Mines Paris

Cycle Ingénieur Civil
Option : MAREVA
Mathématiques Appliqués : Robotique, Vision, Automatique

Sorbonne Université

Faculté des Sciences et Ingénierie
Master Informatique
Parcours : SESI
Systèmes Electroniques
et Systèmes Informatiques

# Graph Neural Network trajectometry algorithm on FPGA

LEMOINE Corentin ; corentin.lemoine@minesparis.psl.eu

**Supervisors** : FEDERICI Luca, BAUDOT Jérôme
luca.federici@iphc.cnrs.fr ; +33 3 88 10 62 92
jerome.baudot@iphc.cnrs.fr ; +33 3 88 10 66 32

Master thesis : march-september, 2022

IPHC – Dpt of Subatomic Research
Team C4Pi & Belle II

## Abstract :

An efficient trigger system is essential in most particle physics experiment to cope with the conjunction of high data rate and high background events. In the Belle II experiment, the Level 1 trigger allows reducing the trigger rate to 30kHZ while the beams cross at 40MHz [1]. This L1 trigger aggregates information from several sensors and takes a decision within 5us based on energy and tracks reconstruction. Deploying such fast online decision logic forces to use of dedicated algorithm and computation platform, namely FPGA. Even though the L1 trigger uses data from several sensors, it relies mostly on the Central Drift Chamber (CDC) for its track trigger and makes poor use of the silicon based vertex detector (or inner tracker) [2]. It can be justified by the difficulties both to extract the high data rate from the inner tracker and to extract track information from this high data volume.

An upgraded version of the inner tracker planned in the coming years will likely install 5 or 7 layers of CMOS Monolithic Active Pixel Sensor close to the Interaction Point, providing precise information on time and position of particles. This upgrade could (if taken into account during the design phase) provide a more efficient solution to read the information from the inner tracker, making them available for the L1 trigger. If the precise hit information could become available for the L1 trigger, the question of an appropriate data processing method for pixel-based track reconstruction remains open. Several tracking algorithms exist, including some based on Learning approaches. More specifically, Dr B. Schwenker from univ Göttingen developed a Graph Neural Network (GNN) tracking algorithm tailored for this inner tracker upgraded pixel sensor. The algorithm shows good performances in the Belle II Analysis Software Framework (BASF2) but its original implementation is not suited for use in the L1 trigger due to its huge computational requirements.

This master thesis presents the efforts to deploy this GNN tracking algorithm on FPGA to try and meet the L1 trigger requirement. It should be compared to similar work in [3, 4] that also target using GNN on FPGA for real-time particle tracking. The main difference from our work is that [3, 4] focus on ATLAS based dataset.

The first section introduces the context of this work. The second present the developed method and validation procedure. The third present results of the previously developed method and analyse the feasibility of the solution.

# Contents

# Part I

# Introduction

## 1 Presentation of the laboratory

### 1.1 IPHC

Figure 1: Logos of the organisms

Institut Pluridisciplinaire Hubert Curien (IPHC) is an UMR (Unité Mixte de Recherche, UMR7178, CNRS, Université de Strasbourg) consisting in 4 departments:

- a physic department
- a chemistry department
- an ecology department
- a (newer) radiobiology department

The institute is also affiliated with the university of Strasbourg : https://www.unistra.fr/

IPHC was created in 2006 as an alliance of previously independent laboratories. It is located in the Cronenbourg campus in Strasbourg altogether with teaching facilities (École européenne de chimie, polymères et matériaux...) and other research institutes such as the Institut Charles-Sadron (UPR 22), iCube (UMR 7357), Institut de Physique et Chimie des Matériaux de Strasbourg (UMR7504)...

IPHC has a global management and each department has its own management. Departments are divided into several scientific projects and technical group. Most technical groups are shared between several research projects. More information on the hierarchy can be found on IPHC website : https://www.iphc.cnrs.fr/.

The department of subatomic research (DRS), under the coordination of CNRS's IN2P3, is the biggest department with around 60 researchers and 90 engineers and

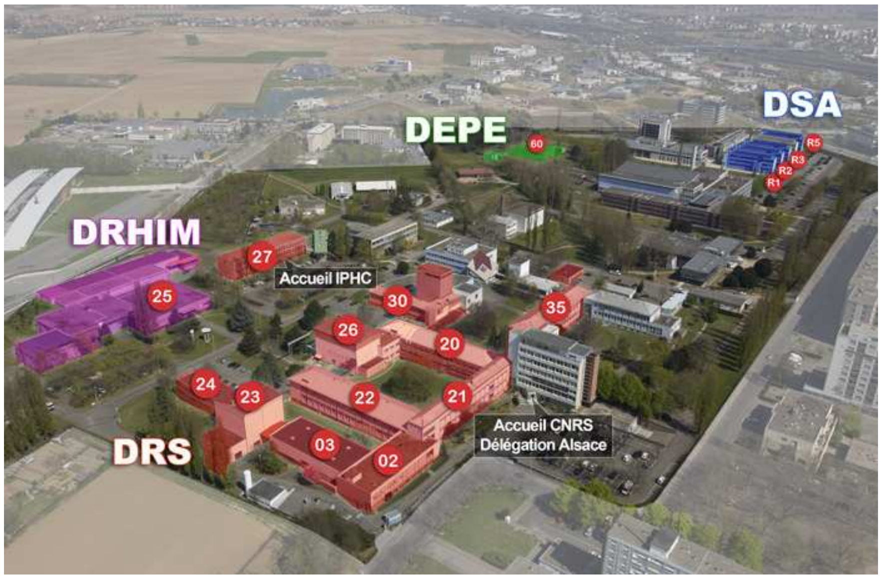

Figure 2: Aerial picture of the Cronenbourg campus with IPHC facilities highlighted. Source: IPHC

technicians (information from 2018). Its scope goes from theory to experimental physics which explains the huge proportion of technicians and engineers working on electronics or mechanical engineering for physics experiments. The DRS collaborates in several international projects, one of those is the Belle II experiment that is a particle physics experiment on the SuperKEKB accelerator in Japan.

### 1.2 C4Pi

Figure 3: Logo du groupe C4Pi

The C4Pi platform of the DRS hold the expertise on monolithic CMOS pixel sensors that are major sensors for high energy physics experiments. They were among the first to develop such sensors for particles detection with the Mimosa1 sensor from 1999 [5]. Since then, they developed new detectors using the latest technologies to increase the performances of the sensors. The platform leaders are Christine Hu-Guo (Responsable Opérationnelle, christine.hu@iphc.cnrs.fr) and Claude Colledani (Responsable Opérationnel Adjoint, claude.colledani@iphc.cnrs.fr) with Jérôme Baudot being the scientific coordinator. The C4pPi group covers the development steps from the conception of the sensors to bounding and then test and characterization. Only the production step (prototype and mass production) is outsourced to big foundries such as Tower Semiconductors that hold the required technologies.

# 2 Context

The subject of the internship is closely linked to particle colliders, here we will give the reader the basic key to understanding the motivations of the internship. Some of the information are simplified to keep them accessible to non-physicists.

## 2.1 Introduction to particle physics and detectors

The simplified goal of particle colliders is to accelerate two groups of “simple” particles in opposite directions and make the particle beams cross. The point where the beams cross is called the interaction point. Some particles will eventually collide and “break” creating other “rare” particles. The probability for two particles to collide is small, the swarms of particles most often cross without such interaction. The particles created in the collision are of huge interest for physicist to confirm or reject theories, that is why they aim at knowing as much as possible about them such as charge, energy, lifetime (the particle may decay in other particles). In order to access those information, various kinds of detectors exist.

For the purpose of the internship, we are specifically interested in CMOS sensors. CMOS sensors are known for their use in camera such as in smartphone camera where the sensor detects photons (=light) and converts it to electrical signal through the photoelectric effect. Charged particles created in colliders also generate an electrical signal in CMOS sensors through ionization of the silicon. CMOS pixel sensor (CPS) provides precise coordinates of the point where such particles cross the sensor, we will refer to them as hits. Usually, CPS are installed around the interaction point in a nearly cylindric or barrel shape following the direction of the beams. Often several layers of sensors are installed at different radii (see Fig-5) so as to be able to reconstruct the trajectories of particles.

## 2.2 Challenges

As previously mentioned, a collision occurs rarely compared to the number of beam crossing (several orders of magnitudes less often). However, due to physical phenomenon, sensors detect way more than just the particles created in collisions: the interesting particles are hidden in “noise” that is called background (it is indeed not really noise, simply particles we are not interested in). There is thus a need to filter out those useless situations that do not correspond to collision.

Closely linked to this problem is the issue of the bandwidth. A full detector can contain billions of pixels that must be read fast enough not to mix different events. Though only a small proportion of pixels is hit, a huge amount of data is produced at a high frequency. For example, the ATLAS experiment at CERN produces approximately 200 000 TB/h of raw data for all kinds of sensors, not only pixel detectors (source: atlas.cern). This is way too much to be stored, considering in addition that most of the data is useless as it contains only background. As a solution most detectors implement a trigger system that runs in real-time and discards most of useless events.

