Innervator:
Hardware Acceleration for Neural Networks

Fereydoun Memarzanjany (فریدون معمارزنجانی)
Student Member, IEEE
E-Mail: thraetaona@ieee.org

***Abstract*—Artificial intelligence ("AI") is deployed in various applications, from noise cancellation to image recognition, but AI-based products often come with high hardware and electricity costs; this makes them inaccessible for consumer devices and small-scale edge electronics. Inspired by biological brains, deep neural networks ("DNNs") are modeled using mathematical formulae, yet general-purpose processors treat otherwise-parallelizable AI algorithms as step-by-step sequential logic. In contrast, programmable logic devices ("PLDs") can be customized to the specific parameters of a trained DNN, thereby ensuring data-tailored computation and algorithmic parallelism at the register-transfer level. Furthermore, a subgroup of PLDs, field-programmable gate arrays ("FPGAs"), are dynamically reconfigurable. So, to improve AI runtime performance, I designed and open-sourced my hardware compiler: Innervator. Written entirely in VHDL-2008, Innervator takes any DNN's metadata and parameters (e.g., number of layers, neurons per layer, and their weights / biases), generating its synthesizable FPGA hardware description with the appropriate pipelining and batch processing. Innervator is entirely portable and vendor-independent. As a proof of concept, I used Innervator to implement a sample 8×8-pixel handwritten digit-recognizing neural network in a low-cost AMD Xilinx Artix-7™ FPGA @ 100 MHz. With 3 pipeline stages and 2 batches at about 67% LUT utilization, the Network achieved ~7.12 GOP/s, predicting the output in 630 ns and under 0.25 W of power. In comparison, an Intel® Core™ i7-12700H CPU @ 4.70 GHz would take 40,000–60,000 ns at 45 to 115 W. Ultimately, Innervator's hardware-accelerated approach bridges the inherent mismatch between current AI algorithms and the general-purpose digital hardware they run on.**

***Index Terms*—FPGA, VHDL, DNN, VLSI, neural networks, hardware acceleration, hardware compiler, Innervator, open-source.**

1. Introduction

Deep neural networks are, in their simplest form, a massive conglomerate of what could be thought of as if–else branches but with fuzzy logic as opposed to strict boolean algebra; with each neuron's output being connected to its proceeding neurons' inputs, a network "learns" to extract interesting features from its inputs, classifying them into relevant categories to feed forward into next layers. Mathematically, the task of a single neuron is to calculate the weighted sum (i.e., dot product) of its given *inputs* X by a series of associated *weights* W, add (or subtract) a *bias* term b, and finally perform an *activation function* $\alpha()$ on the intermediary output: $\alpha(W \cdot X + b)$. With W and X being column and row matrices, respectively, one-dimensional arrays could be utilized to represent both data structures.

However, as the number of layers and neurons grows, so does the length of these arrays. Similarly, with each neuron having its own unique set of weights to work with, the number of these 1-D weight arrays quickly explodes; it becomes far too large to fit in scarce CPU caches. In turn, this means that all these weights—which continue to remain static throughout inferences—will now have to reside in dynamic memory; this introduces significant overhead to matrix multiplication instructions.

This highlights the first performance advantage that FPGAs have over general-purpose CPUs: **data-tailored computation.** In an FPGA, right at the register-transfer level, one could "embed" said static weights as compile-time generics by using the VHDL `constant` keyword. Thus, when the time comes to multiply inputs by their weights, the weight parameters will already be there; the FPGA will not have to spend many extraneous clock cycles to first fetch them from (commonly dual-channel) RAM.

The second advantage is **latency predictability:** with a proper design that meets the target FPGA's timing constraints in expected operating environments, the runtime latency remains the same throughout repeated runs [1], barring supernatural events (e.g., thermal anomalies or radiation and single-event upsets) [2]. For example, if an FPGA is designed (without timing failures) to internally calculate a value in 630 ns, then it will always do so in exactly 630 ns; this will not become 620 or 630 ns, and nor will it become ±1 ns. On the other hand, due to the proprietary nature as well as sheer complexity of modern processors (e.g., branch prediction [3], cache misses [4], microcode engine updates [5], out-of-order execution, variable frequency boosting or throttling, etc.), a CPU / GPU will have an unpredictable latency: they might take 40 ms to accomplish a task, but there is no guarantee that this does not become 20 ms or 60 ms in subsequent runs, even in a "bare-metal" scenario and without an operating system's scheduling or abstraction in

the way. In real-time applications, such timing discrepancies become critically important.

