UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's thesis

Implementation and optimisation of a Graph Neural Network-based track reconstruction pipeline on Intel FPGAs for the ATLAS TDAQ system for HL-LHC

Sara Kjær

Advisor: Alessandra Camplani

Submitted: September 11, 2023

Abstract

The Large Hadron Collider is undergoing the High Luminosity upgrade. With the accelerator upgrade comes an increase in the luminosity of particle collisions, and with that a factor 10 increase in the amount of data collected by the detectors. This puts entirely new demands on the computational power used for particle tracking in the ATLAS detector's TDAQ system.

This project explores the use of graph neural networks (GNNs) for fast and efficient online track reconstruction via implementation on Intel FPGAs. A three-step GNN-based track reconstruction pipeline has already been tested for offline track reconstruction, and is modified to fit the size constraints of an FPGA. We specifically consider FPGAs as the hardware host of the pipeline, due to its parallelism and low power consumption in comparison with GPUs and CPUs.

I explore two main areas of interest regarding the implementation of GNNs on FPGAs. Firstly, the size constraints of Intel FPGAs are studied by obtaining resource estimates for individual steps of the GNN pipeline. An estimate of the appropriate size of the pipeline steps is presented. Secondly, I explore how performance of the pipeline can be maintained whilst reducing its size to fit on an FPGA. I present methods for increasing performance, along with the track reconstruction efficiency obtained for a pipeline suited for FPGA implementation.

Contents

1 Introduction

1.1 LHC, ATLAS and TDAQ

The Large Hadron Collider (LHC) [1] is a particle accelerator built to produce high-energy particle collisions, allowing physicists to study the behaviour of fundamental particles under extreme conditions. The LHC is currently undergoing an extensive upgrade, which will increase its instantaneous luminosity[1] up to $\mathcal{L} = 7.5 \times 10^{34}$ cm^{-2}s^{-1} by 2029. For comparison, the luminosity of the on-going LHC Run 3 is expected to reach a peak luminosity of 3×10^{34} cm^{-2}s^{-1}. This upgrade is called the High Luminosity LHC (HL-LHC) [3]. With it comes a ten-fold increase in the amount of data produced at collision points.

Along the LHC, four main detectors record data from the many collisions. I will focus on the ATLAS experiment [4] in this report, which is a 46-by-25 metre cylindrical detector. It consists of multiple sub-detectors, which are arranged in layers around the collision point. The layers consist of "cells", which are the smallest units of the detector. As charged particles from collisions travel through the detector, they deposit energy in the cells. The signal from these energy deposits is then used to reconstruct particle trajectories ("tracks") through a process called tracking. The steps of the tracking procedure are described in Ref. [5] and are, in summary:

1. **Clustering.** Neighbouring cells where energy has been deposited are registered as a "cluster". Clusters are then converted to 3D space-points, which are determined using the signal from each cell and the detector geometry.

2. **Pattern recognition.** Candidate tracks are initially reconstructed through pattern recognition algorithms. This step uses as input the space-points from the first step. I will from now on refer to these space-points as "hits" in the context of pattern recognition algorithms.

3. **Track fitting.** Duplicate tracks are then removed and, as a final step, the remaining tracks are evaluated with a linearised χ^2 fit.

Since the amount of data produced in the experiment is greater than the amount of data that can be sent to permanent storage, real-time decisions have to be made about which collision events to keep for further analysis and which to discard, based on data from the ATLAS sub-detectors. In the ATLAS experiment, the Trigger and Data Acquisition (TDAQ) system [6] is responsible for making these decisions. Since it operates in real-time, the TDAQ system is referred to as an **online** system for processing ATLAS data. Once the data is sent to permanent storage, **offline** systems analyse the data further. The 3-step tracking procedure outlined above is performed within TDAQ.

With the HL-LHC upgrade, a new sub-detector, the "Inner Tracker" (ITk) [7], will be developed and used to cope with the increased luminosity of particle collisions. Furthermore, the capacity of the TDAQ system will be adjusted in order to keep up with the amount of

[1] Instantaneous luminosity refers to the rate at which particles pass through a certain cross-section [2].

data it has to process. For this, a new "Event Filter" [8] will be added, which will process data coming from the sub-detectors. The new layout of the TDAQ system is presented in Fig. 1.

There are ongoing studies into how the Event Filter can produce track candidates from ITk data in the HL-LHC. In fall 2025, decisions will be made by the Event Filter tracking group regarding the hardware and algorithms used for tracking. In this thesis, I focus on the second step of the tracking process outlined above. Here I explore the use of Graph Neural Networks (GNNs) for pattern recognition, and how they can be implemented onto Field Programmable Gate Arrays (FPGAs).

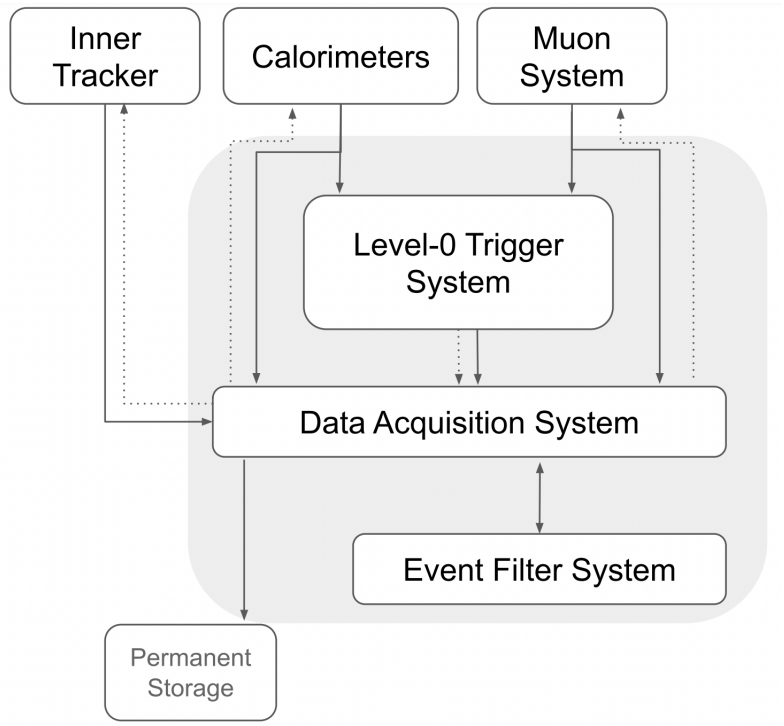


Figure 1: An overview of the TDAQ upgrade design [9]. The Event Filter receives data from the Data Acquisition System (DAQ), which acquires its data from the sub-detectors of ATLAS. The Event Filter sends its data back to DAQ before it is sent to permanent storage. The solid lines represent the readout data, and the dashed lines the Level-0 accept signal.

1.2 Graph neural networks

GNNs are a type of neural network, which are applied to data in the form of graphs [10]. A graph is composed of a set of nodes and a set of edges that connect node pairs. In our case, nodes represent hits and edges represent track segments, i.e. paths that a particle could have taken in the detector. This is illustrated in Fig. 2.

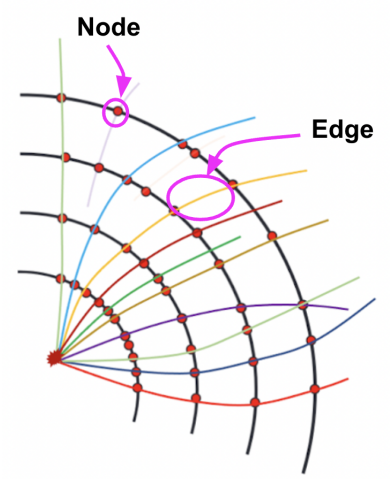


Figure 2: Nodes represent hits in the detector layers, and edges represent paths between hits in consecutive layers.

In graphs, information can be stored on three levels: the node-level, the edge-level, and the graph-level. A GNN applied to a graph can consequently make predictions on either of these three levels. In this project, I use a GNN to predict whether track segments (paths between two hits) are true or false. Since edges represent track segments, predictions are made on the edge-level. In Section 3.2.1, I elaborate on the GNN concept, and how I implemented it as part of a tracking algorithm.

1.3 FPGAs

An FPGA is a semiconductor device, which can be configured based on the demands of the algorithms it executes. It consists of a matrix of configurable logic blocks and programmable interconnections, meaning that the hardware can be customised for very specific applications [11].

Aside from being flexible in terms of design, it is also a device with a high degree of parallelism. Since neural networks, such as GNNs, contain many repetitive operations, parallelising the algorithms on an FPGA can significantly reduce their latency. This is ideal in a set-up like the EF, which sees a high frequency of hit data and has to make rapid decisions about the storage of collision events.

There is a range of different FPGAs on the market: the companies AMD and Intel are the biggest manufacturers of FPGAs. In this project, I target an Intel device: the Stratix 10 GX 2800 FPGA [12]. Further details about the device and its properties are given in Section 4.

2 Data

2.1 From hits to track candidates

When looking at the pattern recognition step of the tracking procedure outlined above, the core task of the algorithm is to group hits in a particle detector together to form tracks. Tracks represent the path a particle has travelled through the detector, and each hit in a given track belongs to the same original particle. In order to convert hit data to candidate tracks, I use a three-stage pipeline. The three stages are **graph construction**, **edge classification** and **track reconstruction**. The pipeline is illustrated in Fig. 3, which I developed in Python, using PyTorch for its machine learning components. In this thesis, I will refer to the pipeline as the **track reconstruction pipeline**. It will be elaborated upon in Section 3.

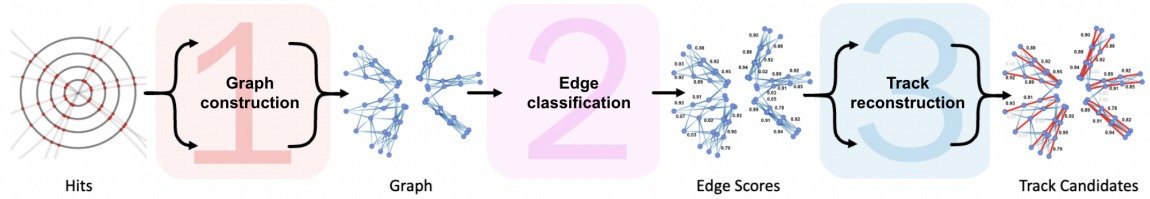


Figure 3: Overview of the three-stage track reconstruction pipeline from hits to track candidates [13].

Before covering the details of the pipeline, I will describe the data that was used during the project. The datasets contain hit data from simulated collisions at the Large Hadron Collider, and I implemented two separate datasets in the pipeline. These will be referred to as "TrackML" and "ITk". The TrackML dataset is a simplified dataset and was initially used to gain some familiarity with the pipeline and to obtain some initial results. The ITk dataset, which bears more resemblance to the data we expect to see in the real ITk detector, was then used to get more realistic performance estimates. Due to differences in the data formats, I used a separate pipeline for each of the datasets. They both, however, follow the same three-stage structure and use some of the same methods.

2.2 Datasets

2.2.1 TrackML

This dataset was retrieved from the Kaggle competition "TrackML Particle Tracking Challenge" [14]. I used the sample dataset, which contains 100 independent collision events. All events are simulated measurements of proton-proton collisions. The pile-up[2] of this dataset is $\mu = 200$, which is the value we expect to see at the HL-LHC.