The trigger (usually) relies on reconstructing the tracks of particles using information from a multi-layer tracker and triggering the detector only if some trajectories originating from the interaction point are detected.

## 2.3 Belle II

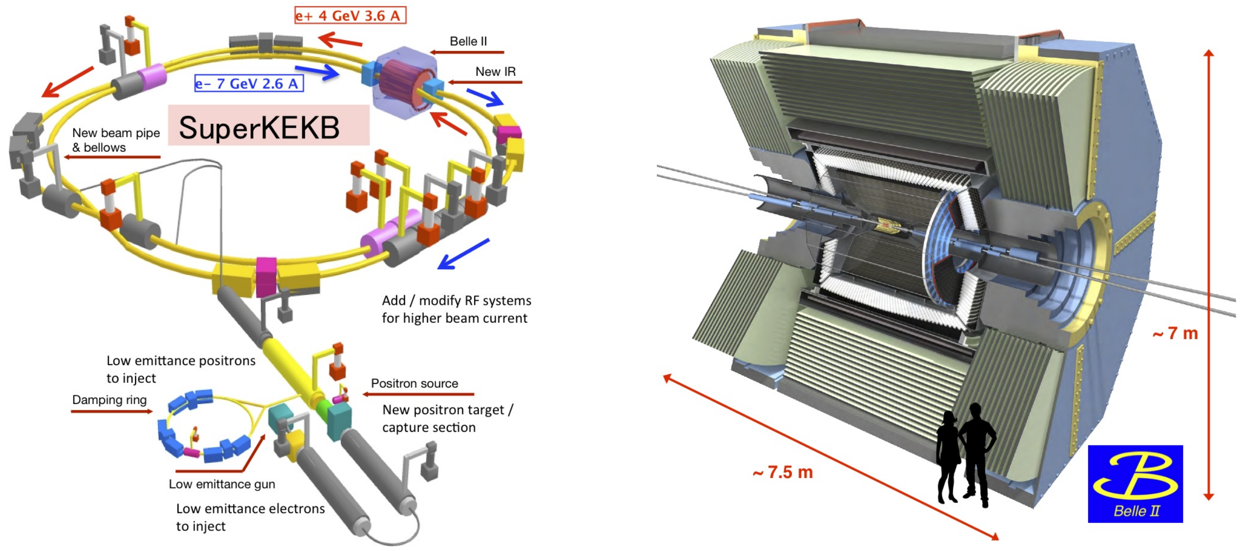
e+ 4 GeV 3.6 A
Belle II
New IR
e- 7 GeV 2.6 A
SuperKEKB
New beam pipe & bellows
Add / modify RF systems for higher beam current
Low emittance positrons to inject
Damping ring
Positron source
New positron target / capture section
Low emittance gun
Low emittance electrons to inject
~ 7 m
~ 7.5 m
Belle II

Figure 4: Schematics of SuperKEKB (left) and Belle II detector (right). Source: Belle II

We will illustrate the explanation with the case of Belle II (an experiment in Japan) that is the focus point of the internship. For Belle II, the particle accelerator SuperKEKB (Fig-13, left [6]) produces counter rotating 7GeV electrons and 4GeV positrons on a 3km circumference accelerator. Those beams cross each other at a frequency of 250MHz (every 4ns) but a collision occurs on average at a frequency lower than 30kHz, that is to say less than 1 out of 10 000 crossings. The physics of SuperKEKB is such that the collision mainly produce B mesons, which is a specificity of this accelerator. Belle II detector (Fig-13, right [1]) is 7m high and 7.5m long with the interaction point at its centre. It consists of several kinds of detectors mostly organised in concentric cylindric layers (and end cap disks). The inner part of the detector is our focus. It relies on an inner vertex detector (yellow in the schematics) and is surrounded by the CDC: central drift chamber (grey in the schematics). In order to discard in real-time collisionless events and reduce the data rate, a Level 1 trigger is implemented on FPGA (Field Programmable Gate Array) to decide in 5µs whether or not the event should be discarded. It is worth noticing that this L1 trigger does not rely on the inner tracker (in yellow) but mostly on the CDC. Indeed, the inner detector produces too much data to be dealt with. The CDC information are used to reconstruct the tracks of particles that follow a helical trajectory. If this system detects trajectories originating from the intersection point, it triggers the detector. This L1 trigger reduces the event rate to 30kHz (beam crossing are at 250MHz) while keeping nearly 100% of interesting physics events.

## 2.4 Belle II upgrade proposal

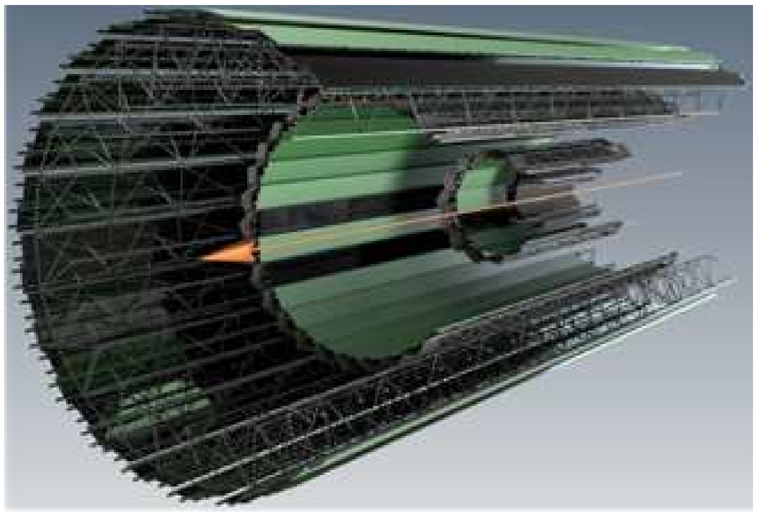

Figure 5: Illustration of Belle II upgrade proposal with the 5 layers of CMOS pixel sensors.Source: S. Bettarini, Belle II

As all accelerators and detectors, Belle II is frequently upgraded to use the latest technologies and remain state of the art. In this context and in order to keep up with the luminosity (equivalent to collision rate) increase of the accelerator, the inner tracker of Belle II will be replaced in the following years. In 2022 luninosity reach $4 \times 10^{34} cm^{-2}.s^{-1}$ and the goal for 2028 is $6 \times 10^{35} cm^{-2}.s^{-1}$. Several possibilities are currently under development [7], one of them is to replace the inner tracker by 5 layers of CMOS pixel sensors. The luminosity increase will also lead to increased background rate, which might in turn saturate the CDC, compromising tracks reconstruction and thus trigger.

In the context of the upgrade, a possible solution to the luminosity increase is to redesign a L1 trigger using data produced by the CMOS sensors. Due to the pixel size and time resolution, CPS based trigger are more robust to background. A solution to implement the trigger using only information from the 5 layers of CMOS sensors was developed by Dr. Benjamin Schwenker, physicist at the university of Göttingen, in the form of a Graph Neural Network (GNN) algorithm. The GNN is used to perform tracking (that is to say reconstruct particle track). The reconstructed trajectories are then used to make the trigger decision. The algorithm was written in python using Pytorch library and tested with simulated data using Belle 2 analysis software framework (basf2) [8]. The performances evaluation procedure still has to be validated but the first results show that the trigger is efficient. The algorithm will be presented in the following part, but we can already mention that it is quite complex (in term of number of operations). As such, it was designed without any real-time considerations and cannot be used for the level 1 trigger. The goal of the internship will be evaluate an hardware implementation of this algorithm and see if it can meet the requirements of the L1 trigger.

## 2.5 Graph Neural Network (GNN)

In this paragraph we will present the GNN algorithm proposed by Dr. Schwenker. The algorithm was originally developed in the context of the ATLAS experiment [9]. The concept of a GNN is to have a neural network that can work on a graph structure. In our case, hits will be the nodes of the graph and two nodes will be linked by an edge under some conditions (being in two adjacent layers, not too far from one another...). Among those edges that connect two hits, some connect two hits that correspond to the same particle and thus are part of a particle trajectory. Such edges are called true edges. The goal of the algorithm is to select true edges among others and thus reconstruct each segment of the trajectories. In the figure bellow, step 1 shows the individual hits (=nodes) that are the raw data from the sensors. Step 2 shows edges in dashed blue connecting nodes. Among them, not all correspond to a particle trajectory. The role of the GNN is to classify edges so we can separate the "false edges" from the "true edges" that correspond to a particle track and produce step 3 where some edges are selected. Finally step 4 shows, the reconstructed trajectories, we can notice that not all hits correspond to a track, the orange stars indicate background hits.