Third, FPGAs are **massively parallelizable.** As an oversimplification, this can be likened to the flow of water within a pipe: given enough water (i.e., electricity), it does not really matter how many branches said pipe splits into, for they all fill at the same time. In contrast, a CPU / GPU would lack pipes and instead resort to using a number of buckets to manually fill each branch from a nearby water source. Obviously, with such rigid and preplanned pipe arrangement, a downside is that an FPGA design becomes a lot more application-specific and rigid, requiring complete overhauls to the compiled bitstream for even the slightest change to the network's numeric parameters. Fortunately, FPGA chips can be reprogrammed out "in-the-field," but a robustly trained neural network intended for edge-device deployment is likely not to require parameter readjustments after the initial programming, in the first place.

The fourth benefit lies in Innervator's own **deterministic design:** each input is uniquely mapped to only one output. The saturation, rounding, and compression of fixed-point data are well-defined and not lossy; for instance, if an Innervator-implemented network predicts that a given image contains the digit '5' with a 0.966125 probability, then it will always return 0.966125, regardless of how many times it is given the same image.

2. Methodology

Innervator operates in a simple manner: it parses a trained network's parameters and assembles a synthesizable design out of it. By assembling, I mean using VHDL's `for-generate` statements to "physically" route and place the neurons where they should be, without forcing the user to manually hard-code each and every one of them by hand, and without using another scripting language / preprocessor to literally transpile them.

2.1. Project overview

2.1.1. Nomenclature. To *innervate* means "to supply something with nerves." Innervator is, aptly, an implementer of artificial neural networks within programmable logic devices. Furthermore, these hardware-implemented neural networks could be called *innervated neural networks*, which also appears as INN in INNervator.

2.1.2. Open-source. Innervator is free hardware design dual-licensed under the GNU-LGPL-3.0 (or later) and CERN-OHL-W-2.0 licenses [6]; its source code and sample data, alongside a live FPGA video demo, is available at the following GitHub repository: https://github.com/Thraetaona/Innervator

2.1.3. Toolchain and device. To ensure maximal compatibility, Innervator has been successfully tested under *Xilinx Vivado v2024.1* (synthesis & implementation) and *Mentor Graphics ModelSim v2016.10* (simulation). The code itself was written using a subset of the **VHDL-2008** language, without any other script or preprocessor involved. Additionally, absolutely no vendor-specific libraries, nor direct instantiations, were used in Innervator's design; only the official IEEE-P1076 packages—`STD` and `IEEE`—were utilized.

Additionally, because simulation itself is an idealized scenario and there exist many classes of bugs that are only replicable in actual hardware,[1] Innervator was implemented and tested to work in a low-cost Xilinx Artix-7 35T FPGA, on a Digilent Arty A7-35T evaluation board. Given that this was a rather small (in terms of hardware resources) FPGA,[2] it was very challenging to fit all the components while keeping the overall execution latency low. To conserve on limited DSP hardware multipliers, I designed each neuron to process its $W \cdot X$ (i.e., the weighted sum) in small user-controlled segments (a.k.a. batches); for example, instead of dedicating 64 DSPs to multiply 64 pairs of numbers together in a single neuron, a more reasonable amount such as $1, 2, 4, 8, \ldots$ could be used. Similarly, given the sheer density of interconnected layers in deep neural networks,[3] a secondary challenge also existed: meeting timing constraints. As the hardware area used by Innervator's logic grew close to the target FPGA's limit, the toolchain's router started to face much more difficulty in timely marshalling signals to and from DSPs—which exist as hard-macro silicon on very specific sections of the chip—all around the FPGA in a single clock cycle; to resolve this, I introduced another user-controlled parameter: number of pipeline stages in DSP-related operations. I will elaborate more on these configurable options later in this paper.

