Graph Neural Network Inference on FPGA

SEPTEMBER 2019

AUTHOR(S):
Kazi Ahmed Asif Fuad

American International University-Bangladesh

SUPERVISOR(S):
Sofia Vallecorsa

PROJECT SPECIFICATION

Graph Neural Networks(GNN), Machine Learning Inference, Field Programmable Gate Arrays(FPGAs), High Level Synthesis(HLS), Synthesis Optimization and hls4ml (High Level Synthesis for Machine Learning).

ABSTRACT

Graph Neural Network possess prospect in track reconstruction for the Large Hadron Collider use-case due to high dimensional and sparse data. Field Programmable Gate Arrays (FPGAs) have the potential to speedup inference of machine learning algorithms compared to GPUs as they support pipeline operation. In our research, we have used hls4ml, a machine learning inference package for FPGAs and we have evaluated different approaches: Pipeline, Dataflow and Dataflow with pipeline blocks architectures. Results show that the Pipeline architecture is the fastest but it has some disadvantages such as large loop unrolling and non-functioning reuse factor. Our solution of large loop unrolling takes more than 100 hours to complete synthesis of Hardware architecture from High Level Synthesis(HLS) C++ code. On the other hand, our implementation of the system using the Dataflow architecture is too slow but it does not solve large synthesis time. So we proposed a modified Dataflow architecture where some of the building blocks are in pipeline architecture. We have found prominent results from this architecture but we have not solved the large synthesis time problem.

TABLE OF CONTENTS

1. INTRODUCTION

Track reconstruction is an important pattern recognition task for any Large Hadron Collider (LHC) experiment. This task identifies useful hits associated with a particle trajectory (‘track’) and while, at the same time, discarding fake hits, segments and tracks. Our project is a part of Deep Underground Neutrino Experiment(DUNE) but results are applicable to any LHC case. The Deep Underground Neutrino Experiment (DUNE) is an international experiment for neutrino science and proton decay studies. One of the main objective of this experiment is to understand Neutrino-Antineutrino (and matter-antimatter) asymmetries which may answer why the universe exists after the Big Bang theory. Track Reconstruction by using Graph Neural Network(GNN), a powerful class of methods from Geometric Deep Learning [5, 6], has both real-time and non-real-time applications based on its implementation technique and use-case. Image based methods such as Convolutional Neural Networks(CNNs) and Long Short-Term Memory(LSTM) are used for track reconstruction as well, but these approaches face issues to scale up to realistic HL-LHC data due to the high dimensionality and sparsity [1-4] of the data. Instead, by representing the data as a space-point structure and treat it as a graph of connected hits (see Fig. 1), we can exploit Graph Neural Network (GNN) to describe the structure of the data with high accuracy. The graph can be constructed by connecting plausibly-related hits using geometric constraints or other pre-processing algorithms like the Hough Transform. A GNN model can learn on this graph representation and solve tasks with predictions over the graph nodes, edges, or global state. Here we work with a binary segment classification model that learns to identify many tracks at once by classifying the graph edges (hit pairs). The inputs to this model are the node features (the 3D hit coordinates) and the connectivity specification [1].

At first, let’s consider a simple graph that has 3 layers and each layer has 3 nodes(hits) (shown in Figure 1). In this case, if a node can be connected to all the nodes of the next successive layer, then the number of possible edges is 18. Our main objective is to identify “Good” segments that connects the nodes with useful edges. The full Graph Neural Network model consists of an input transformation layer followed by recurrent alternating applications of the EdgeNetwork(segment classification) and NodeNetwork (hits classification). The architecture for the segment classification network is illustrated in figure 2. With each iteration of the model, information is propagated through the graph and the network, adaptively learns how to strengthen important connections and weaken the useless ones [1].

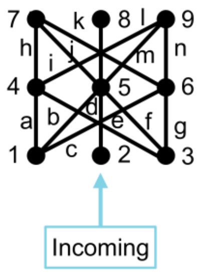


Figure 1. A simple 3 Layer 3 Track Graph [4].

An *EdgeNetwork* computes weights for every edge of the graph using the features of the start and end nodes whereas a *NodeNetwork* computes new features for every node using the edge weight aggregated features of the connected nodes on the previous and next detector layers separately as well as the nodes’ current features. [1].

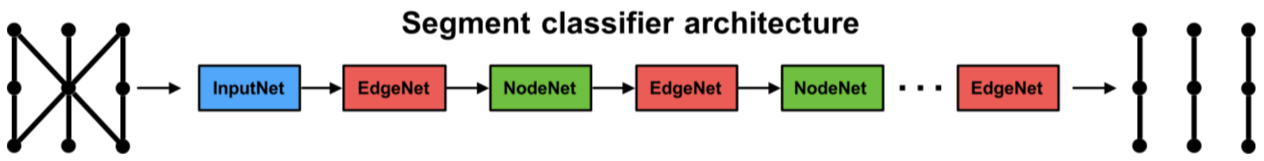


Figure 2. GNN based Segment Classifier Architecture [1]

The InputNet is one layer MLP with tanh activations. On the other hand, The EdgeNetwork is a 2-layer MLP with tanh and sigmoid activation. Edge (E) weight array is

$$w = f_{edge}(R_i^T X, R_o^T X)$$

The NodeNetwork is a 2-Layer MLP with tanh activations. Node features are:

$$X' = f_{node}((R_o \odot w)R_i^T X, (R_i \odot w)R_o^T X, X)$$

Here, X is (N × D) node feature matrix, R_i is a (N×E) association matrix of nodes to input edges and R_o is a (N×E) association matrix of nodes to output edges where N is number of Nodes and E is number of edges. The complete architecture for 3 layers, 3 tracks and 4 layers, 4 tracks is given in figure 3.

