Deep Learning Hardware Language Conversion Using HLS for the ATLAS Sensor

Neural Networks on FPGA for ATLAS Detector Trigger System

Angus Li, Justin Tandjung, Ryan Tjoe, Avery Wong

Project 2512:
Engineering Physics 459
Engineering Physics Project Laboratory
The University of British Columbia

Project Sponsor:

Dr Maximilian Swiatlowski, Research Scientist, TRIUMF

Dr Wojciech Fedorko, Data Scientist / Deputy Dept. Head Scientific Computing, TRIUMF

Executive Summary

This report focuses on the implementation of a DeepSets neural network on field programmable gate arrays (FPGA) through the use of AMD’s Vitis High-Level Synthesis (HLS) and Vivado software suites. The project aims to improve the collision energy reconstruction algorithm within the ATLAS particle detector’s data triggering system.

To improve collision energy reconstruction accuracy, TRIUMF has developed a neural network to complement the current algorithm and has sponsored this project to investigate the feasibility of implementing the neural network on the FPGA-based triggering system. To run the neural network on an FPGA, Vitis HLS is used to convert a C++ representation of the neural network into FPGA-compatible HDL code. Vivado is used to implement the converted code on a physical development board. The results of this project are evaluated according to resource usage and timing constraints set by TRIUMF. While the design does not yet meet the specified constraints, there are avenues of exploration through which the design can be improved. The design is successful in implementing the neural network on FPGA and currently serves as a proof of concept. Future work is planned to focus on optimizing the Vitis HLS conversion according to the specified constraints and on building supporting modules to complete a full data pipeline that can be directly integrated into the ATLAS FPGA system.

Table of Contents

Table of Figures

Introduction

ATLAS

TRIUMF is collaborating with CERN, the world's leading particle physics institute, on the ATLAS project. ATLAS is the world's largest particle detector built for the Large Hadron Collider (LHC), the world's largest particle collider. By discovering and analysing new fundamental particles produced by the LHC's particle collisions, ATLAS has been contributing to the research of modern particle physics.

Fundamental particles are not produced in every collision so to effectively study them, the LHC produces roughly 1 billion collision events every second, leading to approximately 60 terabytes of observation data per second from ATLAS.

Since storing this amount of data is impractical and only a fraction of collisions contain new or interesting information, a real-time filtering system known as a "trigger" is implemented in ATLAS. Based on predetermined criteria, the two-stage trigger selects only 1000 events per 40 million to be stored for further analysis. The first stage hardware trigger performs initial filtering to 100,000 events and passes them to the second stage software trigger as shown in Figure 1. Through this process, ATLAS produces a large but manageable amount of data for further study by scientists.

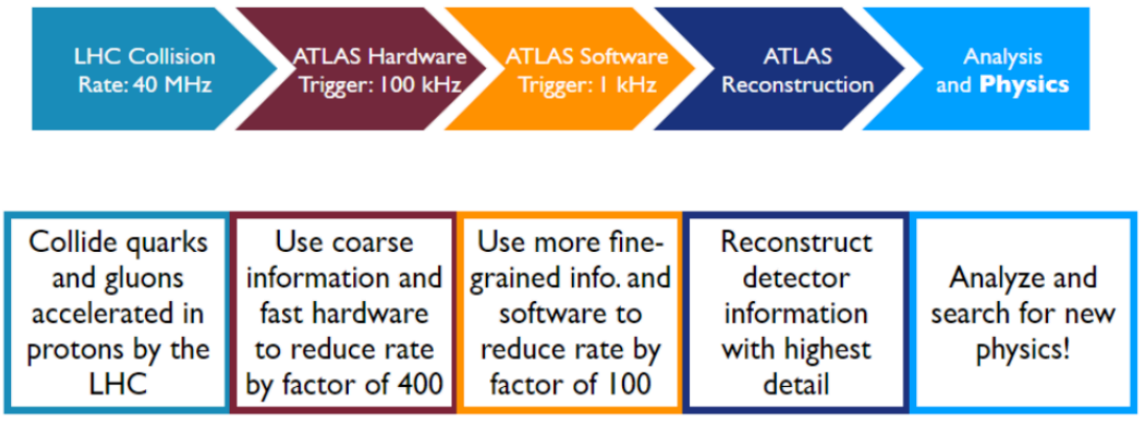

LHC Collision Rate: 40 MHz
ATLAS Hardware Trigger: 100 kHz
ATLAS Software Trigger: 1 kHz
ATLAS Reconstruction
Analysis and **Physics**
Collide quarks and gluons accelerated in protons by the LHC
Use coarse information and fast hardware to reduce rate by factor of 400
Use more fine-grained info. and software to reduce rate by factor of 100
Reconstruct detector information with highest detail
Analyze and search for new physics!