[2]Pile-up refers to the amount of simultaneous collisions in every bunch crossing in the detector [15].

The hit data is shaped by the geometry of the detector. The simulated detector is a generic detector, which is cylindrical and centered around the collision point [16]. It consists of multiple layers, in which the hits are recorded. Layers are divided into smaller so-called "modules", and are grouped into "volumes" to define regions of the detector. All hits are recorded within the detector layers. Fig. 4 shows a side-view of the detector geometry used for the TrackML dataset.

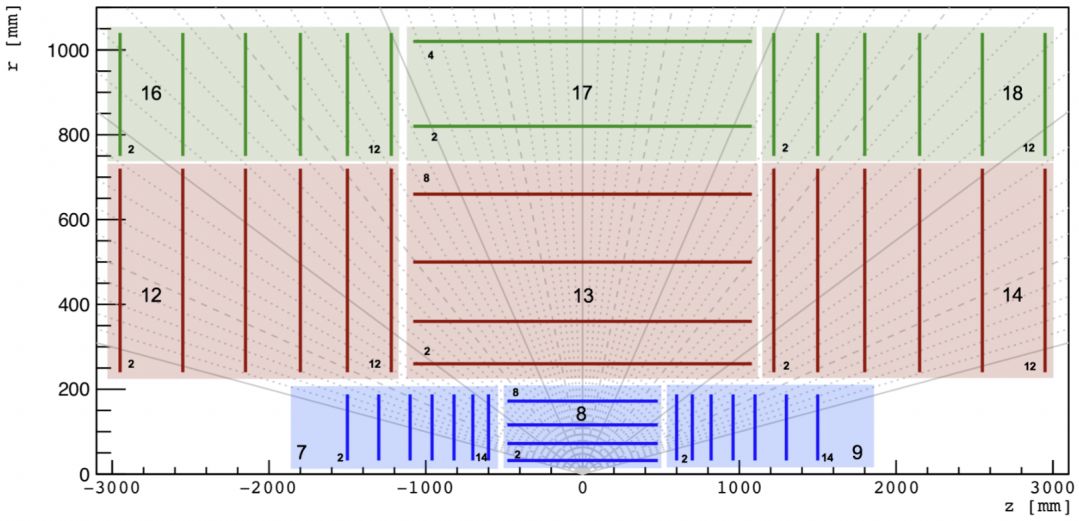


Figure 4: A side-view of the detector geometry used for TrackML data. The vertical and horizontal lines represent layers, and the coloured sections show the volumes. Individual modules are not shown in this plot [14].

In the dataset, the hits are represented by their spatial position, which is denoted by x, y and z coordinates. They are further tagged with a hit ID, along with identifiers for the layer and volume they appear in the detector. A "truth" dataset contains the mapping between each hit and its generating particle. Here, information is provided about the particles' initial momentum, charge and the number of hits generated by each particle. From this information, true tracks can be inferred by connecting hits belonging to the same particle. Two hits are connected if the second hit is further away from the collision point than the first, and if the second hit is the hit from that particle which is closest to the first. The set of edges that connect the hits is then what we define as the true track. The truth data is used during training of the pipeline's neural networks to measure the performance of the models.

Compared to the data we expect to see in the implemented ITk detector, the TrackML data is quite simplified. First of all, secondary particles have been removed, i.e. particles created by other particles away from the collision point. Secondly, noise makes up just 16% of the dataset [17]. Here, noise denotes any hit in the detector which does not belong to

a particle. This is a combination of mis-constructed spacepoints, and particles with a p_T below 100 MeV. When I trained the pipeline on this dataset, some further cuts were also applied to the data. These will be described in Section 5.3.

Due to its simplicity, I used the TrackML dataset as a "toy" dataset to gain some familiarity with the methods used in the pipeline during the early stages of the project. It was used to obtain some initial results for track reconstruction efficiency, and for the first attempts at FGPA implementation. I developed the initial pipeline based on code written for the paper in Ref. [18]. I then modified the pipeline using pieces of code developed by the Exa.TrkX group [19]. The pipeline will be described in Section 3.

2.2.2 ITk

This dataset was retrieved from the examples provided in the GNN4ITk "Common-Framework" [20–23]. It contains hit data from simulated $pp \to t\bar{t}$ events with a pileup of $\mu = 200$, which have been generated with the Athena framework [24].

The ITk geometry is illustrated in Fig. 5. The detector is here viewed from the side, where the red components represent the pixel sub-detector, and the blue components represent the strip sub-detector.

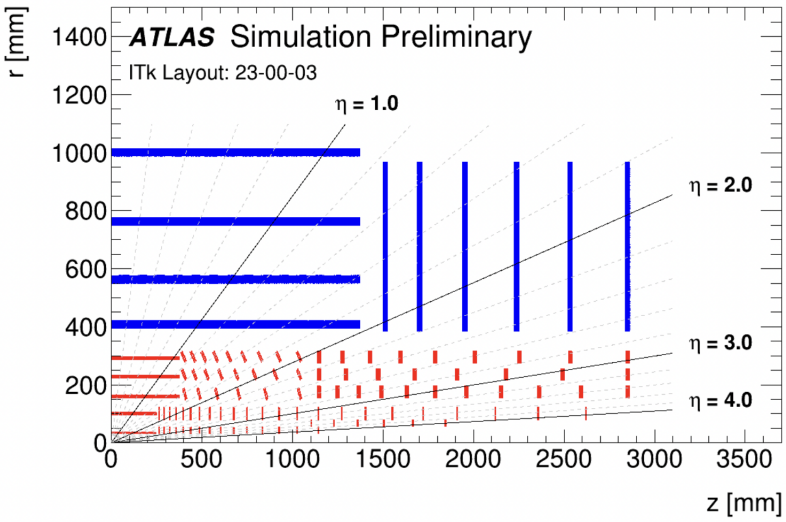


Figure 5: A visualisation of the ITk detector [25].

The spatial positions of the hits are also denoted by x, y and z coordinates, and their corresponding polar coordinates r, ϕ and z are provided. Similar to the TrackML data, hits are tagged with unique IDs, along with identifiers for their detector layer and volume. Truth data is also provided, mapping hits to their generating particles.

The ITk data is, compared to TrackML, more realistic. The events are simulated using real ITk geometry, and fewer cuts have been made to the data. Compared to TrackML, there

are more tracks, and hence more hits, in this dataset. Furthermore, secondary particles are not omitted from the dataset, and noise is, on average, 55%. Table 1 contains a comparison between the two datasets.

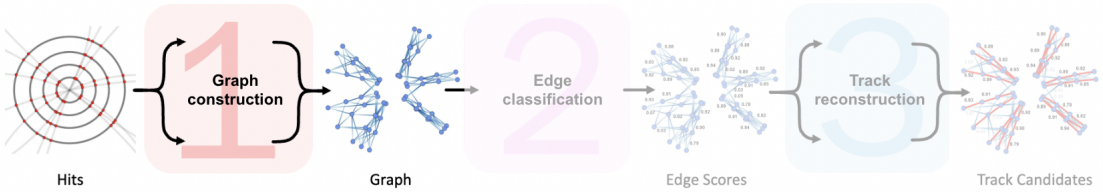
	TrackML	**ITk**
Noise amount (avg.)	16%	55%
Secondary particles	No	Yes
Avg. number of hits	$\approx 110,000$	$\approx 310,000$
Avg. number of tracks	$\approx 9,000$	$\approx 15,000$
Pile-up	200	200

Table 1: Comparison of the TrackML and ITk datasets [17].

The pipeline code for processing ITk data is based on the codebase from the GNN4ITk ComonFramework [20], which was originally developed for offline track reconstruction in the ITk. During this project, I have updated various parts of the code in order to target online implementation of the framework. Similar to the TrackML dataset, some cuts were made on the data during training. These will also be described in Section 5.3.

3 Track reconstruction pipeline

3.1 Stage 1: Graph construction



The goal of the graph construction stage is to convert hit data into graphs. The nodes of the graph are already given from the hit data, so the main job of the graph construction algorithm is to determine which nodes to connect in the graph. In theory, the graphs could be constructed as fully-connected graphs with an edge connecting each node pair in consecutive layers. With an order of $100,000$ nodes in each graph, however, this would generate graphs that are too large to be processed on both FPGAs and GPUs, and too impure for the GNN to make accurate predictions. Therefore, this stage of the pipeline is used to exclude any node connection that is unlikely to belong to a true track, as illustrated in Fig. 6.

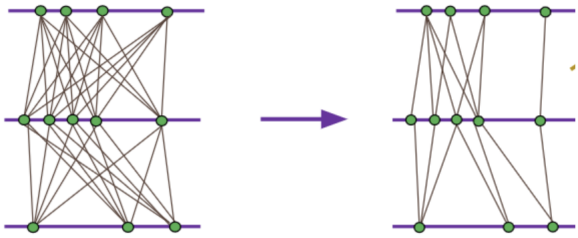


Figure 6: The graph construction stage chooses the edges that are most likely to belong to true tracks. Instead of connecting every node pair in consecutive layers, graph sizes are reduced by constructing chosen connections only [17].

Maintaining a high efficiency (true edges constructed/total true edges) in the graph construction stage is important, since any lost track segment cannot be reconstructed further down the pipeline. This can potentially result in complete failure to reconstruct the given track. Meanwhile, maintaining a high purity (true edges/constructed edges) is important, since it reduces the size of the graphs, while also improving the performance of the GNN stage in the pipeline. Precise definitions for measurements of efficiency and purity will be given in Section 3.4.

During the project, I implemented two methods for graph construction: **Heuristics** and **Metric Learning**. The Heuristic method was initially used for the TrackML dataset. The Metric Learning method was first applied to TrackML data, and later used with ITk data. These two methods will be described in the following sections.

3.1.1 Heuristic method

The Heuristic method connects hits in layer n to hits in layer $n+1$, and applies some filters on the edges to determine specifically which hits to connect. For each edge, the parameters $z0$ and ϕ_{slope} were first determined. They were calculated as follows:

$$\phi_{slope} = \frac{d\phi}{dr} \tag{1}$$

$$z_0 = z_n - r_n \frac{dz}{dr} \tag{2}$$

where r, phi and z are the polar coordinates of the hit point, n represents the layer number, and where:

$$d\phi = \phi_{n+1} - \phi_n \tag{3}$$

$$dr = r_{n+1} - r_n \tag{4}$$

$$dz = z_{n+1} - z_n \tag{5}$$

The edges were then filtered based on their z_0 and ϕ_{slope}. Only edges within the following values were constructed in the graph:

- $-320 \leq z_0 \leq 520$
- $|\phi_{slope}| \leq 0.0055$

I chose these values based on an analysis of the z_0 and ϕ_{slope} distributions of the dataset made in Ref. [26].

3.1.2 Metric learning

I applied the Metric Learning algorithm to both TrackML data and ITk data. This is also the algorithm from which I present resource estimates in sections 6 and 7. It uses a Multi-Layer Perceptron (MLP), which is a type of feed-forward neural network [27] consisting typically of a number of linear layers and activation functions. The MLP embeds the hits into a latent space. Through training, it learns to embed the hits such that hits from the same track are placed close to each other in latent space. This process is illustrated in Fig. 7.