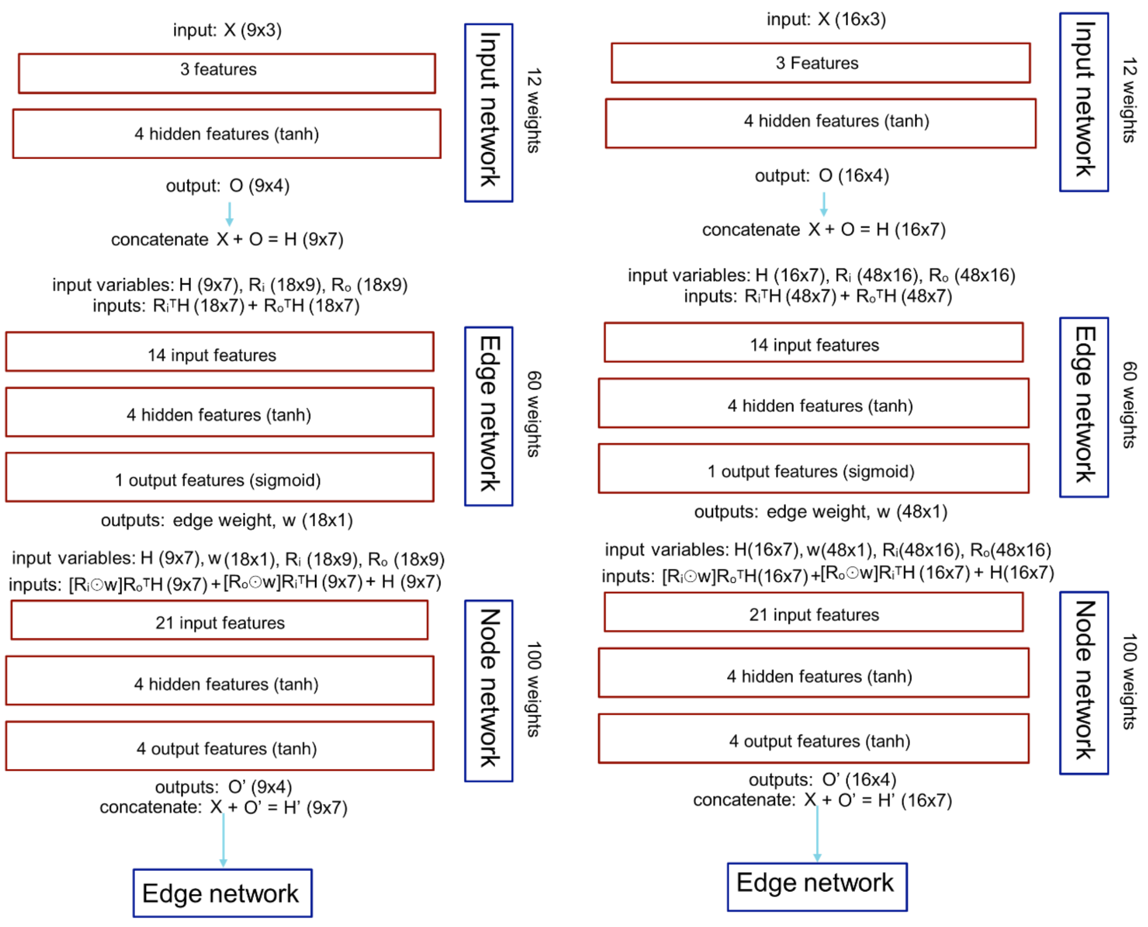

input: X (9x3)
3 features
4 hidden features (tanh)
output: O (9x4)
Input network
12 weights
concatenate X + O = H (9x7)
input variables: H (9x7), Rᵢ (18x9), Rₒ (18x9)
inputs: RᵢᵀH (18x7) + RₒᵀH (18x7)
14 input features
4 hidden features (tanh)
1 output features (sigmoid)
outputs: edge weight, w (18x1)
Edge network
60 weights
input variables: H (9x7), w (18x1), Rᵢ (18x9), Rₒ (18x9)
inputs: [Rᵢ⊙w]RₒᵀH (9x7) + [Rₒ⊙w]RᵢᵀH (9x7) + H (9x7)
21 input features
4 hidden features (tanh)
4 output features (tanh)
Node network
100 weights
outputs: O' (9x4)
concatenate: X + O' = H' (9x7)
Edge network
input: X (16x3)
3 Features
4 hidden features (tanh)
output: O (16x4)
concatenate X + O = H (16x7)
input variables: H (16x7), Rᵢ (48x16), Rₒ (48x16)
inputs: RᵢᵀH (48x7) + RₒᵀH (48x7)
outputs: edge weight, w (48x1)
input variables: H(16x7), w(48x1), Rᵢ(48x16), Rₒ(48x16)
inputs: [Rᵢ⊙w]RₒᵀH(16x7) +[Rₒ⊙w]RᵢᵀH (16x7) + H(16x7)
outputs: O' (16x4)
concatenate: X + O' = H' (16x7)

Figure 3. Complete architecture of 3 layers, 3 tracks (left) and 4 layers, 4 tracks(right) GNNs

For both architectures the number of hidden layers are same, however the number of multiplication increases as the number of hits(nodes) increases. Table 1 shows the number of multiplication for different layers and tracks combinations.

Table 1: Number of multiplication for EdgeNetwork and NodeNetwork.

GNN	EdgeNetwork- 60 Weights	NodeNetworks-100 Weights
3 layers, 3 tracks (9 Nodes, 18 Edges)	18 x 60 =1080 multiplications 18 x 9 x 7= 1,134 multiplications	9 x 100= 900 multiplications
4 layers, 4 tracks (16 Nodes, 48 Edges)	48 x 60 =2880 multiplications 48 x 16 x 7= 5,376 multiplications	16 x 100= 1600 multiplications
5 layers, 5 tracks (25 Nodes, 100 Edges)	100 x 60 =6000 multiplications 100 x 25 x 7= 17,500 multiplications	25 x 100= 2500 multiplications

As the multiplications increases radically with respect to number of nodes and edges, we will need to restrict the number of Digital Signal Processors (DSPs) in Field Programmable Gate Arrays (FPGAs).

2. Why implementation on FPGAs?

Field Programmable Gate Arrays (FPGAs) are electronic devices that are based on matrix of configurable logic blocks (CLBs) connected via programmable interconnects. FPGAs can be reprogrammed to the desired application or functionality requirements after manufacturing. This feature distinguishes FPGAs from Application Specific Integrated Circuits (ASICs), which are custom manufactured for specific design tasks. Mission-critical applications (for example, autonomous vehicle, manufacturing, etc.) require deterministic low-latency. The data flow pattern in such applications may be in streaming form, requiring pipelined-oriented processing. Due to the non-re-configurability after manufacture and the high initial cost of fabrication of Application Specific Integrated Circuits (ASICs), FPGAs are popular for prototyping digital systems. As far as neural networks are concerned the advantage in using FPGAs relies in the fact that we can upgrade the neural model every time the model is trained with better accuracy. This would be impossible on an ASIC. As shown in Figure 4, FPGAs also have an advantage with respect to Graphics Processors (GPU) in that they support pipelined operations, thus speeding up inference [7, 8, 9]. The advantages of FPGA over ASIC and GPU is shown in Figure 4.