Figure 1: Trigger System for ATLAS

Improving the Triggering System

ATLAS and the LHC began operations in late 2009 and have since undergone upgrades to increase collision rate, detect collision energy accurately, and improve trigger hardware and software. Currently, ATLAS scientists face one major problem. In less than 2 decades of operation, the event data produced by ATLAS is mostly based on well understood particles and interactions. Now, they aim to look for collision events which yield data on less understood particles and interactions. One method to accomplish this is to improve the trigger system so that it selects these “interesting” events more often. To this end, TRIUMF aims to improve the collision energy reconstruction algorithm in the hardware trigger by replacing it with a neural network of equal or better performance.

The objective for this project is to investigate the feasibility and performance of an FPGA implementation of a TRIUMF-developed neural network for possible integration into the ATLAS hardware trigger. By developing a deep understanding of the neural network’s architecture, the aim is to leverage AMD’s Vitis HLS and Vivado software suites to translate the neural network into functional FPGA code.

The scope of the project focuses on translating the TRIUMF-developed neural network into functional FPGA code using Vitis and Vivado. The scope also includes building supporting code modules to realize a full data pipeline and interface system that can be directly integrated into the ATLAS hardware trigger. For ENPH 459, the scope was limited to building a proof-of-concept for the successful translation of the neural network to FPGA code. This includes learning to use Vitis and Vivado, testing with the development board provided by TRIUMF, and as a stretch goal, optimizing the translation process to produce better FPGA code. For ENPH 479, the scope will be expanded to building the parts of the system other than the neural network modules as shown in Figure 3. Future scope may also include optimization of the neural network itself depending on feasibility concerns.

Discussion

Neural Network Theory

The TRIUMF-developed neural network is based on the DeepSets architecture developed in Ref. [1]. The advantage of the DeepSets architecture over other architectures is that it can handle sets of data that are variable in length and that do not depend on the order of contained elements. These properties led to its application in studying particle collision events as described in Ref. [2].

The structure of the TRIUMF-developed DeepSets network consists of an arbitrary number of identical multi-layer perceptrons (referred to as ϕ (phi)) equal to the number of elements in the set to be analyzed, a summation step for the ϕ outputs, and one final multi-layer perceptron (referred to as *F*). This is shown in Figure 2 below.

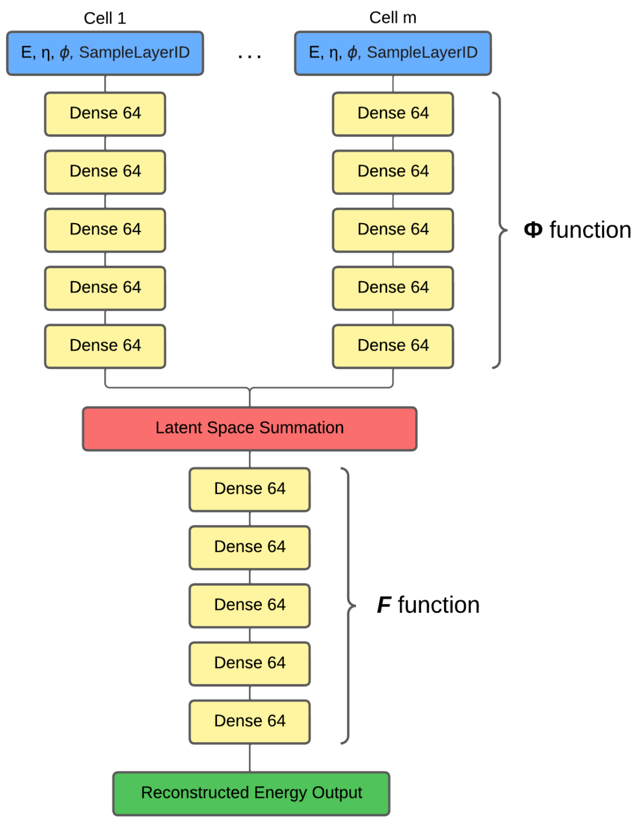


Figure 2: TRIUMF DeepSets Neural Network

The ϕ and *F* function can decomposed into dense layers that can be reconstructed in handwritten code as an input transformed by a matrix multiplication, bias vector addition, and activation function. This decomposition serves as the starting point for translating the neural network into FPGA code.

Tools

FPGA

Neural networks are typically run on GPU architecture due to computational intensity. However, due to the parallel nature of the Phi networks, and the simplicity of the Phi and *F* networks themselves, implementation onto an FPGA offers significant upsides in terms of speed. This project uses the PYNQ2 board, which features a system AMD ZYNQ on chip design. The PYNQ is able to communicate with the on-chip CPU using Python to send signals to the FPGA. The actual hardware system will target a higher end AMD Versal-1802 FPGA, but the PYNQ serves as a prototyping tool for developing the network.