2.1.4. Code quality. Furthermore, it is worth mentioning that all of Innervator's VHDL source files conform to the RFC 678 plaintext document format [7] and the U.S. DoD Military Standard ANSI/MIL-STD-1815A, Ada 83 Quality and Style Guide 2.1.9 ("Source Code Line Length") [8]; it is a good readability practice to limit a line of code's columns to 72 characters and avoid using non-ASCII characters therein. Also, because I was a complete novice to VHDL, AI, and FPGA design myself, I documented and commented on each step as if it was a beginner's tutorial. Writing Innervator led me to find two shortcomings within VHDL itself, too.[4]

2.1.5. Package layout. The top module (akin to the "main" file in conventional software languages) and all its dependencies are contained in the `./src` (source) directory. Additionally, the Source Directory has been fur-

1. For instance, simulation behavior of daisy-chained `variable` registers in VHDL can be vastly different from actual hardware whenever edge-activated on the clock.

2. For comparison, some high-end Xilinx Versal FPGAs can have up to 8,460,288 LUTs or 14,304 DSPs; an Artix-7 35T only has a modest 20,800 LUTs and 90 DSPs.

3. For example, one of my sample networks had 64 inputs, 20 hidden neurons, and 10 output neurons; in total, this meant $64 \times 20 \times 10 = 12{,}800$ individual connections just between the neuron blocks themselves.

4. https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/311 and https://gitlab.com/IEEE-P1076/VHDL-Issues/-/issues/312

ther divided into two sub-directories: `./src/core` and `./src/neural`. The former includes Innervator's miscellaneous infrastructure, such as a UART transceiver, button debouncer, register pipeliner, clock prescaler, and signal synchronizer. The latter, however, is where the main neural logic of Innervator lies: custom neural typing definitions, activation functions, neurons, dense layers, and the network are all defined there. Within the Neural Directory, there is also `./src/neural/utils`, which contains both the relevant mathematics functions, like $argmax()$, and the VHDL-based neural network file parser.

Although VHDL is free from the complexities of traditional programming language's build systems, it still has its own share of issues with package management. Being based on Ada, VHDL files are not viewed by the single-pass compiler as self-contained codebases; instead, declarations and statements are seen as individual "units" [9]. For example, importing a library before an `entity` declaration will only make said library usable within that specific unit's scope; it will have to be re-imported for continued usage after a subsequent declaration. For this reason, attempting to design Innervator in a modular, hierarchical way meant having to be constantly wary of accidentally creating circular dependencies. Hence, I arranged Innervator in the following order:

`config` package. This consists of the top module, `main.vhd`, and the "global import" configuration file, `config.vhd`;

`core` package. This package is standalone and comprises of the `./src/core` directory; and

`neural` package. This includes all files in `./src/neural` but note that due to buggy handling of generic packages in *Xilinx Vivado*, I had to "defer" the actual instantiation of `ieee.fixed_generic_pkg` to `./src/neural/typedefs.vhd` (instead of `./src/config.vhd`), as a workaround.

2.2. Usage guidelines

Other than being vendor-independent and portable, Innervator is also highly configurable and can be controlled via `./src/config.vhd`. All of the following compile-time settings may be configured and fine-tuned to the user's requirements: the target FPGA's clock speeed; the neural network's filesystem path; the polarity and inversion of synchronous / asynchronous resets; input ports' de-glitching stages and button debounce timeouts; number of pipeline stages and batches (i.e., DSP multipliers to use per neuron); and the baud rate of the built-in UART.