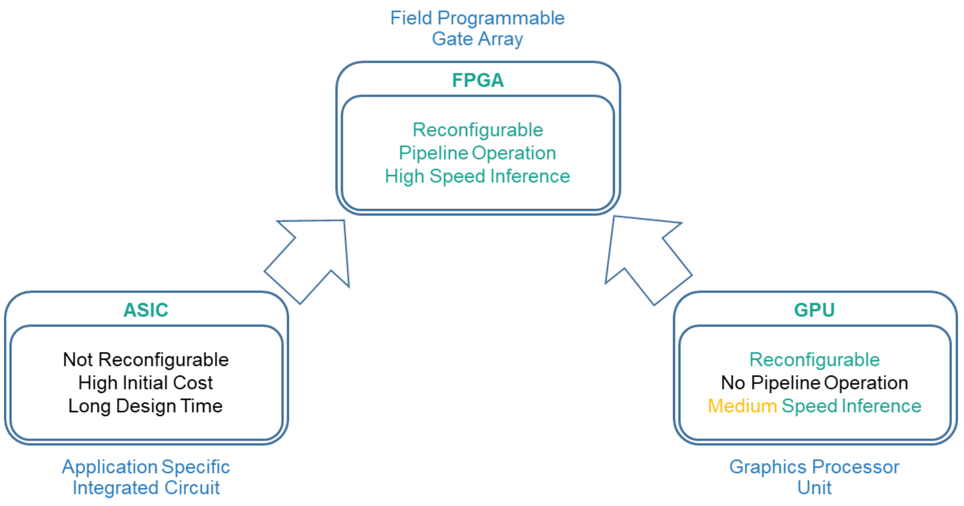
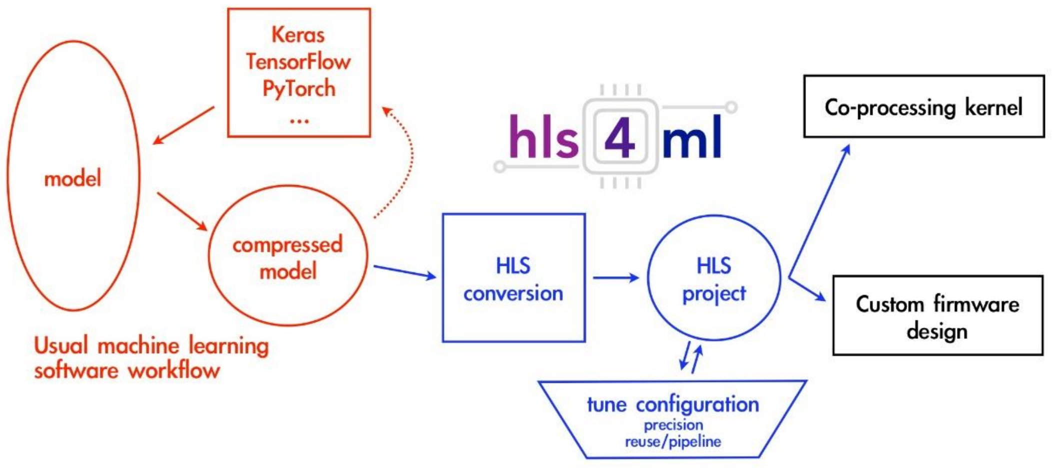


Figure 4. Advantages of FPGA over ASIC and GPU.

3. Hls4ml for FPGAs

hls4ml is a package for machine learning inference in FPGAs that can be used to develop firmware implementations of machine learning algorithms using High Level Synthesis (HLS). The main goal of hls4ml is to provide an efficient and fast translation of machine learning models. Traditional open-source machine learning package models can be translated into HLS which can be configured on FPGAs after some kind of hardware optimization. The resulting HLS project can be then used to produce an IP which can be plugged into more complex designs or be used to create a kernel for CPU co-processing. The workflow for hls4ml is given in Figure 5.



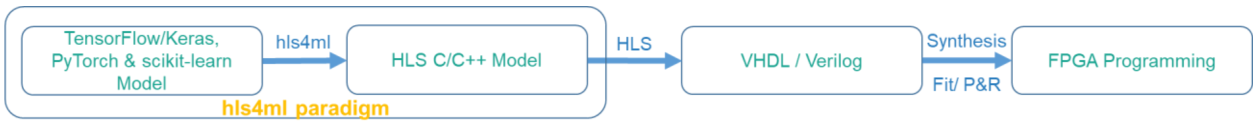


Figure 5. (top) hls4ml workflow[3], (bottom) a simpler hls4ml workflow [3]

Usually, FPGAs are programmed using Hardware Description Language (HDL) such as VHDL or Verilog but this approach is quite time consuming for testing complex algorithms. Due to this fact, High Level Synthesis, translating High Level Languages C/C++ code into VHDL and Verilog code, is getting more and more popular. Still this is a difficult situation for Machine Learning and Deep Learning Practitioners as most of the Machine Learning development platforms works with Python and there is no way to program the FPGA using the Python Machine Learning code. Instead, using hls4ml it is possible to convert trained model into HLS code once the training phase of Machine Learning is completed. The user has the freedom to define many of the parameters of their algorithm to best suit their needs.

The users can control aspects of their model such as:

- size/compression - though not explicitly part of the hls4ml package, this is an important optimization to efficiently use the FPGA resources
- precision - define the precision of the calculations in your model
- dataflow/resource reuse - control parallel or serial model implementations with varying levels of pipelining

The hls4ml package enables fast prototyping of a machine learning algorithm implementation in FPGAs, greatly reducing the time to results and giving the user intuition for how to best design a machine learning algorithm for their application while balancing performance, resource utilization and latency requirements [3].