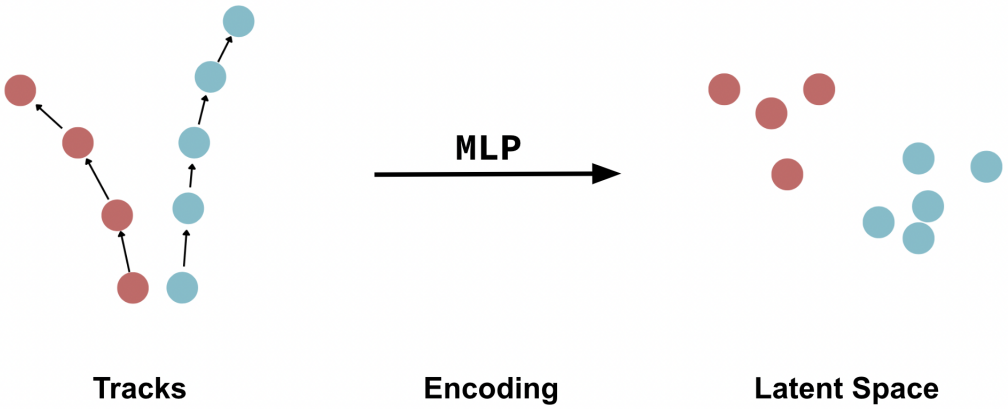


Figure 7: Hits that belong to the same track are encoded such that they are close to each other in latent space.

The architecture of the MLP that I used for the project is presented in Fig. 8. It contains an input layer, 3 hidden layers, an output layer, and a hidden tanh activation between all linear layers.

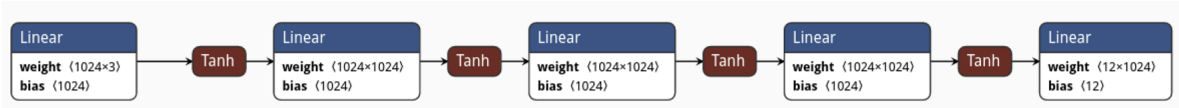


Figure 8: Standard architecture of an MLP used in Metric Learning. The weights and biases are here of dimension 1024, however I adjusted this number during the project to target FPGA implementation.

The trained MLP can then be applied to a new set of hits (referred to as "inference"). It encodes the hits into latent space, and a Fixed Radius Nearest Neighbours (FRNN) algorithm is then applied to find out which hits are closest to each other in latent space, and could hence belong to the same track. Here, a "FRNN radius" is specified, and for each hit, an edge will be established with all hits found within the radius. As a result, graphs are produced where nodes represent the hits, and edges the connections found with the FRNN algorithm. I use the Pytorch Geometric function "radius" [28] for executing the FRNN algorithm. The process is illustrated in Fig. 9.

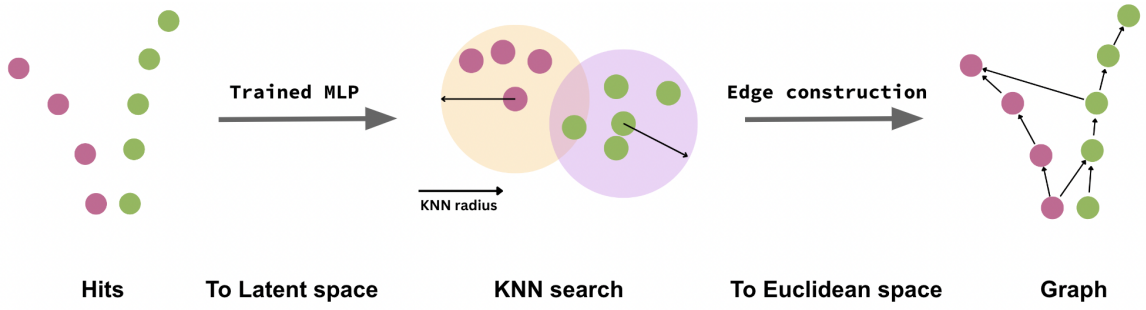
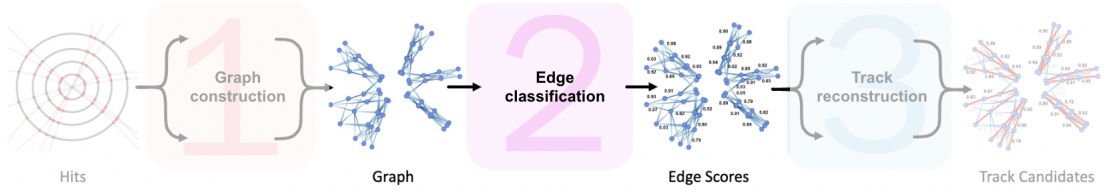


Figure 9: To create a graph from a set of hits, the hits are encoded with the trained MLP, and a FRNN search determines which edges to build.

3.2 Stage 2: Edge classification



The edge classification stage assigns an "edge score" to each edge in the graphs produced in the previous pipeline stage. This score represents the likelihood of the edge being a true edge, and the scores range between 0 and 1. A cut is then applied to the edges in the graph, such that only edges above a given score are predicted to be true. For this stage, I use a GNN.

3.2.1 Graph Neural Networks: An Overview

The core concept of GNNs is the idea of "message passing" where information is passed between nodes along their connecting edges [10]. A generic GNN algorithm has the following steps, and is illustrated in Fig. 10:

1. **Update nodes.** An MLP is applied to the nodes, updating the properties of each node.

2. **Message passing.** Information is passed along edges to neighbouring nodes. This is also done with an MLP.

3. **Aggregation.** "Messages" are aggregated at each node using a permutation-invariant aggregation function.

4. **Classification.** Finally, a classifier is used to make edge-level predictions, i.e. assigning an edge score to each edge.

The first three steps can be repeated to include multiple message passing steps. For each message passing step in the algorithm, the degree of information contained at each node increases. For a GNN with one message passing step, each node will contain information from their first neighbours. Including two message passing steps allows nodes to receive information from their neighbour's neighbours and so forth.

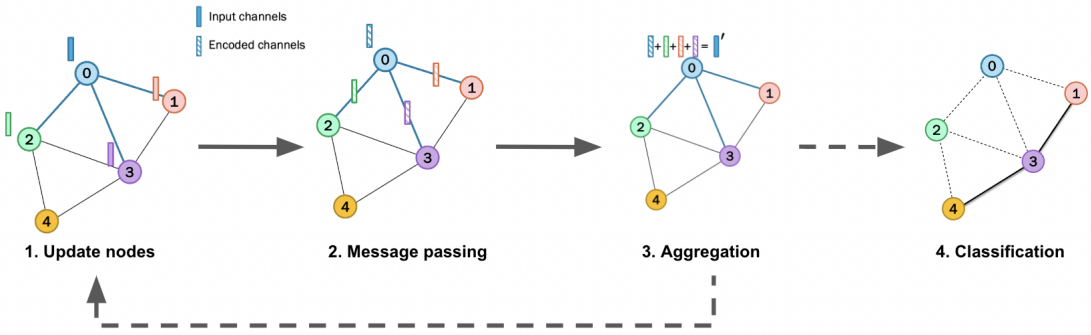


Figure 10: A visualisation of a generic GNN algorithm [17].

In the pipeline, I used a specific type of GNN called an "Interaction Network" [29]. Here, an extra step is added to the generic message passing algorithm, namely an "edge network". This serves the purpose of updating edge features after updating the node features, which allows neighbouring nodes to form unique relationships. Adding this feature to the GNN improves the quality of the edge-level predictions made on the graphs.

3.2.2 GNN architecture

The GNN from the track reconstuction pipeline consists of a set of "networks" which each serve a purpose in regards to either the nodes or the edges of the graph. Each network is based on an MLP, which has two linear layers and a hidden ReLU activation between the layers. This is illustrated in Fig. 11.

The networks are:

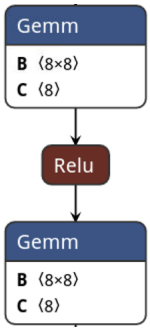


Figure 11: The MLPs used in the GNN consist of two linear layers and a ReLU activation function. The dimensions of the layers can be varied.

- `node_encoder`: encodes node information into latent space.
- `edge_encoder`: encodes edge information into latent space.
- `node_network`: computes new node features.
- `edge_network`: computes new edge features.
- `output_classifier`: classifies edges and outputs their scores. This network has one extra linear layer in its MLP.

Fig. 12 illustrates the architecture of the GNN I used, containing the networks outlined above. The general architecture was the same for both the TrackML and ITk pipeline. Nodes are first encoded into latent space. Edges are then encoded, using information from the encoded nodes. This is where the "messages" are produced. At this point, the message passing algorithm begins. The encoded edges are then aggregated at each node. The aggregated messages, along with the encoded nodes, are run through the node network to update the node features. These, along with encoded edges, are run through the edge network to update the edge features. The output from the node and edge networks can be used as input into another message passing step, or into the output classifier to produce edge-level predictions.

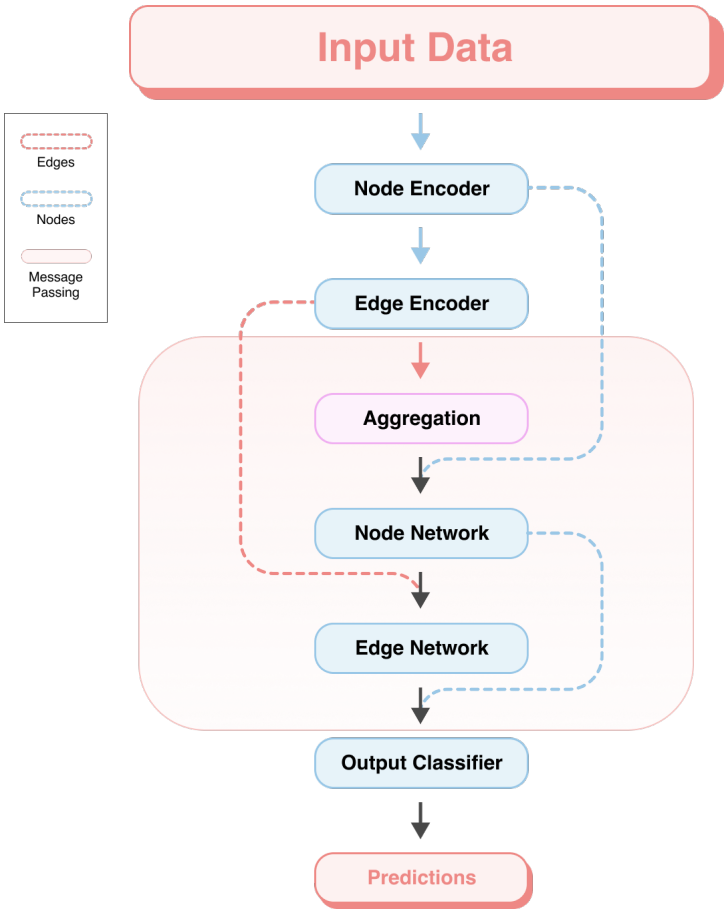
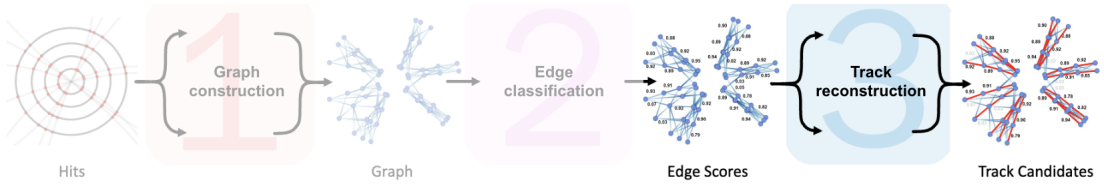


Figure 12: Architecture of the GNN used in the track reconstruction pipeline.

