost). Thus, Hyperopt is a tractable approach for optimizing design parameters with Vivado HLS in the loop for a few hours of trials.

Hyperopt Design Optimization: Starting from the optimized HLS implementation, and the bounded parameter ranges, we begin the Hyperopt optimizations. We choose the cost function $E_x * cycles$, subject to a resource threshold. Designs above the resource threshold are not discarded, but rather aggressively penalized so Hyperopt will stay below the threshold. The thresholds used were 100% for BRAMs and LUTs and 120% 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 DSP requirement. This Hyperopt run produced 5

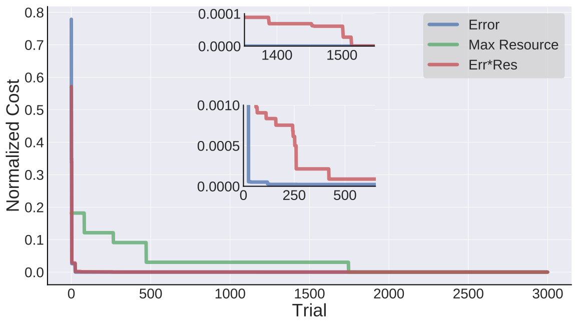


Fig. 7: Normalized convergence of cost minimization for Hyperopt trials. These are independent, best-so-far trends, so combining Error and Resource trends may not result in the overall Error×Resource cost trend.

Pareto optimal designs, of which 1 is infeasible (too many resources) and 3 are slow. The final design of the 5 is the fastest design discovered. The cost function $E_x * resources$ produced 11 Pareto optimal designs 3–4× faster than the $E_x * cycles$ minimization, and all of which are feasible and competitive in performance. As a result, the $E_x * resources$ minimization is analysed herein as it produces multiple good designs, better illustrating the possible solution space. Figure 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 9. The final optimized designs 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.

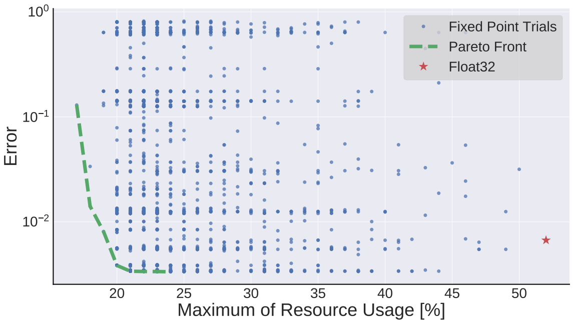


Fig. 8: The Pareto optimal front resulting from Hyperopt trials minimizing the product $E_x * resources$.

Pareto Optimal Designs: Referring to Figure 9, We observe that there is little difference in resource usage between designs of the $E_x * resources$ optimization. These optimized designs are 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 $E_x * cycles$ minimization (labelled Cycles) is unrolled to a factor of 28 ($UFAC = 28$) resulting in slightly higher performance.

As well, there exists an accuracy-resource trade-off where the Pareto optimal fixed-point designs achieve an unroll factor of 24–28, while the floating-point design is limited to a factor of 12. The general fixed-point solution using the largest integer and fraction bit configurations from the Pareto designs can only be unrolled to a factor of 20.

The reduced BRAM usage in design 1 indicates reduced precision in the larger array structures *enc*, *dec*, and *activity* that results in high error. The reduced DSP usage in design 2 indicates reduced precision throughout the arithmetic which also leads to reduced accuracy. Designs 3–8 all have similar resource usage, however designs 3 and 4 appear to be using precision ineffectively. Designs 9–11 begin to show diminishing returns as they use far more resources without a significant improvement in accuracy. The "Cycles" design has comparable resources and error but is justified in its increased speed. The floating-point design also uses comparable resources but it has half the parallelism as the Pareto designs and therefore much worse performance, as discussed later. 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 over large dynamic ranges.

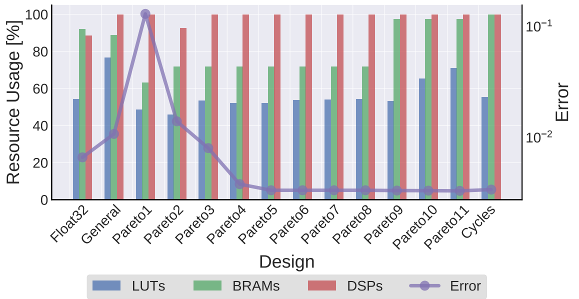


Fig. 9: Resource usage and Error trade-offs for the $E_x * resources$ Pareto optimal designs compared to the $E_x * cycles$ optimal design, the general fixed-point, and the floating-point solution. $N = 200$, $D_{in} = D_{out} = 2$.