4. Basics of FPGA Implementation

Field Programmable Gate Arrays (FPGAs) is built with Look Up Tables(LUTs), Flip-Flops (FFs), Block RAM and Digital Signal Processors (DSPs). LUTs are basically Static RAM which is why any imaginable digital logic can be implemented in a FPGA whereas DSP is used for faster multiplication and additions. Two digital system implementation techniques 1. Dataflow Architecture and 2. Pipeline Architecture are typically used to reduce the latency. In the Dataflow architecture, all the unit block functions are parallelized as shown in Figure 6 [10].

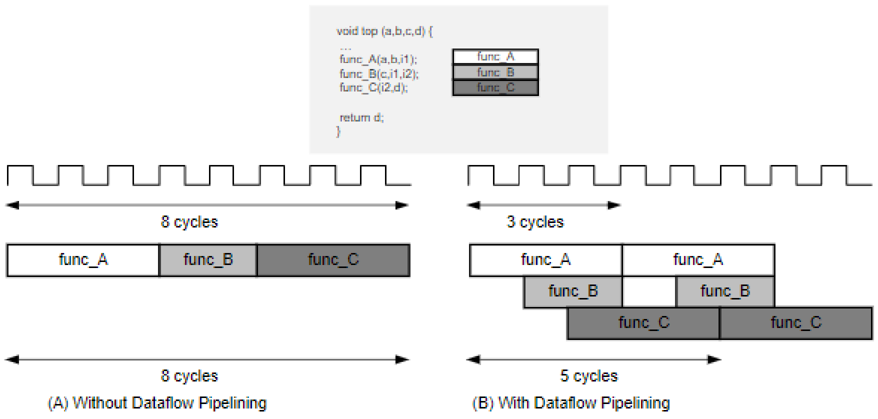


Figure 6. Dataflow architecture [10].

On the other hand, Pipeline architecture parallelize the loops in a function as we can see in Figure 7.

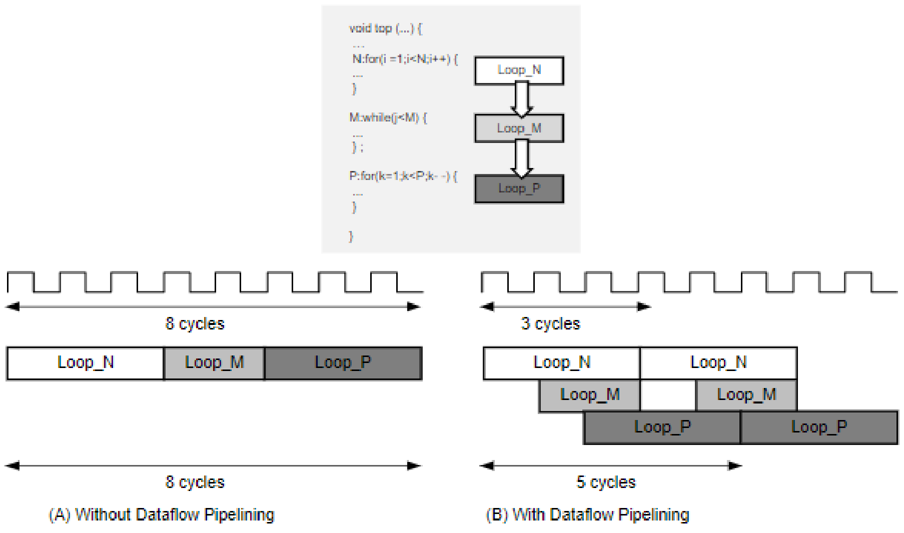


Figure 7. Pipeline Architecture [10].

5. Results and Analysis

Our objective was to study how to run Graph Neural Network on FPGAs and for this, we focused on EdgeNet, the Segment Classifier implemented by the J. M. G. Duarte et al. at Fermilab. The hls4ml workspace, containing the trained model and corresponding HLS codes, was provided in GitHub. Table 2 contains the Vivado C Synthesis results of the reference GNN implementation [4]. The first steps in the HLS implementation are represented by C Simulation and C Synthesis. The C simulation step verifies that the design code for a certain input yields the expected outputs. On the other hand, C synthesis converts the HLS code to hardware architecture. Once we were able to successfully run C simulation and synthesis, we experimented the GNN with different architectures:

1. Complete Pipeline Architecture
2. Complete Dataflow Architecture
3. Dataflow Architecture with Pipelined Blocks

Input layer, EdgeNetwork and NodeNetwork of GNN are Multilayer Perceptron so we integrated improved HLS implementation of dense layers (Multilayer Perceptron) into GNN designed by V. Loncar at CERN. hls4ml uses a special parameter reuse factor which limits the number of multiplier used in the multiplication. In all of our experimentation, only the reuse factors 1,7, 21 and above could be used as this quantity is GNN dense layer dependent. Reference implementations were targeted such as the Kintex Family FPGA: xcku115-flva1517-1-c, which we have used for most of our tests. Vivado HLS C Synthesis Resource Utilization of Reference GNN is given in Table 2.

Table 2: Vivado HLS C Synthesis Resource Utilization Reference GNN.

Reference: Graph Neural Network inference in FPGA									
Tracks Layers		**3 Tracks 3 Layers**				**4 Tracks 4 Layers**			
Resource Utilization		Device: xcku115-flva1517-1-c				Device: xcku115-flva1517-1-c			
		Vivado HLS C Synthesis				Vivado HLS C Synthesis			
		BRAM_18K	DSP48E	FF	LUT	BRAM_18K	DSP48E	FF	LUT
Reuse Factor	Available	4320	5520	1326720	663360	4320	5520	1326720	663360
	Available SLR	2160	2760	663360	331680	2160	2760	663360	331680
1	Total(Used)	184	5067	181959	425524	516	17424	660071	1786841
	Utilization SLR (%)	8	183	27	128	23	631	99	538
	Utilization(%)	4	91	13	64	11	315	49	269
10	Total(Used)	68	578	121855	362723	268	1908	636742	1506481
	Utilization SLR (%)	3	20	18	109	12	69	95	454
	Utilization(%)	1	10	9	54	6	34	47	227