Table 1. `config.vhd` settings

Constant	*Type*	*Default*
Description		
Network Parser		
`c_DAT_PATH`	`string`	*N/A*
Absolute / relative path to the folder with network's parameter files.		
FPGA Intrinsics		
`c_CLK_FREQ`	`positive`	`100e6`
Target FPGA device's clock frequency (in hertz).		
`c_CLK_PERD`	`time`	`1 sec / 100e6`
Target clock's period; automatically calculated from `c_CLK_FREQ`.		
`c_RST_INVT`	`boolean`	`true`
Whether the Reset Signal gets inverted at the I/O region.		
`c_RST_POLE`	`std_ulogic`	`'1'`
Whether reset circuits trigger on an active-low or -high signal.		
`c_RST_SYNC`	`boolean`	`true`
Whether reset circuits are (a)synchronous in relation to main logic.		
`c_SYNC_NUM`	`natural`	`3`
No. of register stages that external input ports pass through at I/O.		
`c_DBNC_LIM`	`time`	`30 ms`
Timeout duration of debounced pushbuttons (in milliseconds).		
Fixed-Point Sizing		
`c_WORD_INTG`	`natural`	`4`
No. of bits used for the integral (i.e., whole) part.		
`c_WORD_FRAC`	`positive`	`4`
No. of bits used for the fractional.		
`c_WORD_SIZE`	`natural`	`4 + 4`
No. of total bits used for fixed-point words; automatic.		
`c_GUARD_BITS`	`natural`	`0`
No. of additional guard (i.e., rounding) bits to be used.		
`c_FIXED_ROUND`	*enum.*	`fixed_truncate`
Whether to truncate or round (when `c_GUARD_BITS` > 0).		
`c_FIXED_OFLOW`	*enum.*	`fixed_saturate`
Whether to saturate or wrap on overflows.		
Neural Configurations		
`c_BATCH_SIZE`	`positive`	`1`
No. of DSPs to concurrently use per neuron.		
`c_PIPE_STAGE`	`natural`	`3`
No. of pipeline registers (also latency in ms) inside each neuron.		
UART Settings		
`c_BIT_RATE`	`positive`	`9_600`
Baud rate (i.e., bitrate) of the built-in UART Tranceiver.		
`c_BIT_PERD`	`time`	`1 sec / 9_600`
Bit period of the UART; automatically calculated.		

2.2.1. Fixed-point numerals. Variable-width `real` data types are not synthesizable in VHDL design tools, and IEEE-754 `float32` or `float64` can be too wasteful in terms of hardware resources *vis-à-vis* the target DNN's actual precision requirement.[5] Given the overall importance of fractional numbers in neural calculations—where outputs of routinely used mathematical functions (e.g., sigmoid activation) are exclusively fractional—much thought went into designing an acccurate yet performant solution. Ultimately, I defined custom subtypes based on the `ieee.fixed_generic_pkg` formal generic package. Innervator's typing definitions file, `typedefs.vhd`, declares the `fixed_neural_pkg` package, which comprises of various types, subtypes, and aliases that are used throughout the rest of Innervator. These subtypes are constrained based on the number of bits allotted to the fractional and integral components of fixed-point values. Out of all types, `neural_bit`, `neural_word`, and `neural_dword` are among the most frequently used in Innervator. Furthermore, arrays of these values (ending in the suffix `_vector`) and nested arrays of their arrays[6] (ending in the suffix `_matrix`) are also provided. Shorthand notations (e.g., `nrl_wrd`, `nrl_vec`, and `nrl_mat`) are available, too.

`neural_bit`. Neural bits are special in the sense that they are the only unsigned neural datatype. Moreover, they have the same bit width as `neural_word` types but with the entire spectrum dedicated to the fractional component; they are "analogue-esque" with a range of $[0, 1)$ and a granularity of $2^{-(\texttt{c\_WORD\_SIZE})}$. Neurons' outputs (i.e., sigmoid range) all make use of this type.

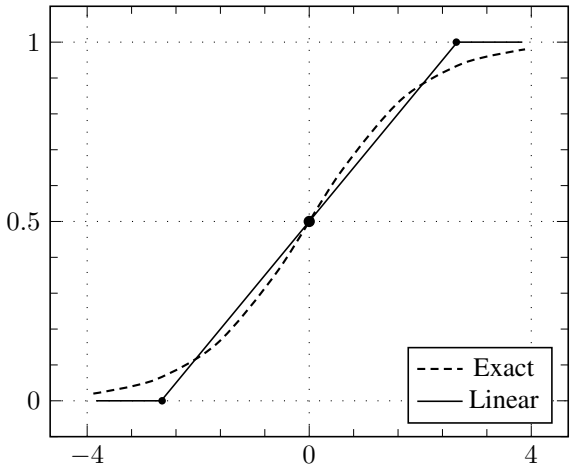
`neural_dword`. This "double-word" is twice the size of a normal `neural_word`, in terms of both the fractional and integral components. For calculations which can potentially overflow and result in loss of accuracy (e.g., weighted sum), this might be helpful. Finally, consistent with traditional naming schemes, even subtypes as large as `neural_oword` (octuple-word) or as small as `neural_nibble` (half-word) exist.