Regarding the architecture of our GNN, the GNN is composed of 3 different networks that all have their own characteristics and purpose. 1) The input network encodes nodes information (i.e., 3 coordinates of the hit) into a node embedding space (here, a 64-dimensional space). 2) The edges network takes as input the two embedding of the nodes defining an edge and produces a single value between 0 and 1 that can be interpreted

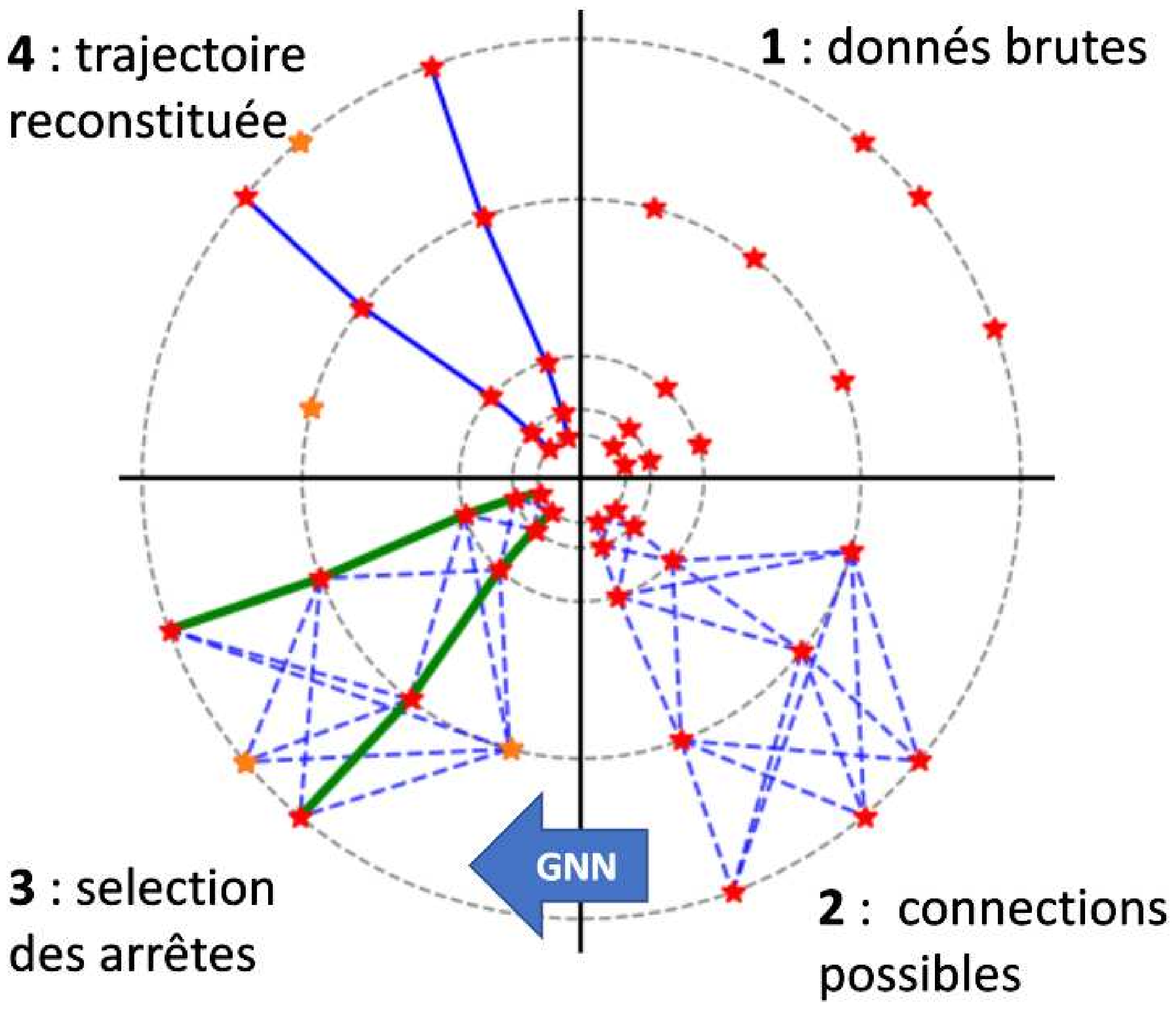
4 : trajectoire reconstituée
1 : donnés brutes
3 : selection des arrêtes
GNN
2 : connections possibles

Figure 6: Figure explaining the steps for reconstructing trajectories from raw sensor hits

as the probability for the edge to be a true edge. 3) The node network takes as input an aggregation (i.e., weighted average with the result of the edge network being used as the weight) of its neighbour embedding and updates node embedding.

The algorithm starts by embedding nodes information using the input network. Then it updates the node embedding by running the edge network on each edge. The results are then use as weight for the aggregation of neighbouring node used as input to the node network. This update step is repeated a certain number of times and allows message passing between nodes. All three networks are fully connected layers with ReLU (Rectified Linear Unit) activation.

Training data as well as well as validation data are available using simulation of the upgraded detector. No real data will be available before the upgrade of the detector and by that time it will be too late to develop the trigger.

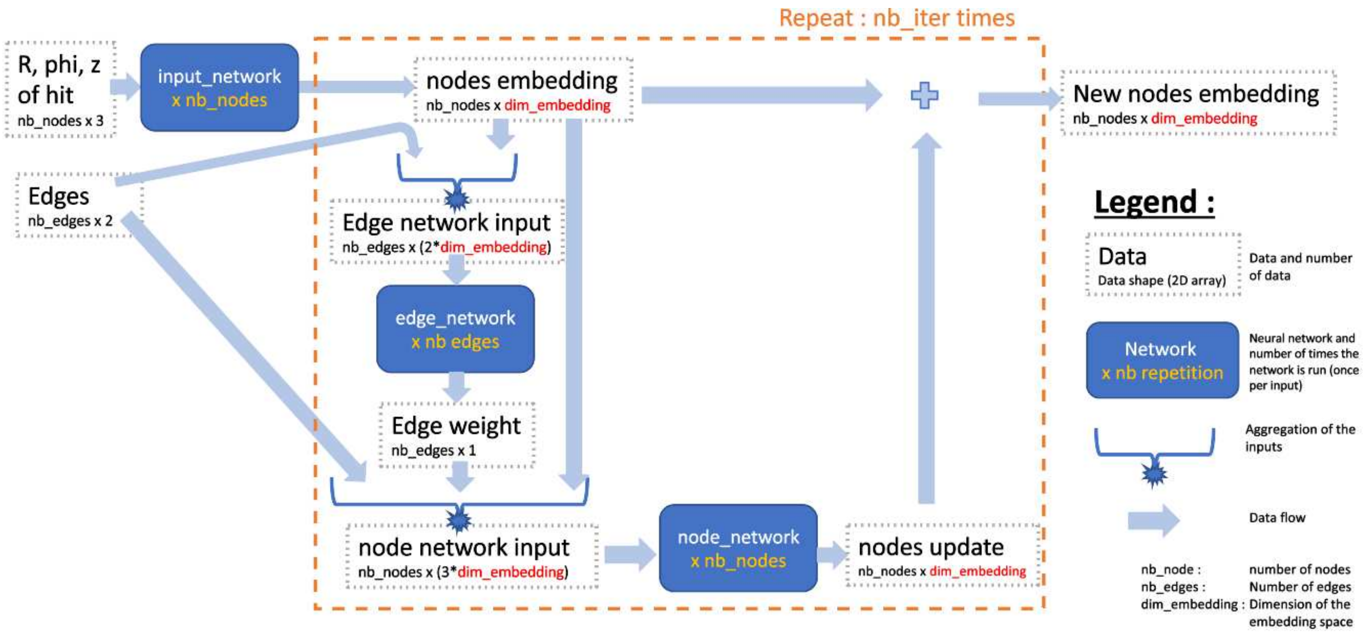

Figure 7: Dataflow of the GNN algorithm

## Part II

# Technical solution

This part present in detail the solution developed. It also contains interesting results about the method.

## 3 Conversion from Python to C++

The solution we planned to use to convert Python code to FPGA uses hls4ml to transform the Python code to C++. Hls4ml however does not fully support Pytorch and fails to convert our "exotic" GNN.

We thus faced a choice between using another tool, improving the tool to fully support Pytorch or manually modifying the generated C++. All solution comes with its advantages and drawbacks. Using other tools means finding something equivalent, also in terms of licencing. Modifying the tools seems to be the best solution but would require too much time for a 6 months internship, especially when the efficiency of the solution is not granted. We thus choose to manually modify the C++ code produced by hls4ml: this break the automation of the process but it is the best trade-off between time and results in the development phase. If the solution proves useful, further support in hls4ml might be added later on.

When using hls4ml on the Pytorch GNN, it produces a vivado HLS project with all required files and the corresponding C++ code altogether with hls4ml library. As explained previously, the main function does not implement the GNN but all layers sequentially. We can thus take advantage of the situation by modifying only the links between networks and not the networks themselves.

## 4 Algorithm decomposition

In the schematics of the algorithm Fig-2.5, we can see that there are three main steps. First is the input network so we get node embedding and edges. Then we update several times the node embedding. Each update just needs the embedding produced by the previous update and the edges. Finally, the edge and/or global network must be run to produce the edge probability/trigger.

We will focus on implementing the update step since it is the most complex step. Depending on the resource use, this update step could then be repeated several times inside the same FPGA or in several FPGAs (each FPGA doing one update step and passing the output to the next one).

In the appendix one can find a simplified version of the code for this update step. In this code each task (edge aggregation, edge network, node aggregation, node network) is separated in a dedicated function.

We can also see that this code uses “pragma HLS”. It gives some information to the tool that will translate the C++ to hardware code. There are indeed many possibilities in this process of translation (some are fast but use a lot of resources, other are slower but more compact...) and using pragmas helps guiding the tools regarding specific requirements.