3.3 Stage 3: Track reconstruction



Once edges have been classified, the graphs enter the final stage of the pipeline: track reconstruction. Here, I apply an algorithm to the labeled graphs to predict which edges form tracks. Each edge in an identified track is given the same tag, and this information is output from the pipeline as the track candidates. I implement two different methods for track reconstruction: a **connected components** method, and a **walkthrough** method.

3.3.1 Connected Components

The Connected Components algorithm identifies which paths within the graph are connected. Only edges with a score above 0.8 are considered by the algorithm. This is the threshold found to produce the highest track reconstruction efficiency. The algorithm and how it is applied to graphs is visualised in Fig. 13. Here, I used the SciPy Sparse function `connected_components` [30, 31]. This algorithm was used when running the pipeline on a GPU, and was used for both TrackML and ITk data.

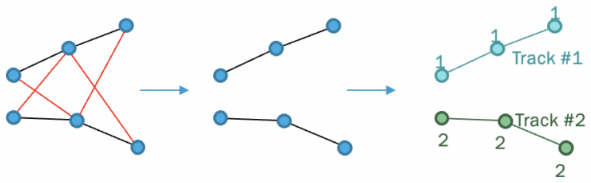
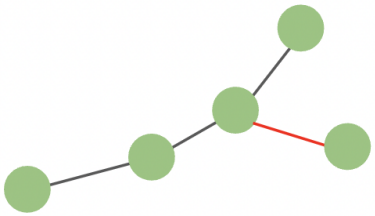
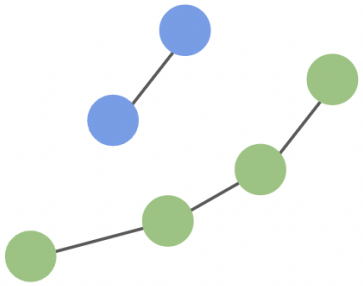


Figure 13: The Connected Components method identifies distinct series of hits, which produce the track candidates [17].

The Connected Components algorithm is very fast and generally gives quite good performance. There are, however, a few cases where it fails to produce tracks that are a single line of connected hits seeded in the innermost detector layer. Fig. 14 demonstrates how such tracks may look. In Fig. 14a, a "branching" track connects the main track (black edges) to another hit (red edge), which is not a part of the true track. In Fig. 14b, the track is very short and not seeded in the innermost detector layer. Such cases will still be considered tracks by the Connected Components algorithm. A solution to this could be to use the Walkthrough algorithm, described in the following section.



(a) Edges form a branching track, where only one of the two right-most edges can be true.



(b) A lonely edge (blue), which is not seeded at the collision point.

Figure 14: Examples of odd types of tracks produced by the Connected Components algorithm.

3.3.2 Walkthrough

The Walkthrough method, on the other hand, traverses the graph edge by edge. It identifies a starting node ("seed") - in our case any node in the innermost detector layer - and chooses the edge with the highest score to traverse along to the next node. This process is continued until there are no more edges reaching into the next detector layer. The set of traversed edges and nodes are then returned as a track candidate, and the process is repeated for other seeds. It is illustrated in Fig. 15.

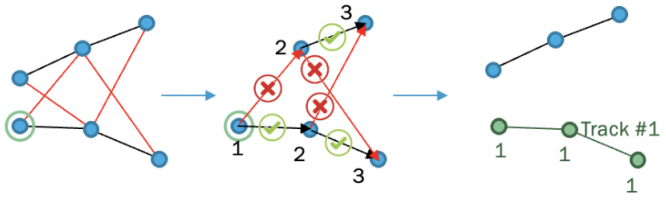


Figure 15: The Walkthrough method constructs track segments by traversing the graph edge by edge [17].

The algorithm will by its nature always produce tracks that are a single line of hits seeded in the innermost layer, and thus avoid constructing tracks of the types presented in Fig. 14. Nevertheless, it is very slow compared to the Connected Components algorithm, and tends to give a lower track reconstruction efficiency.

The Walkthrough method was used for the FPGA implementation of the track reconstruction stage, and was tested on the TrackML dataset on a GPU. While the Connected Components algorithm performs well on a GPU, it contains many loops, which is not ideal for FPGA implementation. The walkthrough method was developed in VHDL language, and will be presented in Section 4.6.

3.4 Testing methods

To test the performance of the three pipeline stages, I measured various metrics. This section covers the definitions of any performance metrics I used.

3.4.1 Truth definitions

Tracks that can be inferred from the "truth" file of the datasets are referred to as "true tracks". As edges represent segments of tracks in the pipeline, a segment of a true track is referred to as a "true edge". A constructed edge that matches a true edge is a "true positive" edge. A constructed edge that does not match a true edge is a "false positive" edge. Correspondingly, an edge connection which is not present in the graphs, but does correspond to a true edge is a "false negative" edge. An edge connection not present, and which also does not match a true edge is a "true negative" edge.

3.4.2 Efficiency and purity

The truth definitions above are used to define the two main performance metrics: efficiency and purity. During the first two stages of the pipeline, these are measured on an edge level. This is the "edge-wise" efficiency and purity, which are defined in the box below. In the following sections, the edge-wise element will be implicit, unless otherwise stated.

Performance definitions

Edge-wise efficiency
Number of true positive edges / number of true edges
Edge-wise purity
Number of true positive edges / number of constructed edges

3.4.3 Signal

There are certain particles that are, from a physics point-of-view, more interesting to reconstruct than others. These are called signal particles, and they produce signal tracks. I use the concept of signal particles when evaluating the performance of the ITk pipeline. To measure the efficiency in reconstructing signal particles, some conditions are defined. A signal particle leaves at least 3 hits in the detector, it has a p_T above 1 GeV, is not an electron and is a primary particle, meaning that it was produced at the collision point. As such, the signal efficiency is the efficiency in constructing edges that belong to signal tracks.

3.4.4 Particle reconstruction and track matching

To measure the performance of the entire track reconstruction pipeline, it is necessary to quantify how many particles were reconstructed by the tracks produced, and how many of the reconstructed tracks correspond to - or "match" - a particle. Definitions of when a particle is reconstructed and when a track is matched can vary across tracking efforts. For this project, I use the "ATLAS matching" convention, which is as follows:

ATLAS matching

Particle reconstruction
A particle is reconstructed if at least one track is matched to it.
Track matching
A track is matched if over X amount of hits belong to a single particle. Here, X denotes the "matching fraction", described below.

I set the matching fraction to 50% for this project, meaning that above half of the hits in a track must belong to the same particle for the track to be matched. Furthermore, only signal particles were considered in the performance evaluation of the model. The definitions for matching and reconstruction above are derived from the Exa.TrkX's guide to matching, which can be found in Ref. [32]. When measuring the performance of the full track reconstruction pipeline, it is conventional to report the efficiency and the fake rate [33]. These are reported for the "selected" particles and tracks, which are particles and tracks that satisfy certain selection criteria. For this project, the selection criteria were i) particles that have at least 3 hits, and ii) tracks consisting of at least 3 hits. Efficiency and fake rate are then defined by:

Track reconstruction performance definitions

Track reconstruction efficiency
Number of reconstructed selected particles / Total number of particles

Track reconstruction fake rate
1− (Number of matched selected tracks / Total number of selected tracks)

3.4.5 Target efficiency

The goal is naturally to have a track reconstruction algorithm that is as efficient as possible. Fig. 16 shows an efficiency plot of a CPU-based tracking demonstrator algorithm presented in the Event Filter tracking amendment to the TDAQ upgrade design report. The simulated data was similar to the data I used in this project: it uses ITk $pp \to t\bar{t}$ events with a pile-up of $\mu = 200$, and reports the efficiency for particles with $p_T > 1$ GeV. The efficiency is, on average, around 90%. With this demonstrator as a base-line, my goal is to present a track reconstruction pipeline that significantly improves the efficiency beyond the 90% reported.

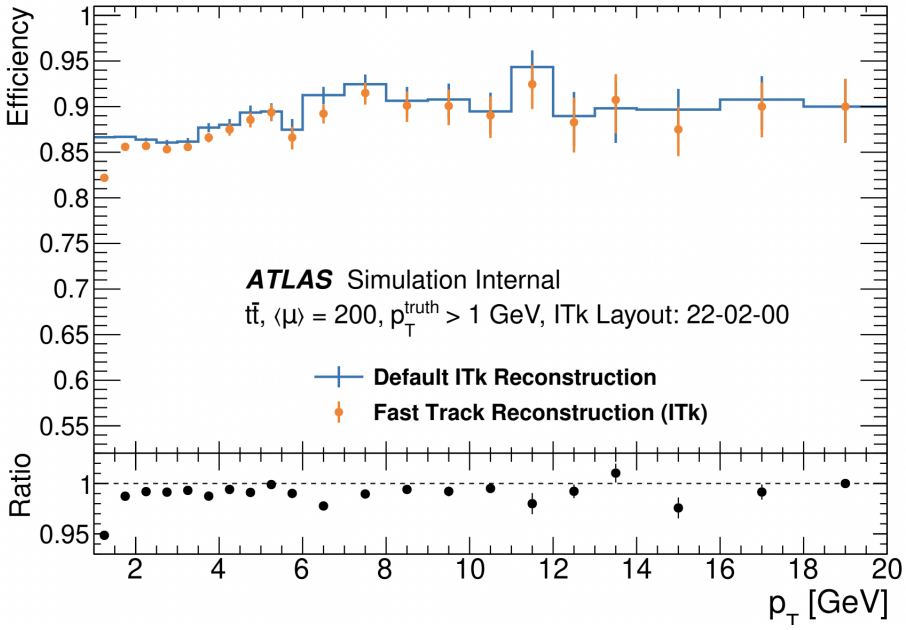


Figure 16: The efficiency of a CPU-based tracking algorithm [8].

4 FPGA implementation

Algorithms are usually implemented in FPGAs with a hardware description language [34]. Integrated development environments (IDEs), like Intel Quartus [35], however also support code written in high-level synthesis (HLS) [36], which is similar to C languages. IDEs are used for FPGA design, and the underlying software produces resource estimates for the implemented algorithms.

I translated some parts of the track reconstruction pipeline from Python to HLS, whereas one part was implemented directly in the hardware description language, in this particular case in VHDL [37]. With the HLS and VHDL code at hand, I produced resource estimates to gauge how much space the model is expected to take up on the device.