2.2.2. Activation function. The `./src/neural/activation.vhd` file defines the sigmoid activation function using fixed-point neural types. Also, given that the sigmoid function contains a division and exponentiation, implementing it in hardware would be very costly in terms of resources; so, I designed the following linearized approximation:

$$\frac{1}{1+e^{-x}} \approx 0.1875 \cdot x + 0.5 \ . \tag{1}$$

Thanks to the saturating nature of neural types, the linearized function's range does not go out of bounds; it remains ≥ 0 and < 1. Both versions have been graphed below:

Figure 1. Linearized sigmoid graph



2.2.3. Network parser. During compilation, Innervator opens the directory specified by `c_DAT_PATH`, which should contain the target network's parameters. In this folder, parameters are expected to have been provided in `biases_{i}.dat` and `weights_{i}.dat` files, where $i \in \mathbb{W}$ denotes which neural **layer** the bias and weight files belong to. Due to current mainstream VHDL synthesizers only being able to read `std_logic_vector` values from files,[7] parameters will have to be preformatted in their fixed-point representations. Inside bias files, each neuron's bias value is provided on its own line. Similarly, in weight files, each neuron's array of weight values has been "flattened," with each weight existing on its own line; furthermore, the end of each neuron should be marked with a row of `X` (logic vector value) sentinel.

Be aware that, due to a bug in *Xilinx Vivado's* `std.textio.file_open()`,[8] Innervator's file parser will have to create at most (i.e., only during the first run) a `weights_{i+1}.dat` empty file; for the purpose of counting total layers, there is no other way in VHDL-2008 to determine whether `weights_{i+1}.dat` exists without actually opening and checking if it is empty.

In fact, an especially difficult part of the project was writing `./src/neural/utils/file_parser.vhd`, and the majority of this difficulty was due to VHDL's own file I/O limitations and FPGA vendors' subpar implementations thereof. *ModelSim 2016* had no issues whatsoever with VHDL's `std.textio` library, but *Synopsys Synplify Pro 2019* and *Vivado 2024* certainly had no shortage of bugs and questionable design choices when it came to filesystem access.

5. Innervator's sample network acheived high accuracy with primarily 8-bit-wide fixed-point types.

6. Purely 2-D arrays are not yet consistently supported across vendor tools.

7. *Synopsys Synplify P-2019* is completely broken in regard to synthesis file I/O, and *Xilinx Vivado v2024* is only slightly better: https://support.xilinx.com/s/question/0D54U00008ADvMRSA1/bug-in-vhdl-textioread-overload-of-real-datatypes-size-mismatch-in-assignment

8. Contrary to VHDL's LRM, *Xilinx Vivado* wrongfully terminates synthesis whenever `file_open()` is called on a non-existant file: https://support.xilinx.com/s/question/0D54U00008CO8pTSAT/bug-fileopen-is-not-consistent-with-ieee-standards

Afterward, the network's contents are parsed into a `network_layers` array, consisting of `layer_parameters` records. These records further contain three fields: `dims` (`dimension` type), `weights` (`neural_matrix` type), and `biases` (`neural_vector` type). `neural_matrix` is a nested array of `neural_vector` arrays, which in turn consist of `neural_word` values. The `dims` (i.e., true dimensions) field has to exist because VHDL arrays can only hold one type of data (and of the same size) within themselves, and because VHDL records (which can hold different datatypes) are not iterable; accordingly, both `biases` and `weights` are "supersized" into the largest existing neural connection in the network, with unused elements existing as padding to work around VHDL's limitations.[9]

2.2.4. UART I/O. The `./src/core/uart` folder contains a UART receiver architecture, allowing runtime communication with an innervated network. The UART uses 8 bits for data, without any parity check, and its baud is configurable via `./src/config.vhd`. After starting, the network waits to receive a number of bytes, which is equal to the number of neurons in the input layer; upon receipt, the network begins its inference. As of the time of writing this paper, Innervator does not yet include a UART transmitter; the onboard LED lights are currently utilized to display the $argmax()$'ed prediction. However, note that the input data will have to be in `neural_bit` formats.