Floating-point vs. Fixed-Point Speed: We now compare the speeds 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 take advantage of both DSP and LUTs effectively to support an unroll factor up to 28. The 1-D trials in Figure 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.

PYNQ vs. Jetson TX1 GPU: The Jetson TX1 is not the latest embedded GPU, however, it uses 20 nm technology and the PYNQ board uses an FPGA with 28 nm technology making the TX1 a reasonable comparison. We developed an optimized CUDA implementation of Nengo 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 $E_x = |f(x) - y|$, 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

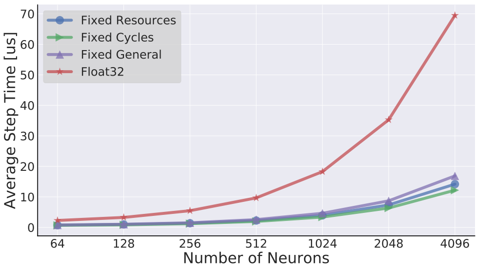
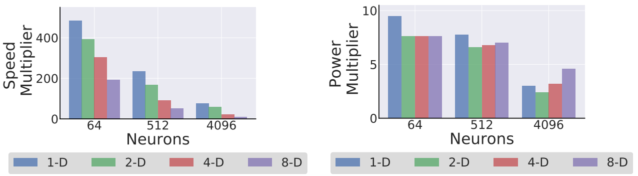


Fig. 10: Comparing execution times of a floating-point design, a general fixed-point design, and two differently optimized fixed-point designs.



(a) Speedup over GPU

(b) Power efficiency over GPU

Fig. 11: GPU performance normalized against FPGA performance. The FPGA is 10–484× faster and uses 2.4–9.5× less dynamic power.

be copied over requiring frequent host-device transfers every timestep of evaluation. Coupled with the small size of the problem, the I/O heavy nature of the computation puts the GPU at a disadvantage compared to the PYNQ FPGA.

The performance of the GPU compared to the FPGA is summarized in Figure 11. As expected, it shows that the FPGA is 10–484× faster, and uses 2.4–9.5× less dynamic power than the GPU (measured with a P3 P4455 plug-in Power Monitor). 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. 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.

Spinnaker: Nengo-Spinnaker can evaluate ≈8K neurons with 16-D at 1 ms execution time on a single 16-core custom ARM chip operating on a 1.5 W power budget. Our PYNQ board can process 8K neurons with 8-D in 106μs with a 3 W power draw on a single Zynq chip. This translates into a 4.5× performance advantage (linear scaling to account for 8-D vs. 16-D) with 2× more power resulting in more than a 2× better energy efficiency ratio in favour of the FPGA.

C. *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:

- A population of 3000 neurons with 6 representational dimensions can form the basis of an efficient adaptive motor controller [4]. NengoFPGA can evaluate a model this size in under 31μs.
- As few as 500 neurons can adapt to a randomly generated 15-joint body simulation [12]. We evaluate 500 neurons with $D_{in} = 30$ (15×2 for position and velocity) in under 24μs.
- 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.

VI. 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. We have shown how a structured HLS approach tailored to 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 cycles by an addition 4× — an overall improvement of over 65×. 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 ARM DMA, from a step time of 715μs to 0.678μs. 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.

This project is the first step in creating a fully featured FPGA backend for the Nengo neural network development environment. Beyond low-power, low-cost embedded scenarios, we plan to extend this work to larger FPGAs to address larger cloud and edge computing workloads.

References

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

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

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

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

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

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

[7] Y. Hao and S. Quigley. The implementation of a deep recurrent neural network language model on a xilinx FPGA. *CoRR*, abs/1710.10296, 2017.

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

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

[10] A. Mundy, J. Knight, T. C. Stewart, and S. Furber. An efficient spinnaker implementation of the neural engineering framework. In *IJCNN*, 2015.

[11] H. Nakahara, T. Fujii, and S. Sato. A fully connected layer elimination for a binarizec convolutional neural network on an fpga. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2017.

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

[13] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. H. W. Leong, M. Jahre, and K. A. Vissers. FINN: A framework for fast, scalable binarized neural network inference. *CoRR*, abs/1612.07119, 2016.

[14] Y. Umuroglu, L. Rasnayake, and M. Sjalander. BISMO: A Scalable Bit-Serial Matrix Multiplication Overlay for Reconfigurable Computing. *ArXiv e-prints*, June 2018.