FPGA Resources

The primary resources on an FPGA board are flip-flops (FFs), lookup tables (LUTs), digital signal processing logic elements (DSPs) and memory (BRAM). On FPGAs, FFs and LUTs are typically abundant while DSPs and BRAMs are more of a limited resource and are the primary concerns regarding FPGA resource usage.

BRAM is a type of FPGA memory which can store large quantities of data and can be configured to have varying memory widths and depths. DSPs are a type of hardware resource that are optimized for multiplication and addition operations and can be utilised by the FPGA for such operations.

HLS

High Level Synthesis (HLS) is a relatively new tool for hardware design. It allows users to design a function in C or C++ and convert that to a hardware description language (HDL) intellectual property core (IP). This results in a black-box logic cell that can be used by an FPGA. Using HLS has three main advantages: it can translate functions that are simple in software, but complex in hardware, has a faster iteration and design cycle, and a lower technical knowledge barrier. As the neural network primarily consists of matrix multiplications, it is much easier to design and quickly iterate on an algorithm in software than hardware with the assistance of HLS. Vitis HLS is AMD’s HLS tool and was chosen because of license availability and anticipation of integration with the AMD Versal-1802 FPGA targeted by the ATLAS collaboration. The tool allows for C and register transfer language (RTL) co-simulation which can verify timing and functionality without needing a physical FPGA. Vitis version 2024.2 was used for this project.

Vivado

To transform the neural network IP generated by Vitis HLS into FPGA-level instructions, AMD’s Vivado tool was used. Vivado is an FPGA bitstream synthesizer software which was used to configure and interface the neural network IP with the memory and on-chip CPU. Also included is a hardware manager which can view signals on a running FPGA. Vivado version 2024.2 was used for this project.

Design

With knowledge of the dataflow through the hardware system, the neural network to be implemented onto the hardware, as well as the tools that will be used to implement the network onto hardware, it is now possible to outline the design of the project. This segment is broken down into two sections; a hardware section that outlines how the algorithm should be implemented in hardware components, as well as a software section that outlines the design of the C++ algorithm to guide HLS into synthesizing the targeted hardware design.

Hardware

The final FPGA implementation will be subject to maximum resource and timing constraints as shown in Table 1. These constraints are not finalized and subject to change.

FF	LUT	DSP	BRAM	URAM	Latency
80,000	40,000	400	0	10,000,000	2.5μs

Table 1: Maximum Resource Usage and Latency

Figure 3 shows the system diagram for dataflow to the hardware trigger system, specifically to the hardware neural network contained within the General Event Processor (GEP) layer. In this diagram, the sensors and MUX layer have been developed by another team and are only shown to understand dataflow. The hardware algorithm in the GEP layer is the focus of this project and is only one of many algorithms (others not shown) that will be implemented in the hardware trigger. The focus of the past 4 months has been on developing and testing the ϕ (phi) network portion of the neural network.

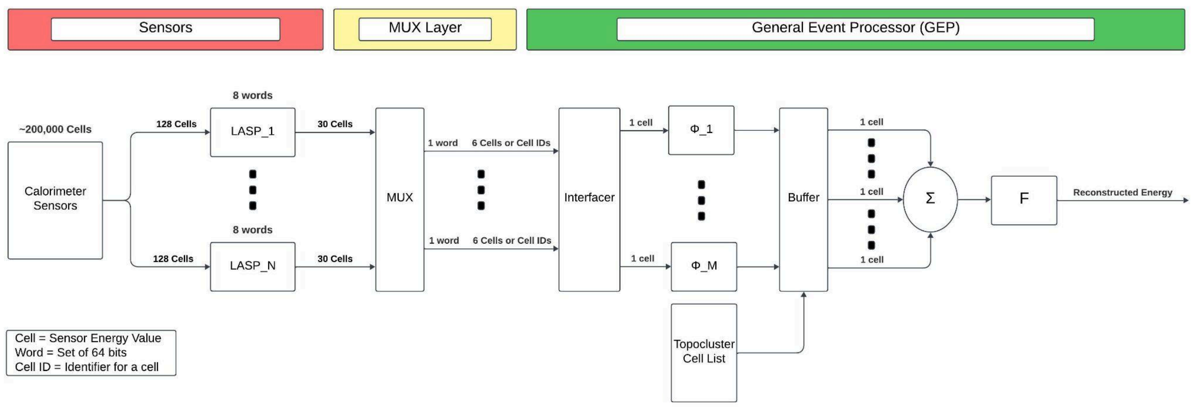


Figure 3: System Diagram

As shown in the DeepSet introduction, the ϕ network consists of 5 dense layers in series, taking a 4×1 input and calculating a 64×1 output, with values being 16-bits. Each dense layer consists of a matrix multiplication and a ReLU activation function. To increase speed throughput, the multiplication operations of the row-column dot products will be parallelized using multiple DSPs. Each operation utilises 64 weights pulled from external memory. Each element in the row-column dot product is then added together before applying the ReLU activation function.