4.1 Device

In this project, I target the device Stratix 10 GX 2800 from Intel [12]. An Intel FPGA was chosen due to its potential integration with Intel OneAPI [38], which is a toolkit for heterogeneous computing. The Stratix 10 device was chosen because it was the largest of the Intel devices available with the translation tool HLS4ML, which is described below. FPGAs consist of different building blocks, and I monitored the usage of these when implementing machine learning models. The resources available on the target device are specified in Table 2. Digital Signal Processing blocks (DSP blocks) perform operations such as matrix multiplications. Adaptive Logic Modules (ALMs) are logic elements typically consisting

of Look-Up Tables (LUTs) and Flip Flops (FFs), which are used to implement and store logic operations. Random Access Memory (RAM) is used for storage and retrieval of larger amounts of data. On Intel FPGAs, this is referred to as "M20K" [3]. In the results section (Section 7), I will refer to M20Ks as just "RAM blocks". In Fig. 17, the layout of the FPGA building blocks are illustrated.

Resource	**DSP blocks**	**ALMs**	**M20Ks**
Availability	5,760	933,120	11,721

Table 2: Resource specifications of the Intel Stratix 10 GX 2800 FPGA [40].

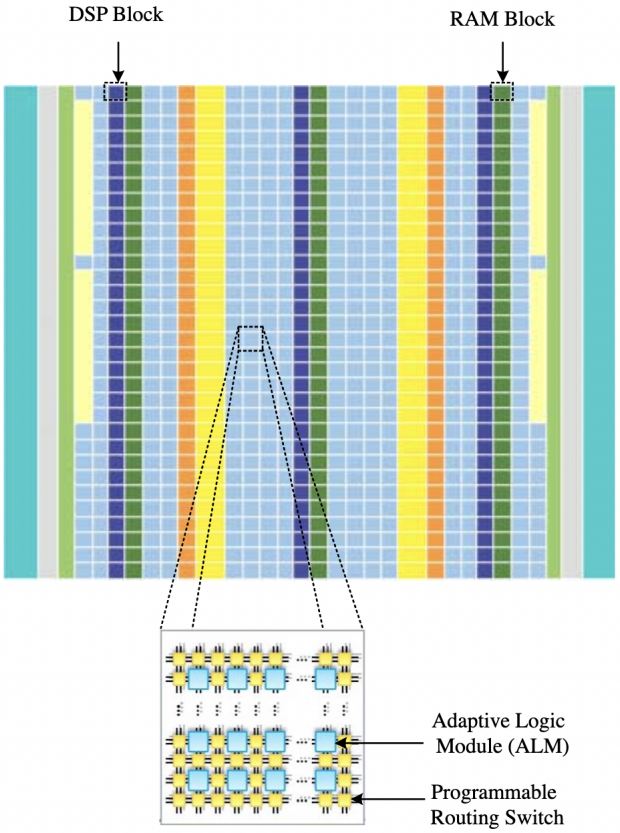


Figure 17: An example overview of the building blocks of an Intel FPGA [11].

[3] An "M20K" is a memory block containing 20,480 programmable bits [39].

4.2 HLS4ML

I translated the machine learning parts of the pipeline code to HLS with the framework HLS4ML ("High Level Synthesis For Machine Learning") [41, 42]. This is an open-source Python library built particularly for translating neural networks to HLS, which can then be used with simulation tools such as Intel Quartus or AMD Vivado. The framework is an ongoing project by the Fast Machine Learning Lab, and so there are certain types of ML models or operations that are not supported yet, and could therefore not be translated. Any unsupported features I came across will be pointed out during this section, and where there was a work-around, this will also be presented.

4.3 Workflow

The general workflow I used for converting models with HLS4ML and obtaining resource estimates is as follows:

Conversion to ONNX

The models, which are written with PyTorch, were as a first step converted to ONNX ("Open Neural Network Exchange") [43]. ONNX is a model format, which is used for interoperability between different machine learning frameworks. At the time of developing this project, HLS4ML did not support conversion of PyTorch models with a Quartus backend, which was needed to target the particular Intel FPGA. As a workaround, the models were first converted to ONNX using `torch.onnx.export`, before I translated them with HLS4ML.

Configure model

The model is configured to HLS4ML. Configuration functions exist for different model formats - here, I use the `config_from_onnx_model` function. The precision of the converted model is also specified here, where I use a fixed-point representation, meaning that weights are stored with a fixed number of decimals before and after their decimal point. This representation is common for programs implemented on FPGAs. The precision was set to `ap_fixed<18,8, AP_RND>` meaning that values contain 18 bits in total, with 8 bits reserved for values before the decimal point, and 10 after the decimal point. `AP_RND` indicates that the weights were rounded to the nearest representable value in case of overflow. I determined the specific precision based on an examination of the accuracy of the HLS models' prediction compared to the prediction of the original PyTorch model.

Convert model

I converted the models to HLS using the `convert_from_onnx_model` conversion function in HLS4ML. There is a range of inputs to this function, which are described below.

- Model (the ONNX model)

- Backend (set to "Quartus" since we are targeting an Intel device)
- Project name
- Config file (from the previous step)
- Output directory (where the project folder will be saved)
- Input data (testbench data for running the HLS synthesis)
- Output data (testbench data for comparison)
- Part (the target device - here, Stratix10 GX)

HLS compilation

A makefile is produced during the conversion towards HLS. When running the makefile, the following will happen:

- **HLS synthesis:** The HLS code is synthesised, and the model is tested on testbench input data.
- **HLS report:** From the HLS synthesis, some initial resource estimates are produced. These are logged into an HLS report.
- **Quartus project:** A Quartus project is generated, which can be used for the Quartus compilation to obtain final resource estimates.

Quartus compilation

I open the generated project into Intel Quartus, and compile it. During compilation, the FPGA design for the model is synthesised, as well as implemented (place and route) in hardware: resource estimates are produced as part of this process. The estimates produced by Quartus are considerably different from the ones generated with HLS. Quartus resource estimates are produced from a compilation that is targeting hardware, whereas HLS estimates come from compiling more C-like code. As such, the Quartus estimates are more reliable than the HLS estimates. The difference between the two types of estimates will be elaborated further upon in Section 7.1.5.

4.4 Stage 1 translation

From the first stage of the pipeline, I translate the MLP used in Metric Learning. The MLP was saved as a checkpoint after training, and I converted it to HLS using a Python script. The original MLP from the CommonFramework normalises each hidden layer using the function "LayerNorm". Normalising weights during training typically allows the model to learn faster, since learned weights are more stable. This function could not be converted with HLS4ML, and so was omitted from the model - both during training and during

conversion to HLS. Removing the layer meant that models needed to train for more epochs before predictions stabilised. In Fig. 18, I plot the validation loss function, which is an indication of a model's performance, for an MLP with and without the LayerNorm function applied. Here we see that the model without LayerNorm reaches approximately the same amount of loss as the model with after around 80 epochs. After this point, the two models are comparable in performance.

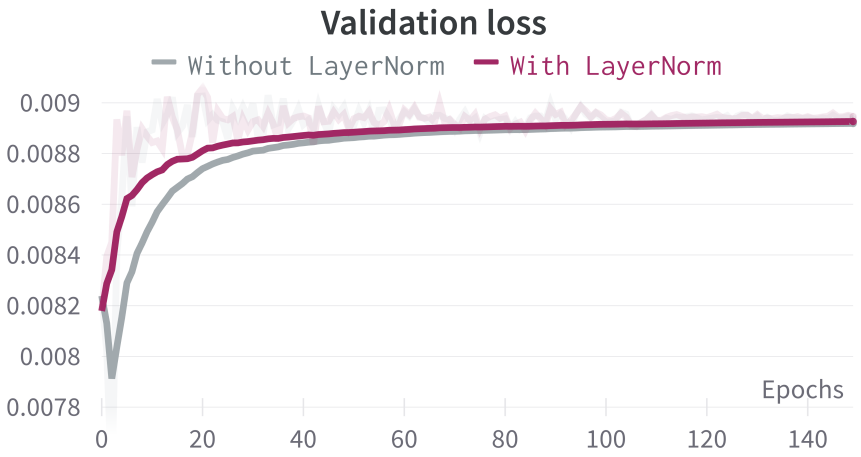


Figure 18: Comparing the loss functions of MLPs with and without LayerNorm. The prominent lines are smoothed versions of the faded lines.

4.5 Stage 2 translation

From the second pipeline stage, I constructed a simplified version of the GNN, which was translated. Two features in the original GNN could not be translated with HLS4ML. The first is the aggregation function, i.e. the summation of messages at the nodes. The second is an indexing operation of the type $a = b[c]$, where a, b and c are tensors. In the model, this operation is used to connect node indices to edge indices, which is used when producing inputs for the node and edge networks. I therefore also omitted indexing operations from the GNN. As a result, the simplified GNN consists of the individual MLPs from the node and edge encoders, node and edge networks and the output classifier. Fig. 19 illustrates the updated GNN architecture.

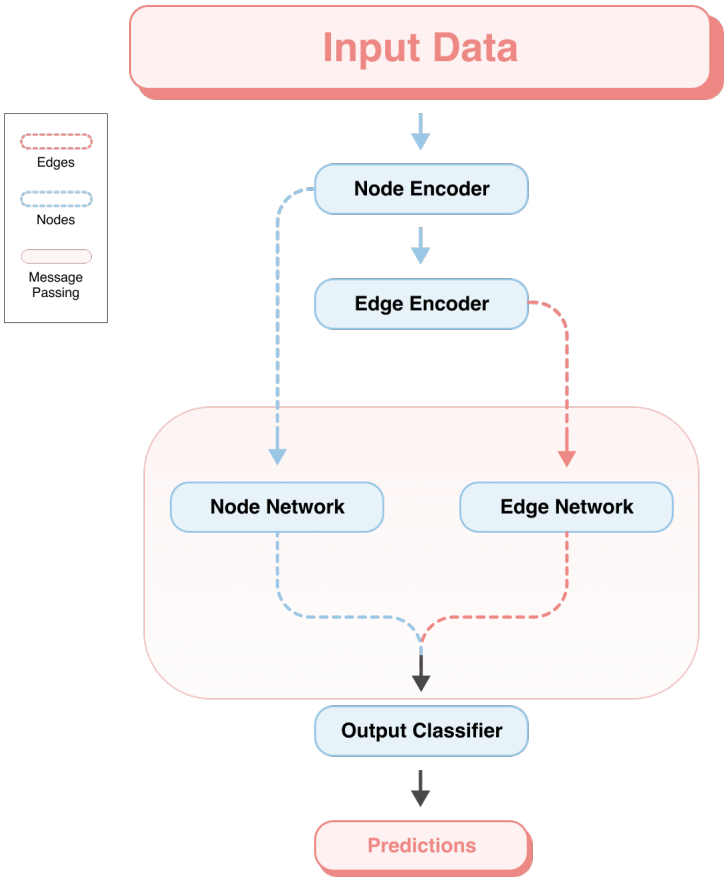


Figure 19: Simplified GNN. The node and edge networks still remain in the message passing block, however removal of the aggregation function removes the functionality of message passing.

Omitting the aggregation function and indexing operations will certainly impact the amount of bit operations performed by the model, and hence its occupancy on an FPGA. I therefore expect the full GNN to give higher resource estimates, were we able to implement it on an FPGA. The amount of extra resources it would occupy has not been evaluated, however could be estimated with a back-of-the-envelope calculation.