As ilustrated in Fig-2.5, executing one update step of the algorithm requires 2 networks and 2 aggregations. Both types of operation must be studied and optimized.

## 4.1 Networks optimisation

The networks implementation is already dealt with by hls4ml. There is not much to improve on this side in term of algorithm. However, the datatypes can be improved: by default, hls4ml uses 16bits fixed point with all the same scaling factor. Choosing better datatypes is possible and was done on our exemple but it is not so simple as many datatypes must be set separately while making sure it does not cause overflow.

Using fewer bits is also possible but it quickly leads to large errors compared to floating point (several %). A very promising opportunity is to use quantization aware training. Basically, instead of training the network with floating point and then translating to fixed point it takes into account the use of fixed point during the training phase. Using this method allows keeping great accuracy with a fixed-point network sometime even with less than 8bits. It is however not in the scope of this internship.

As a summary, the individual networks were not optimized more than what is done with hls4ml.

## 4.2 Aggregation optimisation

Concerning the aggregations, a manual implementation was realized, which may be suboptimal compared to an automatic tool. In particular, the node aggregation step is quite complex. For each edge, the node aggregation basically performs the following instruction:

```
node_network_in[edge_end] += w*node_embedding[edge_start];
```

Where node_network_in and node_embedding are matrices.

The value of edge_end and edge_start dependant on the edge (that is an input) and thus not constant. It means that this simple line has three operations:

1. select the node embedding corresponding to the start of the edge
2. multiply it by the weight w (this weight is the edge weight computed by the edge network). See Fig-2.5
3. add the result to the node_network_in corresponding to the end of the edge.

They seem simple but step 1 and 3 might be quite complex depending on the implementation choice. Indeed, the nodes embeddings are either stored in a RAM inside the FPGA (but then you cannot access several values at the same time) or in distributed small memories by splitting the matrix (but then it uses more resources).

## 5 Synthesis of the C++ to produce HDL

Once the "manual" conversion from Pytorch to C++ was validated on a few simple examples we could focus on synthetising the project to produce HDL code and FPGA implementation. It clearly appeared that synthesis could be a long task when trying to produce a highly parallel code. Even for a small network of 16 neurons the synthesis takes 8h when trying to split all arrays to allow more parallelisation (this is the default comportment of hls4ml).

At this point we must verify the produced HDL but most importantly configure properly the HLS tool (using directives called pragmas) so it produces a code that is fast while keeping a decent resource usage to fit in the FPGA. Hopefully Vivado HLS produces report that allows to estimate the performances of different solutions without having to run the code on real hardware and profile the execution. Both the verification step and the configuration exploration were conducted at the same time : the goal was not to verify the final algorithm but rather to develop the test procedure and validate that the synthesis works "in general".

## 6 Metrics of merit

Synthesis produce HDL code that varies a lot depending on the configuration. For now, we assume the HDL to be functionally equivalent to the initial C++ and we focus on evaluating the performance of this generated code. The actual results of the implementation are given in the result section. This part focus on explaining the concepts required for the analysis of the results. This analysis mostly relies on Vivado HLS report that gives timming and resources usage estimation. Here we explain those metrics and why they are important. An example of such a report can be found in Fig-**??**

Latency: time delay between inputs and outputs. Here our GNN must have a latency of a few µs.

Initiation Interval: Usually, when running a function with no parallelism, we wait for the output of the function to call the function with a new input. This comportment is illustrated on the left of Fig-8, where the horizontal axis is time. The three steps of the first function call (RD, CMP, WR (the meaning of those tasks don't matter)) are run before starting the next function call. The time between two consecutive outputs of the function is then 3 cycles: this is the initiation interval.

However, using pipelining (illustrated on the right of Fig-8) one can improve the throughput by accepting a new input before the previous execution is over.

To some extent, the concept of pipelining can be compared to Fordism. Without pipelining, a multipurpose machine executes operation RD then CMP then WR before starting a new execution. With pipelining, 3 specialized machines are available (one for each task). The first task (here RD) is performed on the first input. As soon as the first RD task is done, its result goes to the second machine that will do the CMP task. In the meanwhile, the RD machine is free and start executing the RD task on the second input and so on. The time for a specific input to be fully treated (=latency) does not decrease but the time between two consecutive inputs (=initiation interval) can be reduced: it is no longer the sum of time of all tasks but the time of the longest task.

Obviously, this pipelining is expensive in terms of resources since several task must be run at the same time but this is a good solution to deal with high input rate.

In technical words, the constraints on the implementation are a latency of a few µs and a pipelining of 100ns. Since a common clock period for the kind of the FPGA we target is around 5ns it means the initiation interval (interval between two inputs) must be around 20 cycles.

Resources: An FPGA is a lattice of reconnectable hardware blocks, the number of resources available is thus fixed. If a project tries to use more resources than the FPGA provides, then it is sure to fail. The main resources are :

- block memory (BRAM) to store big amount of data but with limited access due to the number of ports of the memory
- Digital Signal Processor (DSP) that are tailored to quickly perform simple digital operation like multiplication and addition.

```
================================================================
== Performance Estimates
================================================================
+ Timing:
    * Summary:
    +--------+---------+----------+------------+
    | Clock  |  Target | Estimated| Uncertainty|
    +--------+---------+----------+------------+
    |ap_clk  | 5.00 ns | 5.022 ns |   0.62 ns  |
    +--------+---------+----------+------------+

+ Latency:
    * Summary:
    +---------+---------+----------+----------+-----+-----+----------+
    |  Latency (cycles) |  Latency (absolute) |  Interval | Pipeline |
    |   min   |   max   |    min   |    max   | min | max |   Type   |
    +---------+---------+----------+----------+-----+-----+----------+
    |      106|      106| 0.532 us | 0.532 us |   20|   20| function |
    +---------+---------+----------+----------+-----+-----+----------+

================================================================
== Utilization Estimates
================================================================
* Summary:
+---------------------+---------+-------+--------+---------+-----+
|         Name        | BRAM_18K| DSP48E|   FF   |   LUT   | URAM|
+---------------------+---------+-------+--------+---------+-----+
|Total                |       13|   1760|  199828|   755389|    0|
+---------------------+---------+-------+--------+---------+-----+
|Available            |     7560|   1800| 2148480|  1074240|    0|
+---------------------+---------+-------+--------+---------+-----+
|Utilization (%)      |    ~0   |     97|       9|       70|    0|
+---------------------+---------+-------+--------+---------+-----+
```

Figure 8: Simplified report produced by Vivado HLS

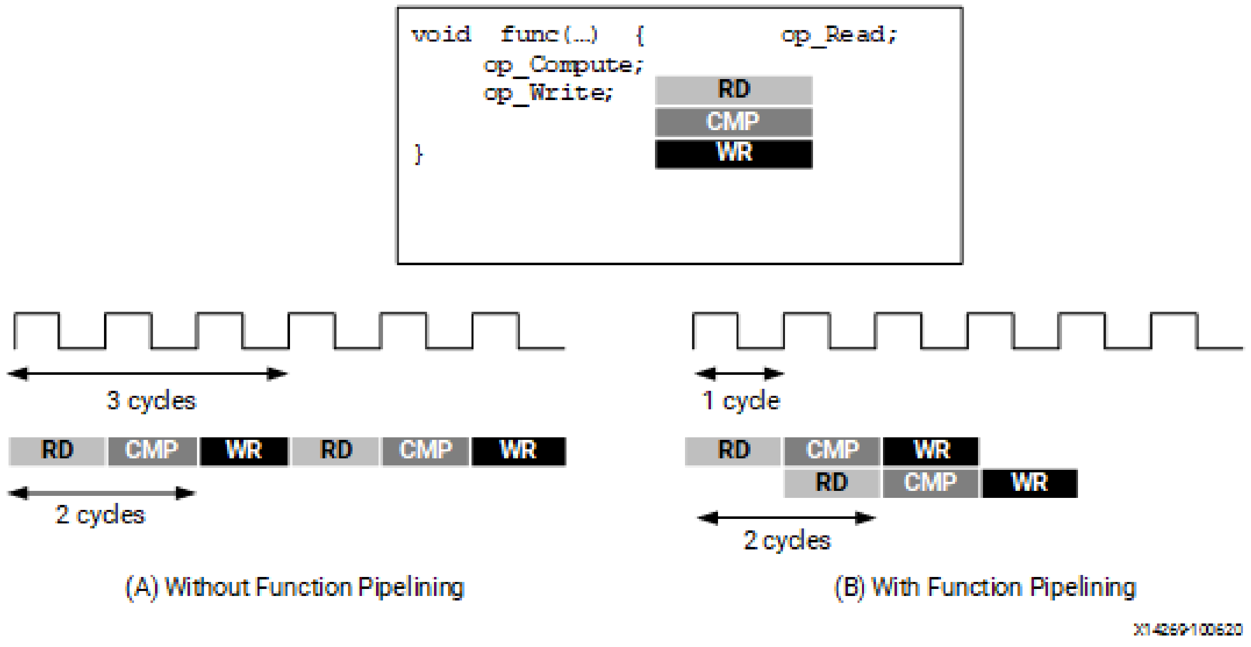

Figure 9: Illustration of function pipelining by Xilinx. Sources Xilinx.