A further discussion of BRAM setup and utilization can be found in Appendix A.

Software

Vitis HLS determines the conversion between C++ code to RTL via pragmas and user-specified configurations. Understanding which pragmas to use and how they work is crucial to achieve the desired hardware implementation.

To perform matrix multiplications efficiently in hardware, certain chunks of data must be accessed at the same time to compute row-column dot products. The matrix elements were prototyped with the 16-bit C++ short datatype, but the 16-bit ap_fixed datatype will likely need to be used to achieve correct multiplications. Notably, ap_fixed datatype increases the latency of the matrix multiplication by a factor of two, likely due to needing an extra clock cycle to perform bit shifting.

In software, matrix multiplication is typically achieved by storing data in arrays and iterating over the desired elements. Figure 4 illustrates which components of the matrix multiplication are being used for a single operation. HLS converts arrays to 2 types of storage depending on size: BRAM or shift registers [3]. Every value in an array can be read in a single clock cycle in shift register storage. BRAM storage only allows for a single entry read per cycle, but has a much higher capacity for data. Due to the size of the weight matrices, HLS defaults to storing the weights in BRAM. *array_partition* and *array_reshape* are two pragma options that allow for parallel access to the weight matrix in BRAM.

$$\underline{\underline{[i_0, i_1, \ldots, i_{63}]}} \begin{bmatrix} w_0 & w_1 & \cdots & w_{63} \\ w_{64} & w_{65} & \cdots & w_{127} \\ \vdots & \vdots & \ddots & \vdots \\ w_{4032} & w_{4033} & \cdots & w_{4095} \end{bmatrix}$$

Figure 4: Matrix Multiplication with Parallelization

array_partition splits an existing array into separate BRAM blocks, allowing for parallel access between the blocks [3]. To access 64 elements at a time, this requires a minimum utilization of 64 BRAMs per weight matrix, which is not feasible considering resource constraints. This also wastes BRAM resources, as each BRAM block would be utilizing very little of the minimum allowed depth. Furthermore, it was found that the performance did not reach a reasonable latency after experimenting with combinations of partitionings. This led to exploration of the *array_reshape* pragma.

array_reshape concatenates specific entries of the array to reduce storage depth and increase storage width [3]. In most cases, it causes HLS to infer the arrays as shift registers instead of BRAM blocks. Either the cyclic or block method and a reshaping factor are selected to direct HLS to determine entries to concatenate. In the ϕ network, the weights array is reshaped using a block factor of 64. This combines entries 0 to 63 of the weights array, which represents an entire row of the weights matrix. For efficient computation, the elements need to span the columns of the matrix. This reshaping allows HLS to fit the weights matrix into a shift register, so that entire columns can be read in a single clock cycle.

Finally, to compute the row-column dot product, the entire row must be used in parallel as well. To access the inputs, intermediate inputs, or biases in parallel, a complete reshaping is used. This puts every element of the desired array into an individual shift register, allowing for fully parallel access.

To achieve the parallelized row-column dot product as described in the hardware design, the *pipeline* and the *unroll* (implicitly) pragmas are used. Pipelining and unrolling are common FPGA throughput optimization strategies. Pipelining allows new inputs to be accepted in less clock cycles by using registers to store values in between operations and unrolling creates duplicate hardware to run the operation in parallel. A visual representation of pipelining is shown below in Figure 5 [3].