To simulate a GNN with multiple message passing steps, I iterated over the node and edge networks, as is demonstrated in the "message passing block" in Fig. 19. When using HLS4ML to translate a GNN with multiple message passing steps, the FPGA implementation will by default parallelise the message passing steps. This leads to decreased latency, but also increased resource usage. It is, however, possible via HLS4ML to force some operations to be executed in series. This is done by increasing the reuse factor setting, which determines how many times a component in the FPGA is reused. It could thus be possible to use an increase in the reuse factor to decrease the GNN's resource usage.

By default, HLS4ML sets the reuse factor to 1. I attempted to increase it to 2 when trans-

lating a GNN with 2 message passing steps, however this was not supported by HLS4ML when using Quartus backend. Most of the models that I translated to HLS were kept to 1 message passing step with a few exceptions.

4.6 Stage 3 translation

To gauge the resources used by the track reconstruction stage, the Walkthrough method was implemented in VHDL. This method was chosen as its implementation is suited for FPGAs (e.g. fewer for-loops and operations).

The input data to this algorithm consists of three arrays. One array ("edge array") contains the information about edges. Here, the index for the array element (i.e. array element number $0, 1, 2, 3$ and so forth) identifies the first node in the node pair that an edge connects. The content of that array element identifies the second node. If the first node shares edges with multiple other nodes, the array element will be wider. For example, if node number 0 is connected via edges to node number 4 and 6, the 0th array element will be $[4, 6]$. Another array ("score array") contains the score associated with each edge, as given by the GNN. The third array ("seed array") contains the hits recorded in the innermost detector layer. For this implementation, we assume that tracks are always seeded in the first detector layer, and therefore the hits in this layer correspond to the seeds of the track. The nodes in the seed array are the starting points of the Walkthrough algorithm.

For a preliminary implementation of the Walkthrough algorithm, only the first edge in the edge array was considered. I.e. if a certain node shares edges with multiple other nodes, such that the array element is e.g. $[4, 6]$, only the edge with node 4 will be passed along the algorithm. This was done under the assumption that edges with the same starting point can be sorted by decreasing score, such that only the edge with the highest score remains.

The VHDL algorithm consists of two parts: a score cutter and the Walkthrough itself. Fig. 20 demonstrates the flow of the algorithm. The score cutter works through the edge scores one by one. For edges with scores below 0.5, it sets the corresponding array element in the edge array to 0. For example, if the edge connecting node 2 and node 8 has a low score, element 1 of the edge array will be set to 0.

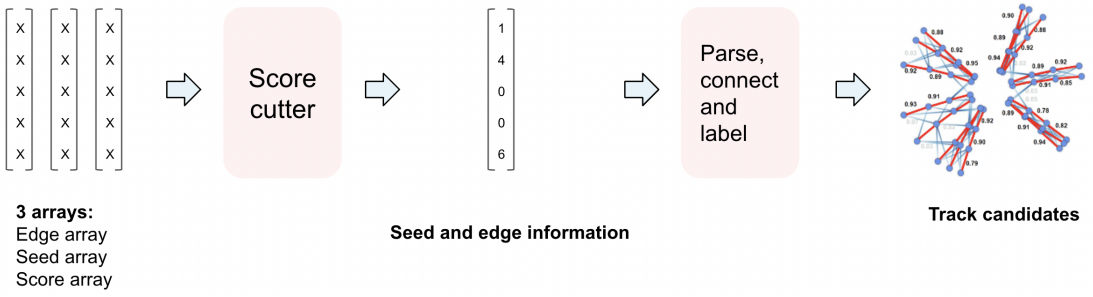


Figure 20: The VHDL implementation of the Walkthrough method includes a score cutter, and the Walkthrough algorithm itself.

Once scores have been cut, the edge array is stored in RAM. Here, the index of the array element corresponds to the address for RAM on the FPGA, where the content of the array element is the content stored at a given RAM address. Now, the Walkthrough algorithm is applied to one seed at a time.

Given the first seed, e.g. 0, the algorithms looks for the address zero in the RAM. The content of the address complete the first edge: let's assume this is the value 2. The algorithm looks then for address 2 and its content and so forth until a zero is found as content. When zero is found, the track is complete and a new seed is taken into consideration for a new track.

Since the edge information is stored in RAM, the RAM assigned to the algorithm on the FPGA must be at least large enough to contain the edge array. If the edge array is larger than the available RAM space, the additional data will be lost.

5 Training and optimisation methods

5.1 Model training

In order to make accurate predictions, machine learning models are trained on a set of training data. The model is presented with some data, it adjusts its weights based on the patterns it sees, and the accuracy of the predictions it makes is calculated. This process is repeated over and over again until predictions stabilise.

Training parameters are set to determine how training is performed. I will here go through some of the key hyperparameters and considerations made when I trained the metric learning MLP and the GNN.

Loss function

A loss function measures the difference between the model's predictions and the truth values. The goal is to minimise the loss function, meaning more accurate predictions. For the MLP in stage 1 of the pipeline, I use a "hinge embedding loss" [44]. This is the case for both the ITk pipeline and the TrackML pipeline. The total loss function, L, is given by:

$$L = \text{mean}(w_i \cdot l_{i,true}) + \text{mean}(w_i \cdot l_{i,false}) \tag{6}$$

where,

$$l_{i,true} = d_i \tag{7}$$

$$l_{i,false} = \max\{0, \Delta - d_i\} \tag{8}$$

and the weights w_i are given by:

$$w_i = \begin{cases} 3 & \text{if signal true edge,} \\ 0 & \text{if non-signal true edge,} \\ 1 & \text{if false edge} \end{cases} \tag{9}$$

Here, d_i is the distance in latent space between the ith node pair, and Δ is the margin, which is a number that determines the scale of the latent space. $true$ and $false$ denote whether the edge between the ith node pair is a true or a false edge. As such, the loss function will be minimised by placing signal hits from the same track close to each other in latent space, and by placing hits from different tracks far away from each other in latent space.

The GNN loss function is a binary cross-entropy loss [45] for both the TrackML and ITk pipelines. Here, the loss function, L, is given by:

$$L = -w_i[y_i \cdot \log x_i + (1 - y_i) \cdot \log(1 - x_i)] \tag{10}$$

Here, w_i are the weights added to the loss function, and are the same as Eq. 9. x_i is the properties of the ith node pair, and y_i is the truth condition of the edge connecting the node pair ($= 1$ when true, $= -1$ when false).

Optimiser

The optimiser is the algorithm that updates the model's weights and biases during training to minimise the loss. For both the MLP and GNN, I used the AdamW optimiser [46]. It uses stochastic gradient descent to minimise the loss, and is able to adapt the learning rate of individual weights in the model during training.

Learning rate

The learning rate determines the step size by which the weights of the model are updated during training. At the start of training, we want the learning rate to be quite high, such that weights are tuned in rather quickly. As training progresses, we want the learning rate to become lower, such that it is able to fine-tune the weights. The learning rate is thus variable for both the MLP and the GNN, starting at 0.01, and decreasing by 30% every 10 epochs.

Epochs

Epochs refer to the amount of times the model sees the data. To best learn from the data, ML models are often trained for multiple epochs. Ideally, a model is trained until the loss function, and hence also the accuracy, stabilises. This is affected by multiple factors such as the learning rate and the size of the dataset. For the models I use in this project, I found the loss to stabilise after 50 to 150 epochs. For the results presented in Section 7, the models have been trained for 150 epochs unless stated otherwise.

Data split

The data was split into a training set consisting of 80 events, a validation set consisting of 10 events, and a testing set consisting of 10 events. The training set was used for training

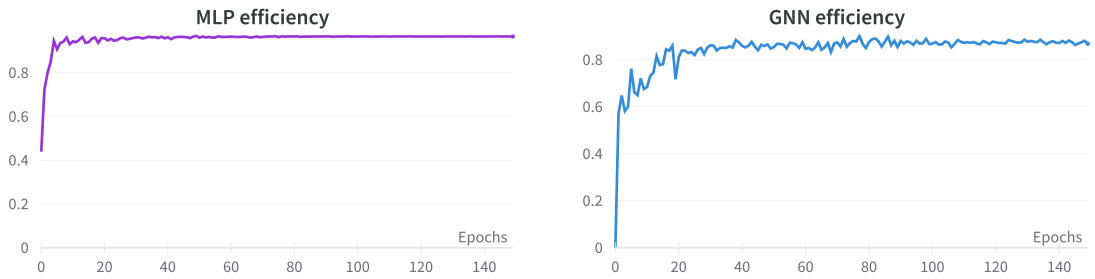
the model, and the validation set for providing feedback to the model during training. The testing set was used to measure the performance of trained models.

5.2 Pruning

Pruning is a technique used to reduce the size of a model while maintaining, to a certain extent, performance. It works by setting the smallest weights to zero, which reduces the amount of matrix multiplication operations it performs. Pruned models could hence potentially consume fewer resources when implemented on FPGAs. Pruning is implemented iteratively during training, where only a small amount of weights are removed at a time. This ensures that the remaining weights can be re-trained to compensate for the removed weights.

To prune the MLP and GNN, I implemented the PyTorch function `ModelPruning` into the CommonFramework. By studying the training behaviour of the two models, it was possible to determine when pruning should start and stop, and by what amount models could be pruned before performance suffers.

In Fig. 21, I train an MLP and a GNN without pruning. We see here that the efficiency of model predictions plateaus after training for a certain amount of epochs. At this point, the model has adjusted its weights sufficiently to know which weights are the smallest. This is where pruning can start. For the MLP, this plateau happens after around 35 epochs. For the GNN, pruning could start slightly later, at around 65 epochs. From the starting point of pruning, a small amount of weights (the "pruning amount") were then removed every 5 epochs. I set the pruning to stop 20 epochs before training stopped altogether, giving the models time to adjust its remaining weights.



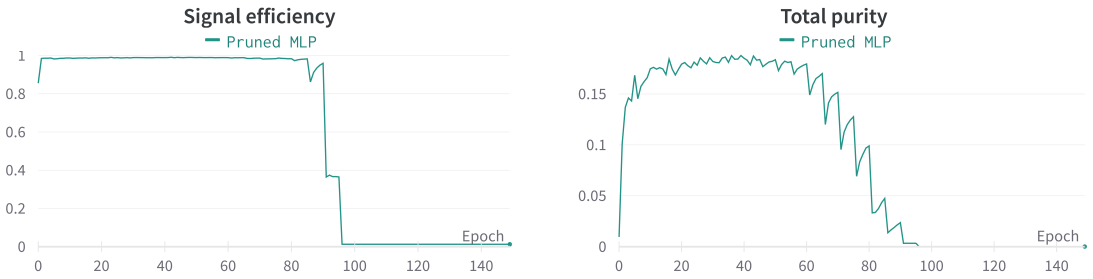
(a) MLP signal efficiency

(b) GNN signal efficiency

Figure 21: Both MLP and GNN efficiency stabilise after training for a certain amount of time. The MLP stabilises sooner than the GNN, allowing pruning to start sooner.