3. Results

Innervator is not restricted to any specific application, and it could compile voice-based neural networks just as well as it compiles image-based ones; to it, inputs and outputs are all mere numeric values. However, as a proof of concept, I used Innervator to implement a sample 8×8-pixel handwritten digit-recognizing neural network[10] in a low-cost AMD Xilinx Artix-7™ FPGA @ 100 MHz. Said network was structured in the following manner: 64 input neurons, 20 hidden neurons, and 10 output neurons. Power consumption and execution speed were shown to have greatly improved compared to software-emulated implementations. In the 1st layer, $(2 \times 64 \times 20) + 20 = 2580$ operations happened over 420 ns; in the 2nd layer, $(2 \times 20 \times 10) + 10 = 410$ operations took place over 210 ns. Therefore, with 3 pipeline stages and 2 batches, **the network achieved a throughput of $^{2990}/_{(420 \times 10^{-9})} \times 10^9 \approx 7.12$ GOP/s,** predicting the output in 630 ns and under 0.25 W of power. In comparison, my Intel® Core™ i7-12700H CPU @ 4.70 GHz would take 40,000–60,000 ns at 45 to 115 W.[11] Additionally, this network consumed about 67% of the FPGA's look-up tables (LUTs).

Of course, using a lower `c_BATCH_SIZE` will result in less resource utilization, at the cost of runtime speed (e.g., 630 ns vs. 1030 ns). Also, with slower clocks, such as 50 MHz ones, a smaller `c_PIPE_STAGE` could be used for latency improvements and less register usage.

Shown below are the FPGA's resource utilization (specifically for the neural engine itself) after one- and two-batched implementations of the same network:[12]

Table 2. Resource utilization on an Artix-7 35T with batches of 1 and 2

Resource	Utilization (1)	Utilization (2)	Availability
Logic LUT	10,233	13,949	20,800
Sliced Reg.	13,954	22,145	41,600
F7 Mux.	620	1,440	16,300
Slice	3,775	6,115	8,150
DSP	30	60	90
Speed (ns)	1030	630	*N/A*

In most cases, the accuracy of this proof-of-concept network had also remained within an acceptable margin of its software-based counterpart that used floating-point numbers:

Table 3. Accuracy of the innervated network

Digit	Prediction	
	FPGA	CPU
0	.30468800	.10168505
1	.57812500	.15610851
2	.50781300	.14220775
3	.21875000	.19579356
4	.00390625	.00119471
5	.20703100	.01840737
6	.21484400	.00273704
7	.13281300	.09511474
8	.24218800	.15363488
9	.69921900	.71728650
Speed (ns)	630	40k–60k

The exact image used in the above-mentioned run was image no. 1759 from the MNIST-8 dataset:

9. Although the Language should have support for dynamically allocated arrays (outside of synthesis), *Xilinx Vivado* has been very unstable in handling that. However, Innervator does specify pointer types (e.g., `neural_word_ptr`) for future use.

10. In order to learn more about machine learning in general, I also wrote and used my own network-training library in Python; its source code is available at this repository: https://github.com/Thraetaona/Innervate

11. According to Intel Corporation, this CPU could exceed even its maximum power draw in short bursts ($\leq$ 10 ms).

12. It is important to note that placement and routing is an inherently random task; as long as results are considered "good enough" by the tool, the utilization and performance can slightly vary between each and every synthesization run, even without any modifications to the source code.

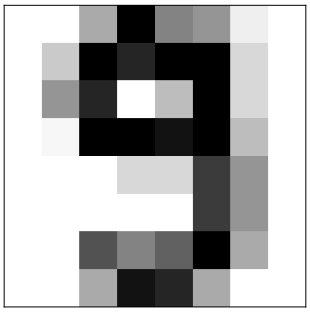


Figure 2. The tested digit

4. Discussion

By default, all inputs and weights are 8-bit-wide and the internal accumulator is twice that (16), ensuring that the entire multiply-add calculation can fit in just one DSP per batch per neuron. While one can configure the bit widths in `config.vhd`, it is better to just pretrain the network to work in reasonable ranges and precision in the first place.