a. Complete Pipeline Architecture

At first, the complete network was designed using the pipeline architecture where all the adjacency matrices and layers were pipelined including the adjacency matrices for Edge and Node Networks (shown in Figure 8). Table 3 contains the Vivado C Synthesis and Vivado Synthesis result of 3 Layers and 3 Tracks GNN for Reuse factor of 3. There is a difference between the usage of DSP blocks and LUTs in Vivado C Synthesis and Vivado Synthesis. Vivado Synthesis tries to optimise speed of the design by using more DSP blocks for computation in place of LUTs. So DSP utilisation increases in Vivado Synthesis. 3 Layers and 3 Tracks GNN with Reuse factor 1 crosses the SLR utilization limit but if we increase the Reuse factor to 7 and above, we can reduce the SLR utilization under 100%. Tuning the Reuse factor is not efficient since we observed that, even increasing the reuse factor, the DSP count does not decrease proportionately. We have found that the DSP count for adjacency matrices follows the Reuse of the DSP block but MLP layers don't follow proportionally. In case of a Reuse factor of 7 and 21, we can see that our design takes 18% and 21% less area than the reference implementation. If we check the timing results in table 4, it can be seen that a fully pipelined architecture is 68% and 35% faster than reference for Reuse factors of 7 and 21 respectively.

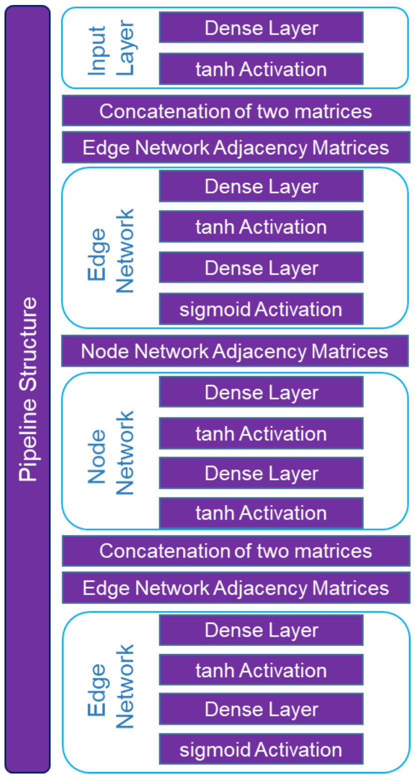


Figure 8. A complete Pipelined GNN.

Table 3: Vivado C Synthesis and Vivado Synthesis of Complete Pipelined 3 layers, 3 tracks GNN.

GNN Resources Usage for Pipeline Architecture		Device: **xcku115-flva1517-1-c**							
		Utilization Estimates: Vivado HLS C Synthesis				Utilization Estimates: Vivado Synthesis			
		DSP48E	Change	LUT	Change	DSP48E	Change	CLB LUT	Change
	Available	5520	na	663360	na	5520	Na	663360	na
	Available SLR	2760	na	331680	na	na	Na	na	na
Reuse=1	Total(Used)	5067	-776.64%	420412	-15.90%	5049	-773.53%	143112	60.55%
	Utilization(%)	91	-810.00%	63	-16.67%	91.47	-814.70%	21.57	60.06%
	Utilization SLR (%)	183	-815.00%	126	-15.60%	na		na	
Reuse=7	Total(Used)	1484	-156.75%	295398	18.56%	3309	-472.49%	106199	70.72%
	Utilization(%)	26	-160.00%	44	18.52%	59.95	-499.50%	16.01	70.35%
	Utilization SLR (%)	53	-165.00%	89	18.35%				
Reuse=21	Total(Used)	1161	-100.87%	285845	21.19%	3023	-423.01%	107392	70.39%
	Utilization(%)	21	-110.00%	43	20.37%	54.76	-447.60%	16.19	70.02%
	Utilization SLR (%)	42	-110.00%	86	21.10%				

Table 4: Timing from Vivado C Synthesis of Complete Pipelined 3 layers, 3 tracks GNN.

	Latency (Clock Cycles)						Timing (ns) Target: 10.00
	Latency				Interval		
	Min	Change	Max	Change	Min	Max	Estimated/Uncertainty
Reuse=1	21	81.58%	21	81.58%	1	1	8.619/1.25
Reuse=7	36	68.42%	36	68.42%	7	7	
Reuse=21	73	35.96%	73	35.96%	21	21	

We could not fit the 4 Layers, 4 Tracks GNN into the Kintex FPGA. In table 5, it can be seen that even if, we increase the reuse factor to 21 we cannot reduce the resource utilization less than 100 percent. Fully Pipelined architecture only improved the speed (given in Table 6).

Table 5: Vivado C Synthesis and Vivado Synthesis of Complete Pipelined 4 layers, 4 tracks GNN.

GNN Resources Usage for Pipeline Architecture		Device: **xcku115-flva1517-1-c**							
		Utilization Estimates: Vivado HLS C Synthesis				Utilization Estimates: Vivado Synthesis			
		DSP48E	Change	LUT	Change	DSP48E	Change	CLB LUT	Change
	Available	5520	na	663360	na	5520	na	663360	na
	Available SLR	2760	na	331680	na	na	na	na	na
Reuse=1	Total(Used)	17616	-823.27%	1798687	-19.40%	5520	-189.31%	2285042	-51.68%
	Utilization(%)	319	-838.24%	271	-19.38%	100	-194.12%	344.46	-51.74%
	Utilization SLR (%)	638	-824.64%	542	-19.38%				
Reuse=21	Total(Used)	4640	-143.19%	1355382	10.03%	2582	-35.32%	1025563	31.92%
	Utilization(%)	84	-147.06%	204	10.13%	46.78	-37.59%	154.6	31.89%
	Utilization SLR (%)	168	-143.48%	408	10.13%				

Table 6: Timing from Vivado C Synthesis of Complete Pipelined 4 layers, 4 tracks GNN.

	Latency (Clock Cycles)						Timing (ns) Target: 10.00
	Latency				Interval		
	Min	Change	Max	Change	Min	Max	Estimated/Uncertainty
Reuse=1	23	78.10%	23	78.10%	1	1	8.619/1.25
Reuse=21	63	40.00%	63	40.00%	21	21	8.619/1.25

To fit the design, we needed a bigger FPGA from Virtex family. So we chose the largest FPGA, xcvu13p-fhga2104-1-I, available in Vivado HLS. From table 7, we can see that, for the reuse factor 21, the 4 layers, 4 tracks GNN fits in the Virtex FPGA. Vivado synthesis presents similar behaviour of optimizing the speed by using more DSPs than LUTs. If we compare table 6 and table 8, it is clear that the speed of this architecture is almost similar for both Kintex and Virtex FPGAs.