I determined the pruning amount by observing the training performance of some trial models that underwent different amounts of pruning. At every pruning iteration, the efficiency and purity of the model dropped slightly. Since pruning was implemented during training the efficiency and purity could, in most cases, recover before the next pruning it-

eration. However, since more and more weights are removed, the drops become larger and larger. If the pruning amount is too large, this could result in the behaviour seen in Fig. 22. The pruning amount was thus determined on a model-to-model basis, where the training behaviour was observed to avoid any large drops in efficiency and purity.



(a) Efficiency **(b)** Purity

Figure 22: These plots are from an initial study on pruning, where pruning was set to start in the 35th epoch. For a while, the efficiency and purity were maintained, however the efficiency started to suddenly drop at 95 epochs, and the purity already suffered at 60 epochs.

In Section 7.3.3, I present a deeper study on the effect pruning has on the efficiency and purity of trained models. Here, I also present its effect on FPGA resource consumption. Some of the pruned models that were used to study FPGA implementation were pruned after training. Since these models were used purely for an initial exploration of FPGA resource usage, it was not necessary for them to maintain performance by pruning during training.

5.3 Training cuts

During the development of the models presented in this report, I applied some cuts to the data. Since events in the datasets are very large, training the models on whole events can both take a long time, and is limited by the size of the GPU on which training takes place. Applying cuts is one way of reducing the data, such that different model features can quickly be tested. In general, I applied cuts such that the models train on signal particles, or particles that meet at least one of the signal requirements, as presented in Section 3.4.

5.3.1 Heuristics cuts

When implementing the Heuristic method for the TrackML pipeline, these were the cuts that I made to the data:

- **Noise removal:** Hits that do not belong to a reconstructible particle were removed.
- p_T **cut:** Any hits belonging to particles with $p_T < 1$ GeV were removed.

- **Double hits:** In the TrackML dataset, some particles left two hits in the same layer. This is due to modules in the detector layers overlapping, and they thus measure the same hit twice. Therefore, I removed any duplicates.
- **Track length:** Hits that belong to tracks with fewer than 4 hits were removed.
- **Detector sectioning:** I section the detector into 8 sections (4 ϕ sections, 2 η sections), such that graphs become smaller.
- **Volume cut:** For this project, I considered only hits in the "barrel" region of the generic detector. This is the region comprised of volume 8, 13 and 17 in the detector pictured in Fig. 4. Thus, any hits belonging to the volumes outside the barrel were cut.

5.3.2 Metric Learning cuts

When training the Metric Learning algorithm, I implemented a 1 GeV cut on the p_T of the data. This was the case for both TrackML and ATLAS ITk data. This effectively also removed noise from the data, since the noise hits do not have a p_T value attached to them. For the ITk pipeline, tracks of all lengths were included, and I did not section the detector. I also conducted a preliminary test for the ITk pipeline with the p_T cut removed. Here, I significantly reduced the MLP and GNN sizes to be able to train the models on the available GPU. Results on performance and resource consumption of this pipeline will be presented in Section 7.5.

6 TrackML Results

6.1 Pipeline performance (on GPU)

At first, I ran the TrackML track reconstruction pipeline with Heuristics for graph construction and Walkthrough for track reconstruction. With these methods, and a GNN with 5 message passing steps, the total track reconstruction efficiency was 94.8%. The fake rate was 0.0%.

In an attempt to improve this result, I substituted the first and final stages of the pipeline, such that I used Metric Learning for graph construction and Connected Components for track reconstruction. The MLP used in Metric Learning had 16 hidden dimensions in its linear layers, corresponding to 1084 parameters. The GNN also had 16 hidden dimensions, along with 5 message passing steps. This corresponds to 2,369 parameters. The track reconstruction efficiency for this pipeline was 97.8%, and the fake rate was 0.8%. Fig. 23 shows the p_T-wise efficiency.

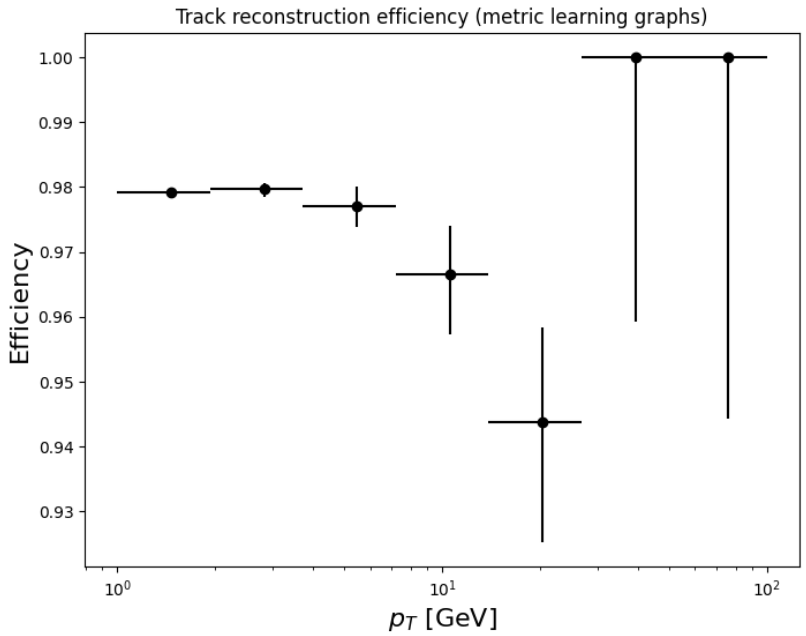


Figure 23: Track reconstruction efficiency vs. pT for the TrackML pipeline.

6.2 Resource estimates

From the TrackML pipeline, I converted the Metric Learning MLP to HLS and compiled it in Quartus. The resource estimates are found in Table 3. From these numbers, we see that DSP blocks are the predominantly used resource. This is expected, since the MLP performs many matrix multiplications, and such multiplications are typically processed by DSP blocks on FPGAs. DSP blocks are thus a limiting factor for the implementation of machine learning components on FPGAs. In Section 7.2 I elaborate on how pruning the model can decrease the DSP usage.

Resource	DSP blocks	ALMs	RAM blocks
Usage on S10 FPGA	19.9%	3.3%	9.9%

Table 3: Resource estimates for an MLP with 16 hidden dimensions (1,084 parameters)

With these estimates, we have an idea of the amount of resources we can expect an MLP of a given size to take up on the target device. This allowed me to make informed decisions about which model sizes to use when training the ITk pipeline.

7 ITk Results

In this section, I present my findings for the ITk pipeline. At first (Section 7.1), I explore the FPGA implementation of individual parts of the pipeline and their respective resource usage. I then look into the effect on resource usage from pruning the models (Section 7.2). After the initial studies on resource usage, I will present my findings on the performance on a GPU of individual pipeline stages, i.e. their respective efficiency and purity (Section 7.3). The influence of pruning on performance will also be discussed (Section 7.3.3). With an understanding of the behaviour of the individual pipeline stages, I then use these results to construct an entire pipeline, and present its performance and resource usage (Section 7.4). Finally, I train a pipeline without the p_T cut, which was discussed in Section 5.3.

7.1 Resource usage

7.1.1 Metric Learning MLP

To study the resources used by the Metric Learning MLP and the simplified GNN, I varied the size of the models. Specifically, I increased the number of dimensions in the hidden layers until the models could no longer be compiled with Intel Quartus. This had a direct influence on the number of parameters in the model, and hence the model size. The number of hidden dimensions started at 2, and was increased to 4, 8, 16 and so forth. I recorded the FPGA resources: DSP blocks, ALMs and RAM blocks (in some figures referred to as just "RAM").

The resource usage of the MLP is plotted in Fig. 24. It was possible to compile the model with Quartus up to 16 hidden dimensions, corresponding to 1,084 parameters in the model. The larger the MLP, the more resources it occupies on the FPGA. It occupies primarily DSP blocks, followed by RAM and ALMs. As was also concluded in Section 6, DSP blocks seem to be the limiting factor in terms of fitting machine learning models onto an FPGA.

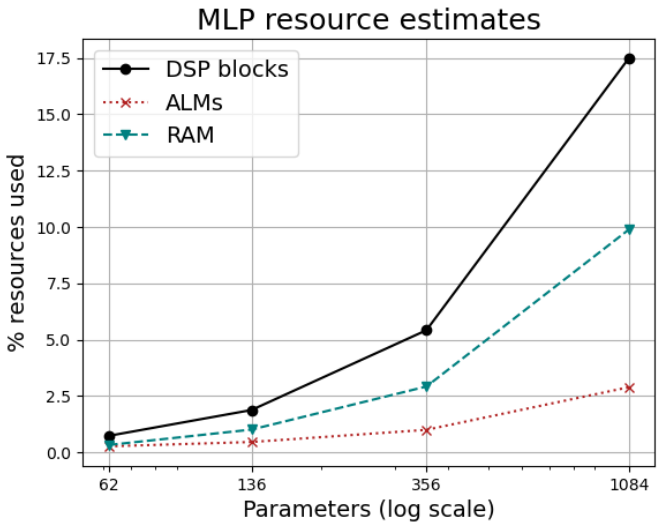


Figure 24: FPGA resource usage for increasing model sizes.

7.1.2 GNN

Taking a look at the GNN, there is a similar trend in resource usage, as plotted in Fig. 25. A GNN (simplified version) with one message passing step, could be compiled with Quartus for up to 16 hidden dimensions, corresponding to 2,369 parameters. Here, DSP blocks are also primarily used, followed by ALMs and RAM.

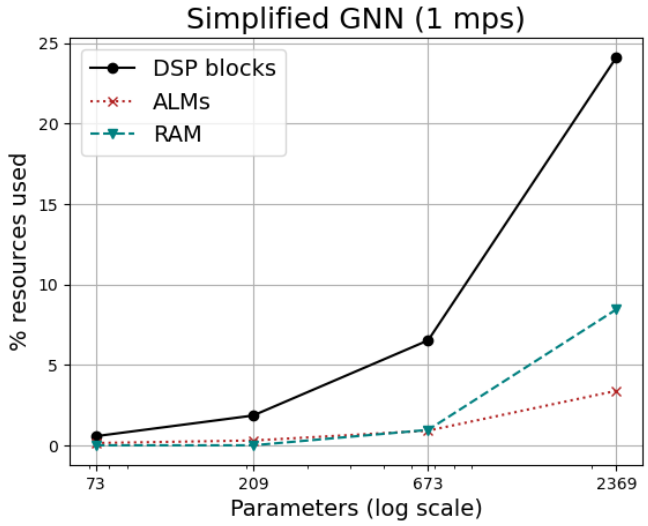


Figure 25: FPGA resource usage for simplified GNN with 1 message-passing step.

Adding multiple message passing steps to the model increases the amount of resources used, as seen in Fig. 26. Here, I translated and compiled a GNN with 4 hidden dimensions and message passing steps ranging from 1 to 10. There is a linear increase in both DSP blocks usage and ALM usage when adding message passing steps, whereas the RAM remains flat. As discussed in Section 4.5, the increase in resource usage might be avoided by increasing the reuse factor.