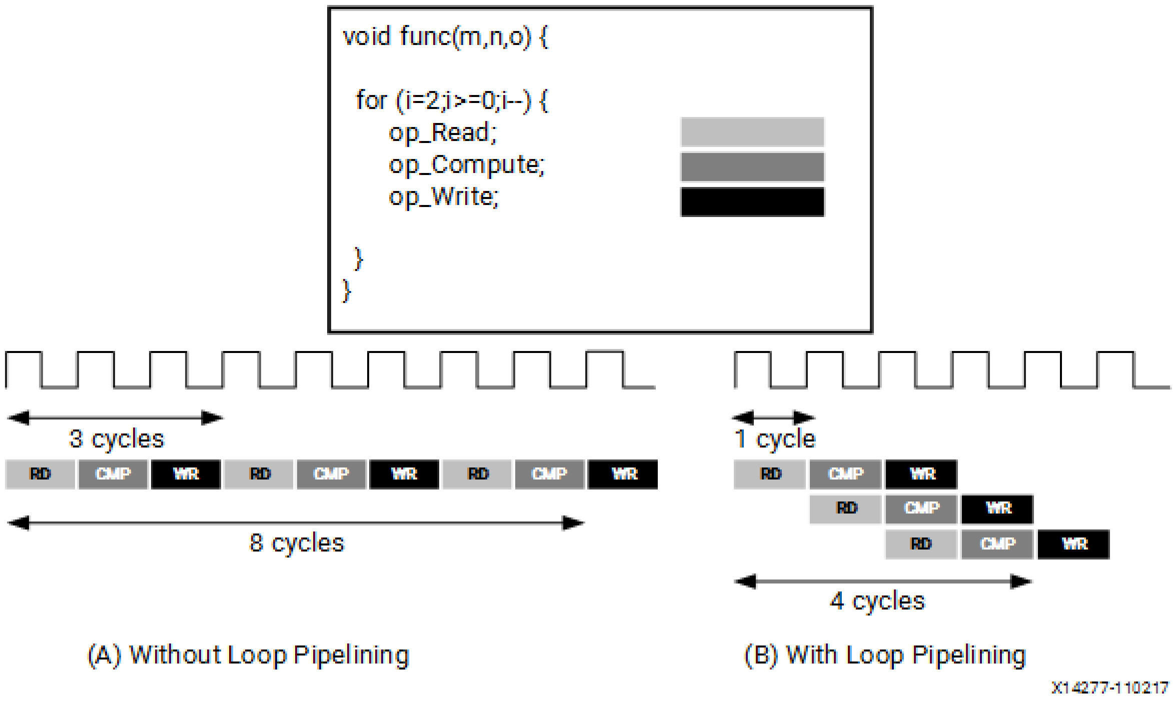


Figure 5: Pipelining Visualization

One nuance of the *pipeline* pragma involves the pipelining of nested for loops. If pipelining is applied to the top loop, *unroll* is automatically applied to the inner loop with no option to turn this off [3]. To implement matrix multiplication, pipelining can be applied to the top loop which lets a new row-column dot product happen every clock cycle. The inner loop is then automatically unrolled, which performs all 64 multiplication operations of the row-column dot product. The inner loop also contains the addition and ReLU operations which can be done within the timing of a clock cycle and do not need a clock edge to compute, resulting in a complete iteration of the top loop in a single clock cycle.

Dataflow is a pragma that was explored but unable to get functioning for the ϕ network conversion. This pragma attempts to parallelize function calls by allowing modules to work in parallel with each other as described by the diagrams in Figure 6 [3].