- Flip Flops (FF) that are small memory (1 FF = 1 bit) with highly parallel access since each bet can be accessed simultaneously
- Look Up Table (LUT). They can be programmed to represent any small combinatorial circuit. By combining several LUT, complex operations can be achieved. Since it can implement any circuit, it can be used to implement multiplication for example but a single operation will use a lot of LUT compared to DSP where one DSP is enough for (usually) 16 bits operation.

Each resource is limited but can sometimes be compensated with another. For example, if FF use is too high then some data can be stored in BRAM instead, but with higher access cost.

## 7 Configuration option

To change the performances, one can change either the algorithm itself or its configuration. Configuration mostly includes changing data representation or using pragmas : we will describe the most import ones. Similarly to previous section, it only describes the concepts that are required to understand the results and not the results themselves.

Fixed-point: The report does not directly show the impact of datatype nor on performance. However, choosing a different data representation (which is part of the

configuration) has a huge impact so we present the concept here.
In floating-point the numbers are stored in mantissa*2ˆexponent form. This has the advantage of representing really small and big numbers with the same format and allows easy calculation from the user point of view. A common issue with floating-floating point is that operations are quite complex to do in hardware. To add two floating-point, you first need to scale them to the same exponent and only then you can add the mantissa. In a computer all those complex operations are done in a floating-point unit so the complexity is solely on the hardware designer and the user can enjoy an easy-to-use data type. However, this requires a lot of resources that we cannot really afford. In fixed point the format looks the same: integer*2ˆscaling with scaling being a constant predetermined integer. As a consequence, two fixed-points with the same scaling can be added with just integer addition. Another advantage is that it uses only integer computation and we can choose exactly the word length to use (16bits, 8bits but also any other value since hardware is not limited to usual datatypes). Using fixed-points and with an appropriate size is an essential step to reduce the resource use in a FPGA. Fixed-point computations are way easier for the hardware but the difficulty relies on the programmer: he must choose the correct scaling and number of bits to store the values. Since the scaling is no longer automatic as it is for floating point, using wrong type can lead to overflows or poor accuracy.

Pragmas: We describe here the main Vivado HLS pragmas

- Unroll : When executing a for loop on a single thread CPU, each iteration of the loop is executed one after the other. Sometime iterations are totally or partially independent from one another and can be executed simultaneously in parallel. Using the unroll pragma forces the HLS tool to try (but not necessarily succeed) executing all or part of the iteration at the same time. Obviously executing several iterations at the same times means repeating the hardware and thus increase resource usage.

- Array partition : C++ array are by default stored in the BRAM of the FPGA. It has the advantage that it can store huge array without resource issue. However, access to those data is thus limited to 2 data per cycles (since BRAM have at most 2 ports) and it can be a problem if one wants to parallelize. A solution to this access problem is to partition the array that is to say split it in several smaller array. Each array is then stored separately and can be accessed concurrently. Once can chose different options to partition the array. For example, a 2D array can be partitioned fully or partially along each dimension. Since small array are stored in FF instead of BRAM fully partitioning array implies a huge use of FF but allows simultaneous access to all elements which can be really useful for highly parallel execution.

- Pipeline : the pipeline concept is explained previously. The pipeline pragma asks the HLS tool to try pipelining a part of the design with a specified initiation interval (with no guarantee of success). To be able to pipeline, the tool will automatically try to unroll loops. One of the main causes pipelining can fail is because it fails to schedule access to data on a BRAM due to the limited number of ports.

# 8 Verification procedure

The verification procedure mostly consits in verifying that the same input running through the Python GNN and the Verilog GNN (or running on a real hardware) gives the same output. We can then also verify that the classification performances of the hardware GNN are similar to the Python ones. If the two version produce numeric results that are close enough then we can expect most of them to be classified the same way and thus similar performances.

First in order to test we must generate lots of different inputs to run the GNN on. Those inputs should cover as much as possible the different values a real trigger would face so they must be carefully generated. Then we will explain the test procedure to validate the Verilog GNN as well as to test it on a real hardware.

## 8.1 Inputs generation

The inputs should be extracted from realistic data produced by a physic simulation tool so we can be as close as possible to a real trigger. A hardware implementation is however more tailored for a fixed number of inputs and the synthesized code that we produce is designed for a fixed number of nodes and edges. Real life is unfortunately not so simple and the number of hits varies from one event to the other. The actual trigger should be able to deal with a reasonable number of hits, that is to say big enough to deal with most of the cases. When (rare) events produce more hits than this fixed limit the trigger would truncate by skipping some hits. Event having less hits that the limit can also be filled with meaningless data as long as it has no impact on the computation of true inputs.

The current version of the algorithm we are able to synthesize in a reasonable size requires however a small number of nodes and edges. For our test inputs we must then select some data in an event that contains a lot of them. In order to generate no bias from this selection we created three selection option that allows to automatically generate fixed size inputs that still represent realistic values. The selection process selects randomly a sector in the detector that contains the required number of hits and then select either the best, worst or randomly picked edges among the acceptable ones. This way we are sure to cover the full “range” of inputs. Once fixed sized inputs are generated, they are written to a text file with the suitable format to be parsed by out simulator. The inputs are also exported in a hardware compatible format by being stored in a ram described with the .mif format. The inputs are then run through the Python “golden” implementation of the algorithm and the expected results are also stored in a file. The input creation is made in such a way that it is deterministic from few parameters such as a random seed. All the parameters are written as comments with the inputs. This way when a problem occurs it is easier to find the exact test pattern that causes the issue and to isolate the problem.

## 8.2 Verilog verification

The verification of the Verilog code is done with simulation using xcelium. The testbench that we use is the C++ testbench : this is in fact cosimulation since the C++ testbench use results from the simulated Verilog code. This method allows easy verification between C++ and Verilog : it the C++ simulates well the Verilog should perform similarly. It is also convenient to use C++ to read inputs and write outputs.

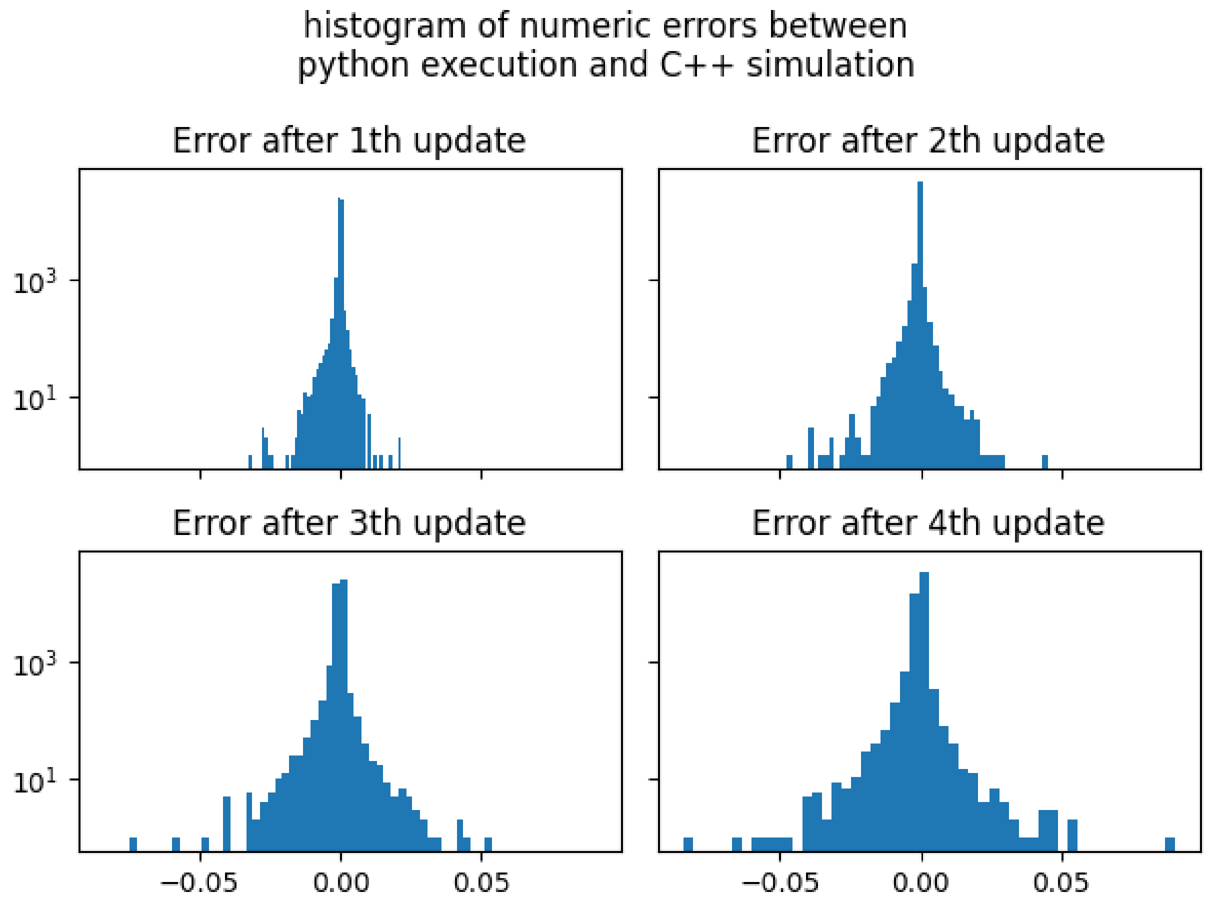

Figure 10: Histogram of numeric difference between the (golden) Pytorch implementation and the C++ simulation output

Once the simulation results are available, we can compare the expected values produced by Python and the one produced by Verilog. As expected, when we repeat several iterations, the error grows as we propagate errors. However, the magnitude of errors is acceptable. Here a logarithmic scale is used and illustrates that a huge proportion of tests outputs have an error of less than 0.02. The plot also shows that the Verilog implementation is not biased since the error are centred on zero and equally distributed between positive and negative.