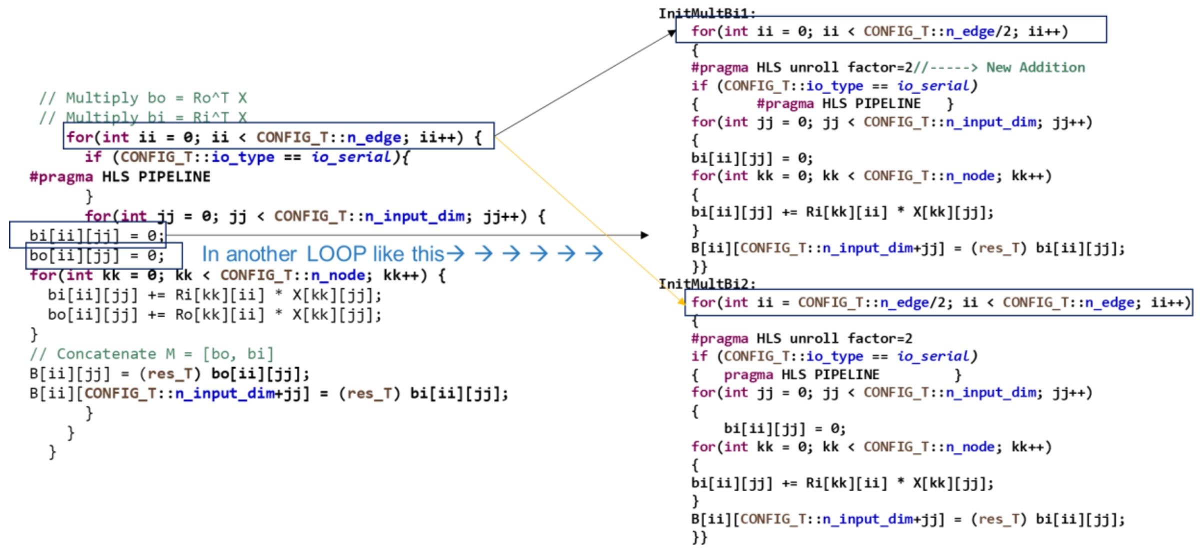
Table 7: Vivado C Synthesis of Complete Pipelined 4 layers, 4 tracks GNN for Virtex FPGA.

GNN Resources Usage for Pipeline Architecture		Device: **xcvu13p-fhga2104-1-i**			
		Vivado HLS C Synthesis		Vivado Synthesis	
		DSP48E	LUT	DSP48E	CLB LUT
Reuse	Available	12288	1728000	12288	1728000
Reuse=1	Total(Used)	17616	1790783	12288	991030
	Utilization(%)	143	103	100	57.35
Reuse=21	Total(Used)	4640	1355242	12272	629730
	Utilization(%)	37	78	99.87	36.44

Table 8: Timing from Vivado C Synthesis of Complete Pipelined 4 layers, 4 tracks GNN for Virtex FPGA.

	Latency (Clock Cycles)				Timing (ns) Target: 10.00
	Latency		Interval		
	Min	Max	Min	Max	Estimated/Uncertainty
Reuse=1	21	21	1	1	8.729/1.25
Reuse=21	61	61	21	21	8.729/1.25

When we moved to 5 layers, 5 tracks GNN, we faced large loop unrolling: in the Vivado HLS pipeline approach, loops are unrolled in order to process all the data in parallel. Unfortunately, the unrolled data for the 5 layers, 5 tracks GNN, reaches the 64GB RAM limit and therefore Vivado kills the synthesis. After a detailed investigation, we found that Vivado HLS can't perform $\boldsymbol{b_o = R_o^T X}$ and $\boldsymbol{b_i = R_i^T X}$ in a same 'for loop' for a high number of edges (in this case 50). So we divide these similar tasks with similar number of iterations into different loops and different size of iterations. Then, we impose the Vivado HLS pragma '*#pragma HLS unroll factor=2'* to limit the unrolling: unfortunately the synthesis took a very large amount of time (around 105 hours). The result is given in table 9 and it shows that this design does not fit into the FPGA.



```
 // Multiply bo = Ro^T X
 // Multiply bi = Ri^T X
   for(int ii = 0; ii < CONFIG_T::n_edge; ii++) {
      if (CONFIG_T::io_type == io_serial){
#pragma HLS PIPELINE
      }
      for(int jj = 0; jj < CONFIG_T::n_input_dim; jj++) {
bi[ii][jj] = 0;
bo[ii][jj] = 0;
for(int kk = 0; kk < CONFIG_T::n_node; kk++) {
  bi[ii][jj] += Ri[kk][ii] * X[kk][jj];
  bo[ii][jj] += Ro[kk][ii] * X[kk][jj];
}
// Concatenate M = [bo, bi]
B[ii][jj] = (res_T) bo[ii][jj];
B[ii][CONFIG_T::n_input_dim+jj] = (res_T) bi[ii][jj];
      }
    }
  }
```

```
InitMultBi1:
  for(int ii = 0; ii < CONFIG_T::n_edge/2; ii++)
  {
  #pragma HLS unroll factor=2//-----> New Addition
  if (CONFIG_T::io_type == io_serial)
  {       #pragma HLS PIPELINE    }
  for(int jj = 0; jj < CONFIG_T::n_input_dim; jj++)
  {
  bi[ii][jj] = 0;
  for(int kk = 0; kk < CONFIG_T::n_node; kk++)
  {
  bi[ii][jj] += Ri[kk][ii] * X[kk][jj];
  }
  B[ii][CONFIG_T::n_input_dim+jj] = (res_T) bi[ii][jj];
  }}
InitMultBi2:
  for(int ii = CONFIG_T::n_edge/2; ii < CONFIG_T::n_edge; ii++)
  {
  #pragma HLS unroll factor=2
  if (CONFIG_T::io_type == io_serial)
  {   pragma HLS PIPELINE          }
  for(int jj = 0; jj < CONFIG_T::n_input_dim; jj++)
  {
      bi[ii][jj] = 0;
  for(int kk = 0; kk < CONFIG_T::n_node; kk++)
  {
  bi[ii][jj] += Ri[kk][ii] * X[kk][jj];
  }
  B[ii][CONFIG_T::n_input_dim+jj] = (res_T) bi[ii][jj];
  }}
```

Table 9: Vivado C Synthesis of Complete Pipelined 5 layers, 5 tracks GNN.