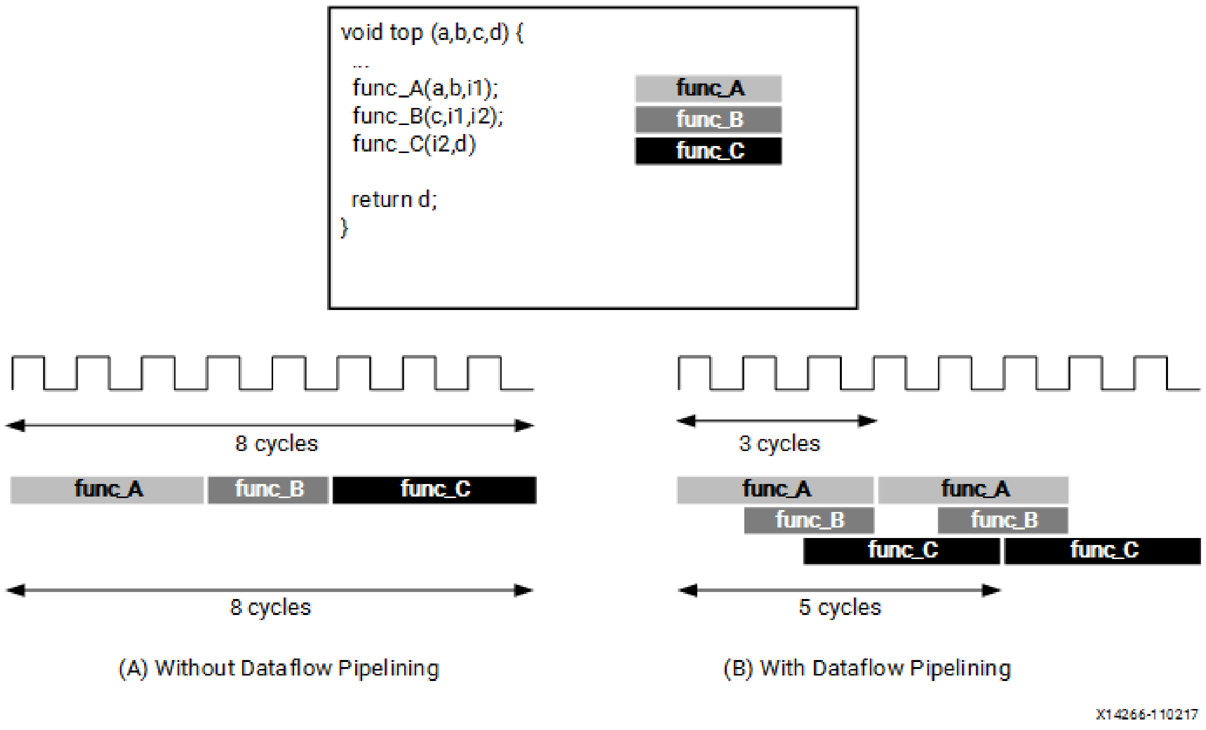


Figure 6: Dataflow Pragma Visualization

The aim was to use the pragma to parallelize function calls to the dense layers, but HLS was unable to find a parallel optimization. This was because of the dense layer algorithm and the connection between layers. The matrix multiplication in the dense layer used the entire input array vector and performed the dot product with the corresponding weights. Since the entire input array was needed to start computation under this algorithm, the entire output array of the previous dense layer needed to be completed for the following dense layer array to start. The *dataflow* pragma respected this code structure, which meant that parallel computations could not be done. However, the *dataflow* pragma did allow the dense layers to read the output of the previous layer as it was computed, reducing some latency.

Another aspect that influences the HLS conversion to RTL is the Vitis configuration file. A brief overview of important settings can be found in Appendix B.

Tests and Results

Testbenches

A key feature of Vitis is the dual C/RTL co-simulation. By writing and running C++ testbenches for the top-level function, Vitis simulates the hardware performances of the converted HDL code. Simulated waveforms show function timing and internal signal values, which are crucial for informing software iteration to reach desired hardware performance. The co-simulation is also much faster than loading the HDL code onto the physical board, and running simulation through Vivado.

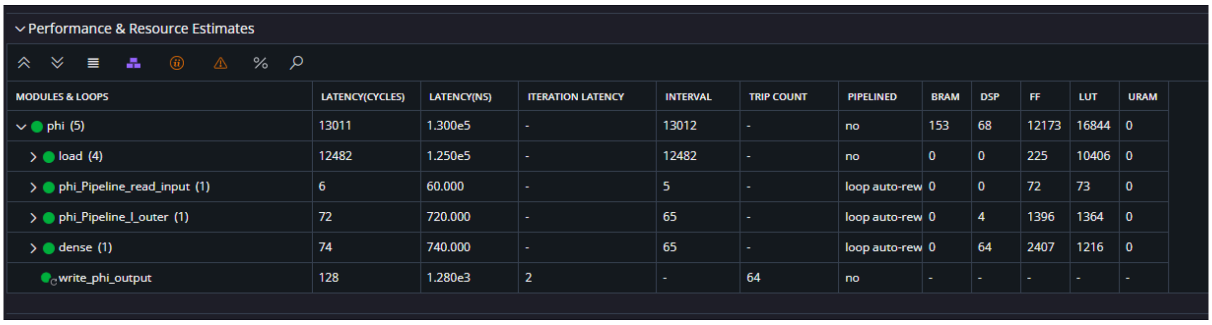
The software testbenches were verified by comparing the outputs to those of the neural network on Python. This confirmed that the hand-written neural network was computationally correct.

Results

The following results are report tables from Vitis HLS and Vivado, showing the implemented ϕ (phi) function. This ϕ function is the implementation described in the design section and does not implement any further algorithm optimizations. Notably, the ϕ function has a total latency of 529 clock cycles (ignoring the time to load the weights and biases into the module as this will be done prior to calculation)**.** The phi_pipeline row describes the first dense layer, and the dense (1) row describes the four other dense layers. Dense (1) is only synthesized as a single component and is simply run four times over.

HLS Reports

Once Vitis has converted the C++ to HDL, it generates a synthesis report on the expected timing and resource usage. This feature is used to inform decisions on changing pragmas, as well as indicate whether or not the software was converted as expected. An example of the HLS synthesis report is shown below in Figure 7.



Performance & Resource Estimates

MODULES & LOOPS	LATENCY(CYCLES)	LATENCY(NS)	ITERATION LATENCY	INTERVAL	TRIP COUNT	PIPELINED	BRAM	DSP	FF	LUT	URAM
phi (5)	13011	1.300e5	-	13012	-	no	153	68	12173	16844	0
load (4)	12482	1.250e5	-	12482	-	no	0	0	225	10406	0
phi_Pipeline_read_input (1)	6	60.000	-	5	-	loop auto-rew	0	0	72	73	0
phi_Pipeline_l_outer (1)	72	720.000	-	65	-	loop auto-rew	0	4	1396	1364	0
dense (1)	74	740.000	-	65	-	loop auto-rew	0	64	2407	1216	0
write_phi_output	128	1.280e3	2	-	64	no	-	-	-	-	-

Figure 7: Vitis HLS C++ Synthesis Report

This table is generated every time the code is synthesized, which allows for rapid iteration.