## 8.3 Hardware tests on Protium

Protium[1] is an ASIC prototyping platform developed by Cadence. It consists in software altogether with a big FPGA that allows to synthesize and execute HDL code. The goal of this tool is to simulate some HDL code on hardware and thus faster than emulated on a CPU. We did run our Verilog code into the Protium. The protium flow to get the code running takes several compilation and configuration steps that we won't describe here. We also had to write and test a Verilog testbench for our program. Everything (Verilog design under test and testbench) was tested first with xcelium simulation and once validated turned to the protium. Protium allows probing some signals out of the FPGA so we could get use Vivado Chipscope to record the probe and check that the timing information produced by Vivado HLS are indeed matched. In the picture we can see that the GNN is ready to process a new input every 20 cycle (pipeline=20 cycles) and that the latency is 96 cycles.

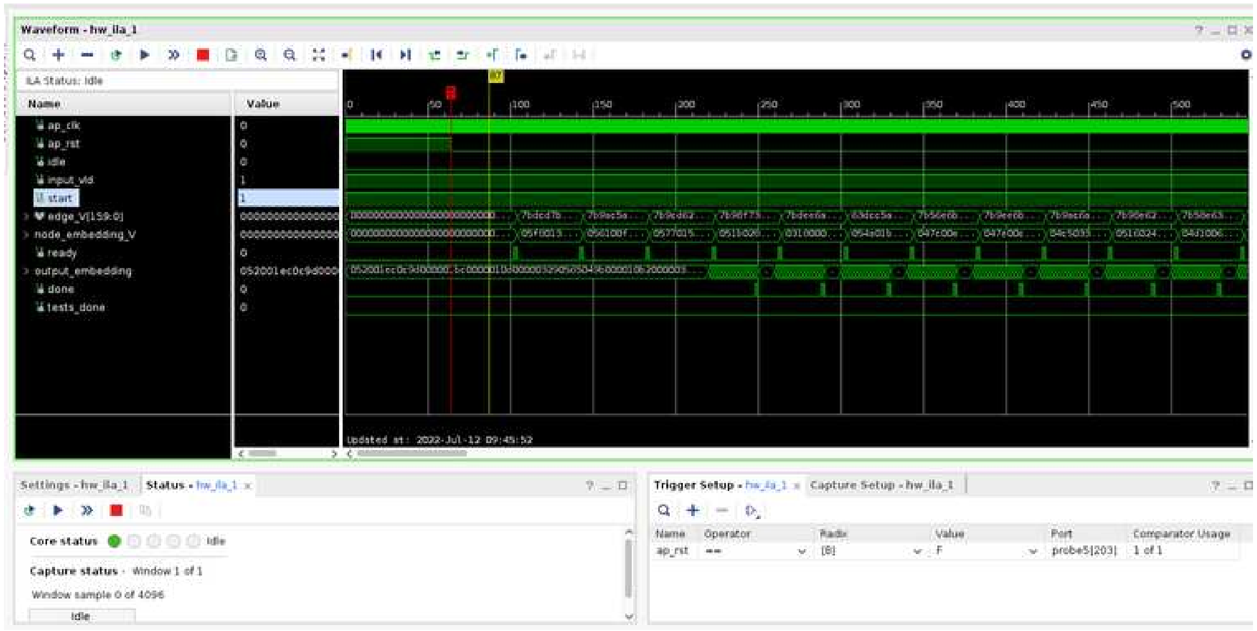

Figure 11: Screen capture of Vivado Chipscope showing the execution of the algorithm on the FPGA of the Protium.

Plotting the histogram of the difference between the C++ simulation output and protium output shows that the maximum error is less than $2^{-17}$ that is to say bellow the precision of the output. These small differences come from rounding of the output when saving them do decimal form. Except this, the results are exactly the same between the C++ simulation and hardware execution.

[1]https://www.cadence.com/en$_U S/home/tools/system-design-and-verification/emulation-and-prototyping/protium.html$

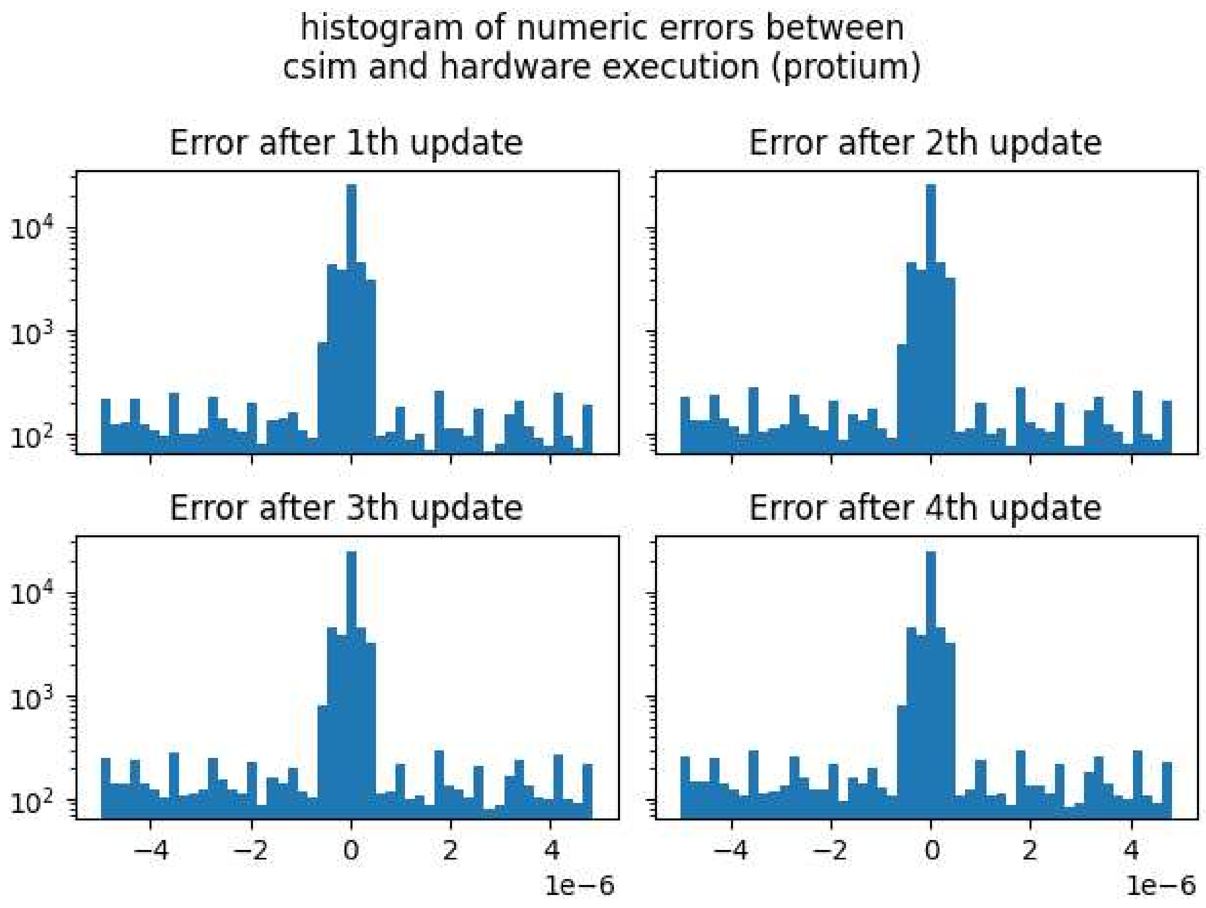

Figure 12: Histogram of numeric difference between the C++ simulation output and protium output

## Part III

# Results

The following result section tries to use the method developed previously to evaluate the possibility of a GNN based trigger. It remains a quick estimate though as a precise solution and detailed answer would require several months.

The initial algorithm proposed by B. Schwenker uses 64 embedding dimensions and 64 neurons per layer. For a medium sized input, it corresponds to more than 55 millions of multiply-accumulate operations. Even by using all the capacity of the biggest FPGA this is not possible to execute that many operations in 5us. The result section first present detailed result on a smaller version of the algorithm : 10 embedding dimension, 32 neurons per layer and a fixed input size of 16 nodes and 16 edges. This small version is optimized to meet timing constraints but cannot be extended as it already takes nearly all resources of the FPGA. A second version, that is optimized to allow big input size is then presented. This version no longer passes the timing requirement but the input size can match what we expect.