Graph Neural Network inference on FPGA: 5 Tracks 5 Layers									
Resource Utilization		Device: **xcvu13p-fhga2104-1-i**							
		Vivado HLS C Synthesis							
		BRAM_18K	DSP48E	FF	LUT	Latency		Interval	
	Available	5376	12288	3456000	1728000	Min	max	Min	Max
Reuse=28	Total(Used)	550	9300	604827	4153953	38	38	28	28
	Utilization(%)	10	75	12	240				

Resource utilization in different layers was investigated as it was observed that DSP blocks were not reduced proportional to the increase of the Reuse factor. From the table 10, it can be seen that NodeNetwork adjacency matrices scales down in DSP count in accordance with reuse factor where EdgeNetwork does not use any DSP blocks as it is only Boolean operation. But MLP layers were not following the Reuse factor.

Table 10: Resource Utilization in different layers in Complete Pipelined 3 layers, 3 tracks GNN.

3 Tracks 3 Layers Device: xcku115-flva1517-1-c						
	EdgeNetwork Adjacency Matrices		NodeNetwork Adjacency Matrices		All the MLPs	
Reuse Factor	DSP48E	LUT	DSP48E	LUT	DSP48E	LUT
1	0	145,440	2268	166687	2799	106,149
7	0	72720	324	176843	1160	43055
21	0	72720	108	171915	1053	38723

In conclusion, we found two main issues with the pipeline implementation: first, a large synthesis time for large networks and second, Reuse factor is not working as expected. So we moved to investigate to Dataflow.

b. Complete Dataflow Architecture

Due to the long simulation time and Reuse factor issue, the complete network was designed using the Dataflow architecture where all the functions and layers were in Dataflow including the adjacency matrices for Edge and Node Networks (shown in Figure 9).

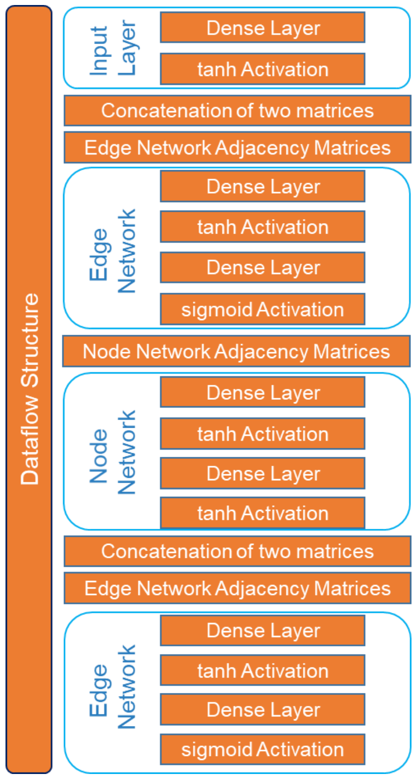


Figure 9. A complete Dataflow GNN

In the Dataflow approach, the design uses less amount of resources and less synthesis time but it requires huge latency. At the same time, the Reuse factor functions properly but the large unrolling issue is not still solved for 5 layers, 5 tracks GNN. Table 11 and table 12 contain Vivado C synthesis and timing results. If we compare the results with the pipeline implementation, it is evident that this implementation is area efficient but the delay makes it not feasible for further implementation.

Table 11: Vivado C Synthesis of Complete Dataflow 4 layers, 4 tracks GNN.

GNN Resources Usage for Dataflow Architecture		Device: **xcku115-flva1517-1-c**			
		Utilization Estimates: Vivado HLS C Synthesis			
		DSP48E	Change	LUT	Change
	Available	5520	na	663360	Na
	Available SLR	2760	na	331680	Na
Reuse=1	Total(Used)	2473	-327.85%	477649	-31.68%
	Utilization(%)	44	-340.00%	72	-33.33%
	Utilization SLR (%)	87	-335.00%	144	-32.11%
Reuse=21	Total(Used)	120	79.24%	483488	-33.29%
	Utilization(%)	2	80.00%	72	-33.33%
	Utilization SLR (%)	4	80.00%	145	-33.03%

Table 12: Timing of Complete Dataflow 4 layers, 4 tracks GNN.

	Latency (Clock Cycles)					
	Latency				Interval	
	Min	Change	Max	Change	Min	Max
Reuse=1	3003	-2534.21%	3003	-2534.21%	632	632
Reuse=21	5684	-4885.96%	5765	-4957.02%	1424	1442

c. Dataflow architecture with Pipelined Blocks

Complete Dataflow supports functioning reuse factor but long latency made this GNN impractical. So we tried to pipeline some blocks of the Graph Neural Network. The modified architecture is given in Figure 10.

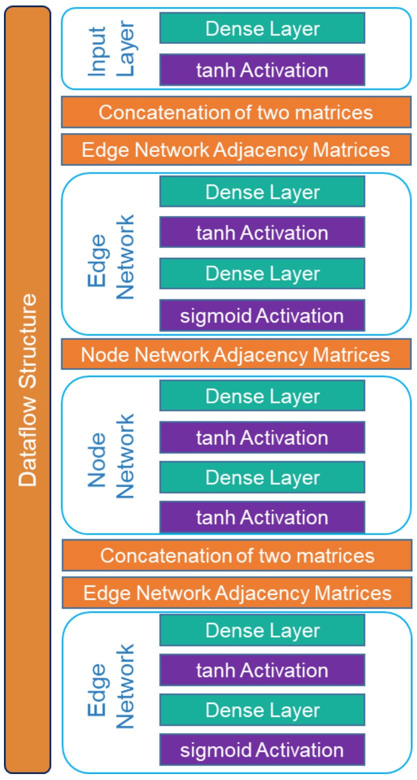


Figure 10. A Dataflow GNN with Pipelined Blocks.