Hardware Reports

Table 2 shows the resource usage numbers on the actual FPGA from Vivado for the ϕ network IP. Also shown are the resource usage predictions from Vitis HLS as well as the total resources available on the PYNQ-Z2 board. The major difference between the two resource usage reports from Vitis HLS and Vivado lies in the BRAM usage: Vitis HLS predicts significantly more BRAM blocks will be used. This is due to Vivado inferring some BRAMs within the HLS-generated IP to be registers based on the data width and length. The DSP usage predictions ended up being identical, but it is unclear whether this is consistent.

	HLS Report (ϕ)	Vivado (ϕ)	PYNQ-Z2 Limit	% of PYNQ Limit
LUTs	16844	15447	53200	29.03%
FFs	12173	12259	106400	11.52%
BRAMs	153	65	140	46.42%
DSPs	68	68	220	30.91%

Table 2: Resource Usage Reports for ϕ Network IP

Table 3 shows the total resource usage on the board for the PYNQ verification including all elements external to the ϕ network IP. The additional flip-flops are mainly used by external BRAM controllers and additional BRAMs are used to externally store the weights and biases. Currently, the whole system fits comfortably within the PYNQ-Z2's resource limitations. If future systems aim to include the latent space summation and the *F* network, there will likely be issues caused by BRAM availability. It is recommended that the BRAM usage of the current system be reduced before implementation of the *F* network in the next phase of the project if a larger development board is not available.

	Vivado (Total)	PYNQ-Z2 Limit	% of PYNQ Limit
LUTs	17133	53200	32.20%
FFs	14061	106400	13.21%
BRAMs	95	140	67.85%
DSPs	68	220	30.91%

Table 3: Resource Usage Reports for Full ϕ Network System

PYNQ Verification

The PYNQ development board was used for verification to physically run the hardware. The PYNQ platform includes a Jupyter environment for scripting in Python which can be used to send signals, allocate memory, and read memory on the FPGA. In the finalized system, most of these signals should be sent from the interface which receives signals from other systems on the FPGA. For ease of development, it is recommended to continue using the Jupyter environment for communication with the FPGA.

AXI, or Advanced eXtensible Interface is a standard for buses used by Vivado IPs to communicate between each other. Many library IPs within Vivado 2024.2 are designed to use the AXI standard, such as the processing system for the development board, BRAM, and BRAM controllers. Since there are many signals in the AXI buses, avoiding the use of AXI for simpler tasks such as writing and reading to memory was considered. Due to the scale of the project, it was determined that writing new custom IPs for every component is currently unfeasible and using the AXI standard is recommended moving forward.

Integrated Logic Analyzer (ILA) cores are used to view signals within the FPGA fabric using Vivado Hardware Manager. They are useful tools for debugging and for verifying functionality on the physical board. Only signals made external to the HLS-generated IPs can be viewed easily, and ILA cores should be removed from the final design since they utilize BRAM.

Using the Jupyter notebook functionality of the PYNQ-Z2, it was possible to verify the outputs of the system based on the inputs. Weights and biases were loaded onto the BRAMs via the onboard CPU, and final outputs were written to an allocated memory space by the system. ILA cores were used to debug and verify behavior of the system by viewing AXI and BRAM interfaces.

Conclusion and Recommendations

With this project, the relationship between the various Vitis HLS C++ pragmas and our metrics of timing and resource usage is better understood. Specifically, *array_partition* and *array_reshape* can be used to increase throughput by allowing parallel memory access at the cost of memory resources, with *array_reshape* using less resources while having stricter limitations on amount of memory access. Additionally, *pipeline* and *unroll* pragmas can be used to also increase throughput by creating hardware that utilizes parallel computation, but uses more resources.

Since HLS-generated HDL code has been loaded onto the PYNQ development board with the correct functionality, it is clearly possible to implement the DeepSet model onto an FPGA using HLS. However, the current estimates for timing and resource usage are far from the metrics initially discussed at the beginning of the term. Even though these metrics were rough estimates and subject to change, our findings point us to believe that the goals are not feasible to achieve with Vitis HLS or even if the HDL were to be implemented by hand.

Future work should explore the parallelization of dense layer calculations and row-column dot products as outlined in the design section. These optimizations are conceptually sound but remain difficult to implement in Vitis HLS. FPGA resource and throughput constraints for this algorithm in regards to the greater hardware trigger system should also be explored and finalized. The current plans to optimize the ϕ network are discussed in Appendix C.

Deliverables

The deliverables include all code uploaded to a TRIUMF gitlab repo, a readme detailing how to run and synthesize the code in the repo, an amended copy of this report, and a copy of the slide decks prepared for and shared with TRIUMF during weekly meetings.