## 9 State of the art

Using a GNN as a tracking algorithm for particles physics detectors was first proposed in the context of the ATLAS experiment at CERN as one of the solutions to a Kaggle problem[2]. Since then, a few papers tried accelerating such algorithm on FPGA. The original Kaggle problem aimed at finding new tracking algorithm without focusing on real-time computation. As a consequence, the proposed solutions are far from satisfying time constraints for a trigger (several tens of seconds). The Kaggle competition was proposed by CERN targeting the ATLAS detector so the dataset contains several thousand of hits per event.

When trying to accelerate GNN most proposals start from a lightweight algorithm with networks as small as possible. FPGA acceleration of GNN tracking algorithm is a niche topic mostly covered by [3] and [4]. [3] and [4] both use 8 neurons per layer and represent node embedding with small dimension of 8 or less. Even with this small algorithm, all attempts to implement a GNN level 1 trigger failed to respect timing constraints while dealing with the full-size detector inputs. [3] and [4] both divide the detector in several sectors to split the computation in several FPGA and maintain reasonable resources usage.

## 10 Time optimized version

Our first tests to synthesise a GNN trigger were focussing timing. As a consequence, all arrays were fully unrolled and stored directly in the logic part of the FPGA to allow highly parallel access and thus reduce latency. The required resources for such approach are however quickly raising with the size of the input graph. A quick study of the node aggregation in Fig-13 proves a quadratic scaling of the required number of LUT. This comportment is expected as, in order to guarantee parallel access to all elements needed for the node aggregation, all nodes' embeddings must be accessible simultaneously. The synthesis tool thus builds multiple multiplexers. This number of multiplexers is proportional to the number of nodes and the size of the multiplexers is also proportional to the number of nodes. The steps in Fig-13 are due to the fact that the number of stages of the multiplexers increase at each power of 2. The maximum size of the input for this strategy is dominated by the number

[2]https://sites.google.com/site/trackmlparticle/home
https://www.kaggle.com/competitions/trackml-particle-identification/overview

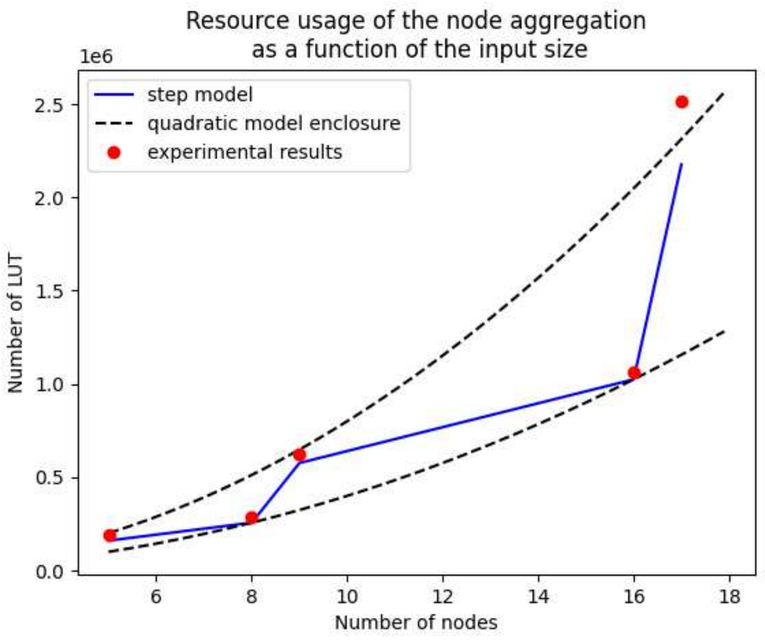

Figure 13: Ressource usage of the node aggregation as a function of the input size

of LUT. This value depends on several other factors (such as the number of bits of the fixed-point representation or the embedding dimension). For the standard 16 bits datatype and 10 embedding dimension this limit is around 16 nodes. For the Belle II detector the number of hits (in 99% of event) is less than 419 so way more than the 16 nodes reached here. By using less than the 10 embeddings dimension and less than 16 bits, it is possible to increase slightly the number of nodes but reaching more than 32 seems unachievable with this method.

## 11 Resource optimized version

As an attempt to deal with enough nodes to incorporate the full detectors ( 400 hits) in a single FPGA, a resource optimized version was developed. In this version, the high LUT use is solved by not fully partitioning arrays so LUT multiplexers are replaced by RAMs. As a consequence, we expect much longer execution time since RAMs can only supply two read ports yielding low parallel access. The main drawback of this method is that the sequential access of elements in the RAMs prevent pipelining the full algorithm and leads to a latency (= execution time) proportional to the size of the input. Indeed, adding mode nodes increase the number of node aggregation and, since they cannot be parallelized, latency. However, the resource limitation is no longer an issue: for 128 nodes and 256 edges, the LUT use is around 17% while the BRAM use remains bellow 1%, latency reach already 5us for a single update step.

## 12 Summary

On the contrary to [3] and [4], the GNN we tried to implement was initially overly big. The first accomplished step was to reduce the size of the network while keeping convergence. After optimising the GNN, it could still achieve reasonable performances (95% efficiency) with 8 neurons per layer and a node embedding dimension of 8. Even after optimisation the algorithm was still bigger than [3] and [4] (due to the fact that the algorithms are not the same).

The time optimized version of the GNN is able to perform a node embedding update in 0.5us while being pipelined at 100ns, but cannot deal with more than 32 nodes. A similar problem is reported by [3] and [4], to solve this issue, [3] offer to split the detector in 128 sectors. This solution is however too expensive for the Belle II project.

On the opposite the resource optimized version can scale to handle many nodes but the latency scales proportionally. It can deal with all the hits of the detector but not in the required time for the L1 trigger. As mentioned in [3] and [4] this slower version cannot be used for the L1 trigger but can be used to accelerate tracking in High Level Trigger (a second trigger system build on top of the Level 1 trigger with less timing requirement).

As a conclusion, even though the context and algorithm are different than [3] and [4] we reach similar results : Using a GNN for a level 1 Trigger is not mature

# Lexicon

- ATLAS : A Toroidal LHC Apparatus. One of CERN's experiment.
- ASIC : Application specific Integrated Circuit.
- BRAM : Block Random Access Memory. See more details with FPGA resources.
- CERN : European Organization for Nuclear Research (Centre Européen pour la Recherche Nucléaire).
- DPS : Digital Signal Processor. See more details with FPGA resources.
- FF : Flip Flop. See more details with FPGA resources.
- FPGA : Field Programmable Gate Array. Can be described as a reprogrammable hardware.
- HDL : Hardware Description Language. Programming language capable of representing hardware comportment (parallel and not sequential). In this project we use Verilog.
- HLS : High Level Synthesis. Method to produce HDL code without writing the code directly but instead by starting from a high level representation of the algorithm (in our case C++) and a set of user defined constraints and parameters (pragma) to automatically generate the HDL code.
- II : Initiation Interval.
- LUT : See more details with FPGA resources.
- Pragma : in our context, compiler directives provided directly in the C++ code that configure and guide Vivado HLS execution.
- Vivado HLS : High Level Synthesis tool produced by Xilinx. Capable of producing HDL from C++.

# Acknowledgement

I would like to thank Mrs Courtin, director of IPHC, Mrs Hu, head of the C4Pi and Mrs Ripp-Baudot, head of the Belle II group in IPHC, for approving and financing my internship. The internship took place in the C4Pi technical group but was funded by the Belle II group. I am gratefull to both groups for their support.

I would also like to thank my supervisors Jérôme and Luca. They knew how to channel the project and unblock complicated situations. I would also like to compliment Luca for his ability to always find my mistakes or oversights, even the most subtle ones. This was sometimes frustrating but it greatly improved the scientific quality of the work.

This work would not have been possible without the help of Benjamin Schwenker who provided the initial algorithm and motivation for this study. I would like to thank him for the time he took to describe the algorithm, explain it and answer my questions.

As far as technique is concerned, I would like to thank most of the team because many of them were able to help me on certain aspects: Christian, Frédéric, Grégory, Xiaochiao, Jean, Claude...

This year was also special because it was a record number of students for the C4Pi platform. It was a pleasure to share this time with other students both from a conviviality and learning point of view. I wish them every success in their thesis/job.

Last but not least, it is very important for me to thank Claude, Christine and Jérôme for all the help they gave me in the beginning of my career. Whether it was in terms of orientation or recommending my application, they certainly did everything they could to help me.

## References

[1] T. Abe et al. *Belle II Technical Design Report.* 2010. DOI: 10.48550/ARXIV.1011.0352. URL: https://arxiv.org/abs/1011.0352.

[2] Valerio Bertacchi et al. "Track finding at Belle II". In: *Comput. Phys. Commun.* 259 (2021), p. 107610. DOI: 10 . 1016 / j . cpc . 2020 . 107610. arXiv: 2003 . 12466 [physics.ins-det].