The overall GNN was developed in Dataflow manner but different units were pipelined in different ways. Activation layers are completely pipelined whereas in Dense layers only for loops are pipelined. In this way, Reuse factor is working for all the layers except dense layers. DSP usage in dense layers are fixed in this implementation but they are small in number which does not create problem for small implementation. But in this implementation, Vivado Synthesis crashes for 4 layers, 4 tracks GNN. It takes up full 62.4GB RAM and crashes. We followed a similar Xilinx forum solution for this but it did not solve the issue.

Table 13: Vivado C Synthesis of Complete Pipelined 3 layers, 3 tracks GNN.

GNN Resources Usage for Pipeline Architecture		Device: xcku115-flva1517-1-c			
		Utilization Estimates: Vivado HLS C Synthesis			
		DSP48E	Change	LUT	Change
	Available	5520	-	663360	-
	Available SLR	2760	-	331680	-
Reuse=1	Total(Used)	2473	-327.85%	445565	-22.84%
	Utilization(%)	44	-340.00%	74	-37.04%
	Utilization SLR (%)	89	-345.00%	149	-36.70%
Reuse=7	Total(Used)	529	8.48%	507872	-40.02%
	Utilization(%)	9	10.00%	76	-40.74%
	Utilization SLR (%)	19	5.00%	153	-40.37%
Reuse=21	Total(Used)	313	45.85%	503840	-38.90%
	Utilization(%)	5	50.00%	75	-38.89%
	Utilization SLR (%)	11	45.00%	151	-38.53%

Table 14: Timing of 3 layers, 3 tracks GNN for Kintex FPGA.

	Latency (Clock Cycles)					
	Latency				Interval	
	Min	Change	Max	Change	Min	Max
Reuse=1	153	-34.21%	153	-34.21%	27	27
Reuse=7	160	-40.35%	160	-40.35%	28	28
Reuse=21	158	-38.60%	158	-38.60%	21	21

Table 15: Vivado C Synthesis of Complete Pipelined 4 layers, 4 tracks GNN for Virtex FPGA.

GNN Resources Usage for Pipeline Architecture		Device: **xcku115-flva1517-1-c**			
		Utilization Estimates: Vivado HLS C Synthesis			
		DSP48E	Change	LUT	Change
	Available	5520	-	663360	-
	Available SLR	2760	-	331680	-
Reuse=1	Total(Used)	10953	-474.06%	2118547	-40.63%
	Utilization(%)	198	-482.35%	319	-40.53%
	Utilization SLR (%)	396	-473.91%	638	-40.53%
Reuse=7	Total(Used)	1737	8.96%	2165281	-43.73%
	Utilization(%)	31	8.82%	326	-43.61%
	Utilization SLR (%)	62	10.14%	652	-43.61%
Reuse=21	Total(Used)	713	62.63%	2136505	-41.82%
	Utilization(%)	12	64.71%	322	-41.85%
	Utilization SLR (%)	25	63.77%	644	-41.85%

Table 16: Timing of 4 layers, 4 tracks GNN for Virtex FPGA.

	Latency (Clock Cycles)					
	Latency				Interval	
	Min	Change	Max	Change	Min	Max
Reuse=1	289	-175.24%	289	-175.24%	57	57
Reuse=7	289	-175.24%	289	-175.24%	57	57
Reuse=21	289	-175.24%	289	-175.24%	51	51

The issue about long synthesis time is not still solved for large designs like 5 layers, 5 tracks GNN. Even for this structure, the synthesis took 125 hours.

Table 17: Resource Utilization in different layers in Complete Pipelined 5 layers, 5 tracks GNN.

Graph Neural Network inference on FPGA: 5 Tracks 5 Layers										
Resource Utilization		Device: **xcku115-flva1517-1-c**								
		Vivado HLS C Synthesis								
		BRAM_18K	DSP48E	FF	LUT	Latency		Interval		
	Available	4320	5520	1326720	663360	Min	max	Min	Max
	Available SLR	2160	2760	663360	331680	511	511	103	103
Reuse=21	Total(Used)	2088	1877	760229	7049620				
	Utilization(%)	48	34	57	1062				
	Utilization SLR (%)	96	68	114	2125				

6. Discussions and Conclusions

We have presented a study on the possibility to synthetize GNN architectures on FPGA exploring different approaches. We have been able to improve the GNN synthesis using the Pipeline architecture and achieving 60% improvement in terms of speed with respect to the reference implementation. Using the Pipeline Architecture, 3 Layers x 3 Tracks and 4 Layers x 4 Tracks network fit into an FPGA. As far as the Dataflow approach is concerned, we have been able solve the large memory usage of loop unrolling but we could not improve on the time needed to run the synthesis: the Dataflow architecture turned out to be too slow for practical applications. On the other hand, we devised a modified Dataflow architecture improving its speed compared to the standard Dataflow approach, even though issues remained concerning the large unrolling memory usage and large synthesis time.

References

[1] Novel deep learning methods for track reconstruction. Available at: arXiv:1810.06111v1

[2] M.M. Bronstein, J. Bruna, Y. LeCun, A. Szlam, P. Vandergheynst, CoRR abs/1611.08097 (2016), 1611.08097

[3] J. Duarte et al., "Fast inference of deep neural networks in FPGAs for particle physics", JINST 13 P07027 (2018), arXiv:1804.06913.

[4] ACTS Meeting of August 30, 2018. Meeting report available at https://indico.cern.ch/event/753577/contributions/3123602/attachments/1707996/2752966/acts-gnn-Aug30.pdf

[5] HEP.TrkX Meeting. Report Available at: https://indico.cern.ch/event/658267/contributions/2881175/attachments/1621912/2581064/Farrell_hept rkx_ctd2018.pdf

[6] HEP.TrkX: https://heptrkx.github.io/

[7] Arrow.com website article. Available at: https://www.arrow.com/en/research-and-events/articles/fpga-vs-cpu-vs-gpu-vs-microcontroller

[8] Lance Simms blog. Available at: https://lancesimms.com/Microprocessors/CPU_vs_GPU_vs_FPGA.html

[9] Numato Lab online article. Available at: https://numato.com/blog/differences-between-fpga-and-asics/

[10] Xilinx Documentation. Available at: https://www.xilinx.com/support/documentation/sw_manuals/xilinx 2015_2/sdsoc_doc/topics/calling-coding guidelines/concept_data_flow.html#concept_data_flow__fig _mnp _jxh_ks