Appendices

Appendix A - Hardware Setup

BRAM Specification

BRAMs can be set up as read-only or read-write, as well as single and dual port. Read-only BRAMs must be initialized with a COE or memory initialization file provided before the synthesis and implementation processes. The choice of BRAM type should be based on how often the weights and biases will be changed. If often, a dual-port BRAM should be chosen so weights can be updated via the processor and controller. Otherwise, a single port read-only BRAM may be sufficient.

In the current system iteration, the neural network weights and biases are loaded into dual-port BRAMs using the CPU. The neural network IP directly reads these weights and biases from the external BRAMs into internal BRAMs initially and reads from the internal BRAMs for the calculations. The next iteration aims to bypass the internal BRAMs and read the weights and biases directly from the external BRAMs.

In hardware, storage is defined as having a width and a depth, corresponding to the number of bits in each entry, and the number of entries. For example, the standard weight matrix for the dense layers has a width of 16 and a depth of 4096.

Appendix B - HLS Setup

HLS Configuration file

Aside from the HLS pragmas, the HLS configuration file can influence how the C++ code is synthesized. During exploration, changed configurations include *compile.pipeline_loops* which was set to zero to remove auto-pipelining and *array_partition.throughput_driven* which was set to *off* to disable auto-partitioning of arrays. In addition, to make the most use of the C/RTL co-simulation, the *cosim.trace_level* setting should be set to all, and the *cosim.wave_debug* setting should be enabled. This opens the Vivado waveform viewer with the most detailed information for debugging.

Appendix C - Algorithm Optimizations

Parallelizing Dense Layers

Since the dataflow pragma did not work for this function, other ways of improving latency were explored. The first algorithm discussed involves parallelizing the dense layers themselves so HLS infers similar behaviour to the dataflow pragma.

Since the input to every intermediate dense layer is the output from the previous, the goal is to compute as far down the chain of dense layers as possible, for every output. This ends up being only a single dense layer further, which should still lead to an improved latency.

As shown in the following figure, once the first row-column dot product is computed, we get a single element of the output vector, o_0. With this, a part of each entry in the next output is computed by performing a transposed matrix multiplication with only a single entry. Everytime a new o_n is computed, another one of these multiplications can be done, resulting in 64 partial outputs. By summing these together, the output of the next layer is computed one at a time, with w^2_n representing the entries of the second weights matrix.

$$\begin{bmatrix} i_0 & i_1 & \cdots & i_{63} \end{bmatrix} \begin{bmatrix} w_0 & w_1 & \cdots & w_{63} \\ w_{64} & w_{65} & \cdots & w_{127} \\ \vdots & \vdots & \ddots & \vdots \\ w_{4032} & w_{4033} & \cdots & w_{4095} \end{bmatrix} = \begin{bmatrix} i_0w_0 + i_1w_{64} + \cdots + i_{63}w_{4032} & 0 & \cdots & 0 \end{bmatrix}$$

$$o_0 = i_0w_0 + i_1w_{63} + \cdots + i_{63}w_{4032}$$

$$\begin{bmatrix} w^2_0 & w^2_{64} & \cdots & w^2_{4032} \\ w^2_1 & w^2_{65} & \cdots & w^2_{4033} \\ \vdots & \vdots & \ddots & \vdots \\ w^2_{63} & w^2_{127} & \cdots & w^2_{4095} \end{bmatrix} \begin{bmatrix} o_0 \\ 0 \\ \vdots \\ 0 \end{bmatrix} = \begin{bmatrix} o_0w^2_0 \\ o_0w^2_1 \\ \vdots \\ o_0w^2_{63} \end{bmatrix}$$

The computation of the next output cannot be started until the incomplete layer is finished summing since the activation ReLU function must be applied on each entry. This brings the total latency from 5 layers to 3 layers and results in a 40% decrease in latency. However, this causes DSP usage to multiply which may make this approach infeasible for implementation.

Parallelizing row-column dot product

Another potential throughput optimization involves parallelizing the row-column dot product operations by running multiple iterations of the top for loop. This will use more DSPs and will require further consideration of weight and biases shaping/partitioning, but may lead to a proportional throughput increase.

References

[1] M. Zaheer, S. Kottur, S. Ravanbakhsh, B. Poczos, R. Salakhutdinov, and A. Smola, “Deep Sets,” Apr. 2018, Accessed: Apr. 08, 2025. [Online]. Available: https://arxiv.org/abs/1703.06114

[2] P. T. Komiske, E. M. Metodiev, and J. Thaler, “Energy Flow Networks: Deep Sets for Particle Jets,” *Journal of High Energy Physics*, vol. 2019, no. 1, Jan. 2019, doi: https://doi.org/10.1007/jhep01(2019)121.

[3] “AMD Technical Information Portal,” *Amd.com*, 2025. https://docs.amd.com/r/2024.2-English/ug1399-vitis-hls/ (accessed Apr. 08, 2025).