[3] Abdelrahman Elabd et al. "Graph Neural Networks for Charged Particle Tracking on FPGAs". In: *Frontiers in Big Data* 5 (Mar. 2022). DOI: 10.3389/fdata.2022.828666. URL: https://doi.org/10.3389%5C%2Ffdata.2022.828666.

[4] Aneesh Heintz et al. *Accelerated Charged Particle Tracking with Graph Neural Networks on FPGAs.* 2020. DOI: 10.48550/ARXIV.2012.01563. URL: https://arxiv.org/abs/2012.01563.

[5] R Turchetta et al. "A monolithic active pixel sensor for charged particle tracking and imaging using standard VLSI CMOS technology". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 458.3 (2001), pp. 677–689. ISSN: 0168-9002. DOI: https://doi.org/10.1016/S0168-9002(00)00893-7. URL: https://www.sciencedirect.com/science/article/pii/S0168900200008937.

[6] Kazunori Akai, Kazuro Furukawa, and Haruyo Koiso. "SuperKEKB collider". In: *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* 907 (2018). Advances in Instrumentation and Experimental Methods (Special Issue in Honour of Kai Siegbahn), pp. 188–199. ISSN: 0168-9002. DOI: https : / / doi . org / 10 . 1016 / j . nima . 2018 . 08 . 017. URL: https://www.sciencedirect.com/science/article/pii/S0168900218309616.

[7] Francesco Forti. *Snowmass Whitepaper: The Belle II Detector Upgrade Program.* 2022. DOI: 10.48550/ARXIV.2203.11349. URL: https://arxiv.org/abs/2203.11349.

[8] T. Kuhr et al. "The Belle II Core Software". In: *Comput. Softw. Big Sci.* 3.1 (2019), p. 1. DOI: 10.1007/s41781-018-0017-9. arXiv: 1809.04299 [physics.comp-ph].

[9] Xiangyang Ju et al. "Performance of a geometric deep learning pipeline for HL-LHC particle tracking". In: *Eur. Phys. J. C* 81.10 (2021), p. 876. DOI: 10.1140/epjc/s10052-021-09675-8. arXiv: 2103.06995 [physics.data-an].

```cpp
void edge_network(
    node_embedding_t edge_input[2*DIM_EMBEDDING],
    logit_edge_weight_t &logit_edge_weight
){
    // edge network : MLP
    // must be run on each edge
    layer4_t layer4_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer4_out complete dim=0
    nnet::dense<node_embedding_t, layer4_t, config4>(edge_input, layer4_out, w4, b4); // edge_network.0

    layer5_t layer5_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer5_out complete dim=0
    nnet::relu<layer4_t, layer5_t, ReLU_config5>(layer4_out, layer5_out); // edge_network.1

    layer6_t layer6_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer6_out complete dim=0
    nnet::dense<layer5_t, layer6_t, config6>(layer5_out, layer6_out, w6, b6); // edge_network.2

    layer7_t layer7_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer7_out complete dim=0
    nnet::relu<layer6_t, layer7_t, ReLU_config7>(layer6_out, layer7_out); // edge_network.3

    layer8_t layer8_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer8_out complete dim=0
    nnet::dense<layer7_t, layer8_t, config8>(layer7_out, layer8_out, w8, b8); // edge_network.4

    layer9_t layer9_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer9_out complete dim=0
    nnet::relu<layer8_t, layer9_t, ReLU_config9>(layer8_out, layer9_out); // edge_network.5

    nnet::dense<layer9_t, logit_edge_weight_t, config10>(layer9_out, &logit_edge_weight, w10, b10); // edge_network.6
}
```

```
void node_network(
    node_network_in_t node_network_in[3*DIM_EMBEDDING],
    node_network_out_t node_network_out[DIM_EMBEDDING]
){
    // node network : MLP
    // must be run on each node
    #pragma HLS ARRAY_PARTITION variable=node_network_in complete dim=0
    layer11_t layer11_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer11_out complete dim=0
    nnet::dense<node_network_in_t, layer11_t, config11>(node_network_in, layer11_out, w11, b11); // node_network.0

    layer12_t layer12_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer12_out complete dim=0
    nnet::relu<layer11_t, layer12_t, ReLU_config12>(layer11_out, layer12_out); // node_network.1

    layer13_t layer13_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer13_out complete dim=0
    nnet::dense<layer12_t, layer13_t, config13>(layer12_out, layer13_out, w13, b13); // node_network.2

    layer14_t layer14_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer14_out complete dim=0
    nnet::relu<layer13_t, layer14_t, ReLU_config14>(layer13_out, layer14_out); // node_network.3

    layer15_t layer15_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer15_out complete dim=0
    nnet::dense<layer14_t, layer15_t, config15>(layer14_out, layer15_out, w15, b15); // node_network.4

    layer16_t layer16_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer16_out complete dim=0
    nnet::relu<layer15_t, layer16_t, ReLU_config16>(layer15_out, layer16_out); // node_network.5

    layer17_t layer17_out[DIM_EMBEDDING];
    #pragma HLS ARRAY_PARTITION variable=layer17_out complete dim=0
    nnet::dense<layer16_t, layer17_t, config17>(layer16_out, layer17_out, w17, b17); // node_network.6

    #pragma HLS ARRAY_PARTITION variable=node_network_out complete dim=0
    nnet::relu<layer17_t, node_network_out_t, ReLU_config18>(layer17_out, node_network_out); // node_network.7
}
```

```
void aggregation_edges(
    input_edge edge[2],
    node_embedding_t node_embedding[N_NODE][DIM_EMBEDDING],
    node_embedding_t edge_input[DIM_EMBEDDING]
){
    //edge aggregation for a single edge
    int edge_start = edge[0];
    int edge_end = edge[1];
    for (int i=0; i<DIM_EMBEDDING; i++){
        edge_input[i] = node_embedding[edge_start][i];
        edge_input[i+DIM_EMBEDDING] = node_embedding[edge_end][i];
    }
}

void aggregation_nodes(
    node_embedding_t node_embedding[N_NODE][DIM_EMBEDDING],
    input_edge edge[N_EDGE][2],
    edge_weight_t edge_weight[N_EDGE],
    node_network_in_t node_network_in[N_NODE][3*DIM_EMBEDDING]
){
    // node aggregation for ALL edges
    for (int i=0; i<N_NODE; i++){
        for (int j=0; j<3*DIM_EMBEDDING; j++)
            node_network_in[i][j] = 0;
    }
    for (int e=0; e<N_EDGE; e++){
        int edge_start = edge[e][0];
        int edge_end = edge[e][1];
        edge_weight_t w = edge_weight[e];
        for (int j=0; j<DIM_EMBEDDING; j++){
            node_network_in[edge_end][j] += w*node_embedding[edge_start][j];
            node_network_in[edge_start][j+DIM_EMBEDDING] += w*node_embedding[edge_end][j];
        }
    }
    for (int n=0; n<N_NODE; n++){
        for (int j=0; j<DIM_EMBEDDING; j++){
            node_network_in[n][j+2*DIM_EMBEDDING] = node_embedding[n][j];
        }
    }
}
```

```cpp
void myproject(
    node_embedding_t input_node_embedding[N_NODE][DIM_EMBEDDING],
    input_edge edge[N_EDGE][2],
    node_embedding_t output_node_embedding[N_NODE][DIM_EMBEDDING]
) {

    // pragmas : give guidance to the HLS tool
    // array_partition : tells the tool to split the array along the given dimension
    //                   instead of having a "big" block in memory it can then have
    //                   several small arrays (and thus access them at the same time)
    // pipeline : ask the tool to try pipelining the function in a given cycles number
    //            (it might still fail)
    #pragma HLS ARRAY_PARTITION variable=input_node_embedding complete dim=0
    #pragma HLS ARRAY_RESHAPE variable=edge complete dim=0
    #pragma HLS ARRAY_PARTITION variable=output_node_embedding complete dim=0
    #pragma HLS INTERFACE ap_vld port=input_node_embedding,edge,output_node_embedding
    #pragma HLS PIPELINE II=20


    logit_edge_weight_t logit_edge_weight[N_EDGE];
    for (int e=0; e<N_EDGE; e++){
        node_embedding_t edge_input[2*DIM_EMBEDDING] = {0};
        aggregation_edges(edge[e], input_node_embedding, edge_input); // edge aggregation
        edge_network(edge_input, logit_edge_weight[e]); // edge network
    }


    edge_weight_t edge_weight[N_EDGE];
    #pragma HLS ARRAY_PARTITION variable=edge_weight complete dim=0
    nnet::sigmoid<logit_edge_weight_t, edge_weight_t, Sigmoid_config>(logit_edge_weight, edge_weight); // sigmoid


    node_network_in_t node_network_in[N_NODE][3*DIM_EMBEDDING] = {0};
    aggregation_nodes(input_node_embedding,edge,edge_weight,node_network_in); // node aggregation


    for (int n=0; n<N_NODE; n++){
        node_network_out_t node_network_out[DIM_EMBEDDING] = {0};
        node_network(node_network_in[n],node_network_out); // node network
        for (int j=0; j<DIM_EMBEDDING; j++)
            output_node_embedding[n][j] = input_node_embedding[n][j] + node_network_out[j]; // update
    }
}
```