The architecture is not completely unrolled. For example, for a hypothetical neuron that has 64 inputs, if the number of batches is equal to 4, then 64 input / weight pairs get unrolled into batches of 4 across 16 *iterations*; had the batch been only 1, they would have gotten unrolled into single calculations across 64 iterations. However, the layers and neurons themselves are unrolled as-is: if you have 100 of them, all 100 will "physically" exist in hardware. This logic size–speed trade-off would allow for pipelining, similar to a car assembly line, meaning that the next input would not have to "wait" the entire duration of a full network prediction before it starts to get processed. If only a single neuron was reused per layer—or even across all layers—the FPGA would require many more clock cycles and quite a lot of memory to store the intermediate output of each virtual layer for the proceeding one.

Finally, there were three main areas that I did not explore in my research:

1) As an accuracy–speed trade-off, neural networks could be trained or post-processed to use only integers (i.e., no fractions) in their calculations, bypassing the slight resource inefficiency that fixed-point numerals have;
2) FPGAs "emulate" RTL designs with look-up tables, making hard-silicone ASICs more efficient in accelerating neural networks, although they are also costlier and harder to design; and
3) Beyond ASICs, analogue hardware would be far better (in terms of power consumption, speed, and space) for neural networks.

Regarding analogue hardware, VHDL-AMS (analogue mixed-signal extensions to VHDL-2002) might be helpful, though I have not yet found a consumer-accessible simulator for VHDL-AMS.

5. Conclusion

Admittedly, FPGAs might not be very well-suited for "real-world" applications of neural networks. For small networks, such as the one described in this research, Innervator is indeed very efficient but almost overly so: this has been akin to using a crane to drive a nail into drywall. However, for large networks (e.g., *Meta LLaMA*), FPGAs simply do not have enough "room" (i.e., logic resources); even to combat the lack of space, one can only pipeline or segmentize the design so much before the sheer latency causes the FPGA to perform just as slow as a CPU / GPU. Also, high-end FPGAs with more space can quickly become incredibly costly.[13] Finally, there has been comparatively little research and development done on FPGA-based neural networks, making designing and getting started with traditional CPU / GPU workflows much easier and far less time-consuming.

6. Acknowledgments

Innervator had been written from scratch in an independent manner, but I would like to use this opportunity to thank *Dr. Dave Rosoff* at *the College of Idaho* for allowing me to define it as my own independent study course in order to earn academic credit-hours for my out-of-college work, despite my first-term freshman status. Also, Innervator was presented at the College's *19th Annual Student Research Conference*, on May 2, 2024.

References

[1] M. Mac and C. Wysocki, "Guaranteeing silicon performance with FPGA timing models," *EE Journal*, 2009. [Online]. Available: https://www.eejournal.com/article/2010081901-altera

[2] AMD, "RT Kintex UltraScale FPGAs for ultra high throughput and high bandwidth applications," 2020. [Online]. Available: https://docs.amd.com/v/u/en-US/wp523-xqrku060

[3] J. Engblom, "Analysis of the execution time unpredictability caused by dynamic branch prediction," in *The 9th IEEE Real-Time and Embedded Technology and Applications Symposium, 2003. Proceedings.*, 2003, pp. 152–159.

[4] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *2007 5th IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, 2007, pp. 143–148.

[5] Intel Corporation, "Mitigations for jump conditional code erratum," 2020. [Online]. Available: https://www.intel.com/content/dam/support/us/en/documents/processors/mitigations-jump-conditional-code-erratum.pdf

[6] F. Memarzanjany, "Innervator," Jun. 2024. [Online]. Available: https://doi.org/10.5281/zenodo.12712831

[7] "Standard file formats," RFC 678, Dec. 1974. [Online]. Available: https://www.rfc-editor.org/info/rfc678

[8] "ANSI/MIL-STD-1815A: military standard Ada programming language," United States Department of Defense, Feb. 1983. [Online]. Available: https://quicksearch.dla.mil/qsDocDetails.aspx?ident_number=37152

[9] P. Faes, "'use' and 'library' in VHDL," Sep. 2013. [Online]. Available: https://insights.sigasi.com/tech/use-and-library-vhdl

13. For example, as of the time of writing, a VCK190 FPGA board can cost about $13,000; an NVIDIA® TITAN RTX™ costs about $1,300.