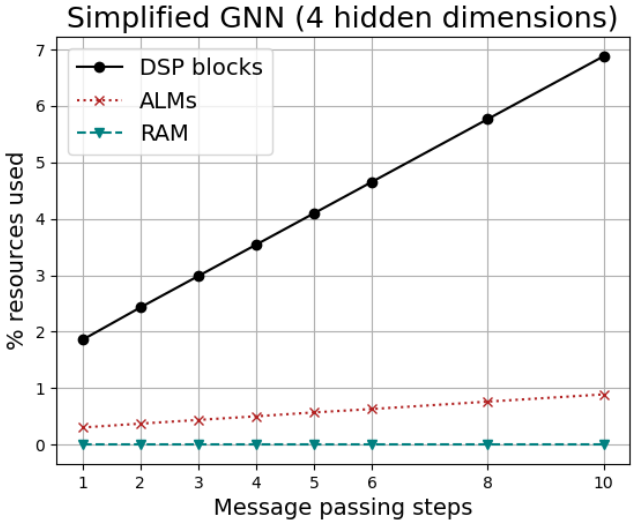


Figure 26: Adding more message passing steps increases the FPGA resource usage. This model has 4 hidden dimensions, corresponding to 209 parameters.

7.1.3 Walkthrough algorithm

The walkthrough algorithm described in Section 4.6 was simulated using dummy data. In the compilation, I used an edge array with a length of 18,000. This roughly corresponds to the average amount of nodes in the graphs coming from the edge classification stage. The seed array was set to a length of 8,000, which is a typical amount of hits in the innermost layer. The resource estimates obtained from the simulation are listed in Table 4.

Resource	DSP blocks	ALUTs	RAM
Usage	0%	69.7%	0.3%

Table 4: Resources used by the VHDL implementation of the Walkthrough algorithm.

From the synthesis in Quartus, ALM estimates were not available. Instead, I report here the usage of Adaptive Look-Up Tables (ALUTs). ALUTs are components placed in ALMs, and there is one ALUT used for each ALM in Intel FPGAs [47]. The ALUT estimate

is therefore a good first estimate of how many ALMs will be used by the design. From these resource estimates, we see that the implemented Walkthrough algorithm uses mostly logic modules - 69.7% of the device's ALUTs are occupied by the algorithm. Since the algorithm does not perform any calculations, no DSP blocks are used. A low amount of RAM blocks are used for storage of the edge arrays. This comes out to 0.3% of the RAM blocks available in the device. The Walkthrough algorithm thus behaves quite different to the machine learning components of the pipeline, where DSP blocks and RAM were used more predominantly.

7.1.4 Resource limits

In the studies above, the number of hidden dimensions was doubled at every increase in model size. For the MLP, for example, the 32-dimensional model could not be compiled with Quartus, whereas the 16-dimensional model only occupied 17.5% of DSP blocks. To narrow in on the concrete limit where Quartus fails to compile the models, I compiled models where the number of hidden dimensions was incremented by 2 in the range 16 to 32 dimensions. Here, I found that Quartus resource estimates could be obtained for up to 28 hidden dimensions. This model had 2,896 parameters, and it occupied 96% of DSP blocks and 23% of ALMs. This narrows down the concrete limit on model size to just below 3,000 parameters if we allow the model to occupy the full FPGA. In reality, when designing FPGAs, there are stricter limits on the amount of resources that can be used: to be able to close a design from a timing perspective (using a certain amount of resources at a certain clock speed) an FPGA's resources are never fully used.

7.1.5 HLS vs Quartus estimates

As described in Section 4.3, resource estimates can also be obtained from HLS synthesis. These estimates are considerably different to the estimates produced by Quartus. In Fig. 27, I plot the estimated RAM block, DSP block and ALUT usage obtained from Quartus and HLS, respectively. HLS did not produce a resource estimate for ALMs, however it was possible to compare the ALUT estimates. The Quartus estimate for DSP blocks is always double that of HLS. The Quartus RAM estimate is between 80 and 90% that of HLS. For ALUTs, the discrepancy varies a lot. For the 62-parameter model, the Quartus estimate is 85% of the HLS estimate. For the 1,084-parameter model, the Quartus estimate is 50% of the HLS estimate.

Some models that do not compile in Quartus do pass HLS compilation. For these models, the HLS estimate can give a rough indication of what the Quartus estimate would look like. For example, if HLS predicts a model to use 3,000 DSP blocks, we may infer that the Quartus estimate is 6,000, which is more than 100% of the resources available, and could lead to a failed Quartus compilation.

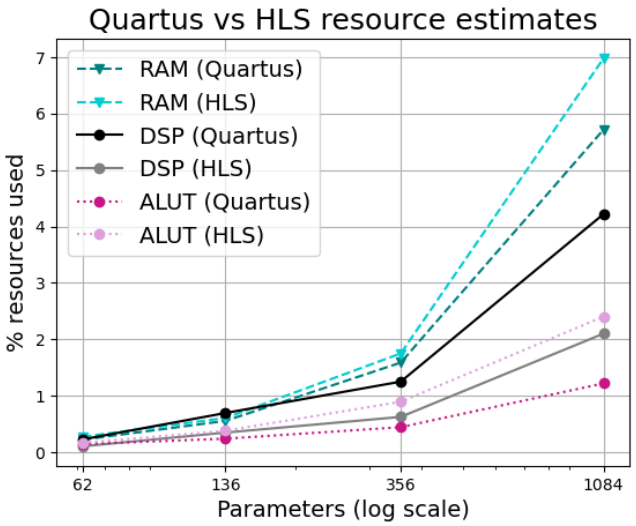


Figure 27: Comparing Quartus and HLS resource estimates (DSPs, ALUTs and RAM blocks). Resource estimates were obtained from pruned MLPs with increasing sizes.

7.2 Pruning studies

Pruning was implemented with the methods presented in Section 5.2. I present here my findings on the effect of pruning on FPGA resource usage. In Section 7.3.3, I will go through the effect pruning has on model performance.

7.2.1 MLP resource usage

Fig. 28 demonstrates the effect of pruning on FPGA resources. Here, I pruned an MLP with 8 hidden dimensions (356 parameters), and recorded the resource usage for various amounts of pruning. The pruning was performed during training. We see here that increasing the amount of pruning leads to a slightly concaving decrease in resource usage.

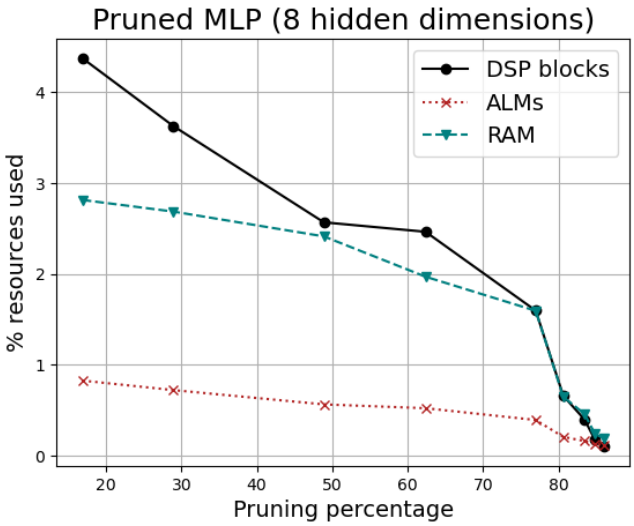
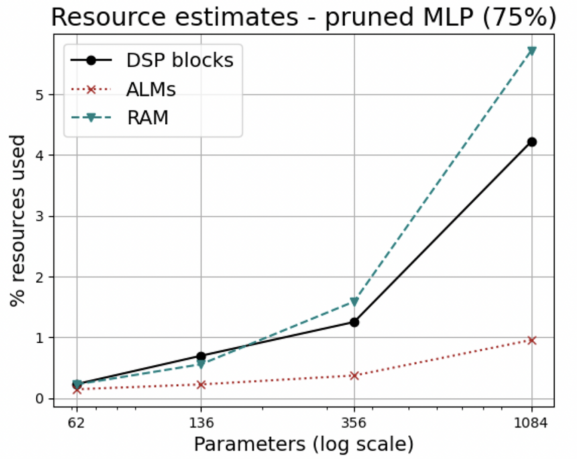
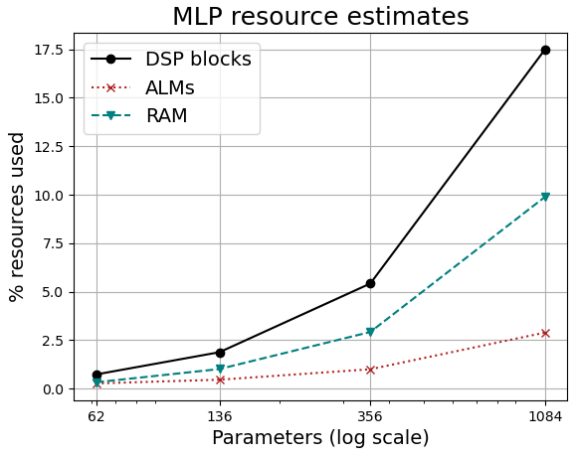


Figure 28: Resource estimates for an increasingly pruned MLP with 8 dimensional hidden layers

Varying the model size while keeping the pruning amount constant allows us to compare pruned models to the unpruned models presented in Fig. 24. Fig. 29a contains a plot of the resources used by MLP models that have been pruned by 75%. In this case, the models were pruned after training. Comparing this plot to Fig. 24, we see that DSP block usage has been reduced by 75%, whereas RAM decreases only by 32 - 46%. We see also that RAM becomes the predominantly used resource as models grow larger.



(a) Resource estimates for an MLPs of various sizes, all pruned by 75%. The x-axis contains the total number of parameters for comparison with Fig. 24. The number of non-zero parameters will be 25% of the total number.



(b) Resources estimates for MLPs of various sizes, unpruned. This figure is identical to Fig. 24

Figure 29: A comparison of the resource usage of pruned and unpruned models

7.2.2 GNN pruning

To study the effect of GNN pruning on FPGA implementation, I pruned the simplified GNN after it was initialised. The full GNN was also pruned during training to study the effect on performance, which will be described further in Section 7.3.3. In Fig. 30, I plot the FPGA resource usage of a GNN, which has been pruned between 20 and 90%. Similar to

the MLP, more pruning results in fewer resources used across both DSP blocks, ALMs and RAM. The decrease is almost linear, but slightly concaves for higher pruning percentages.

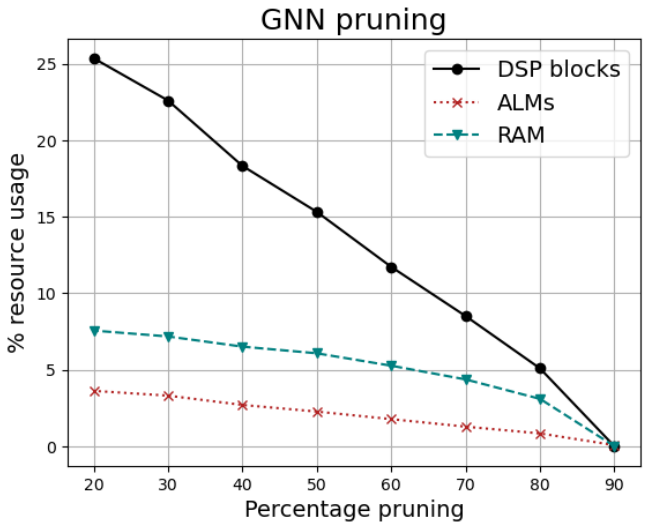


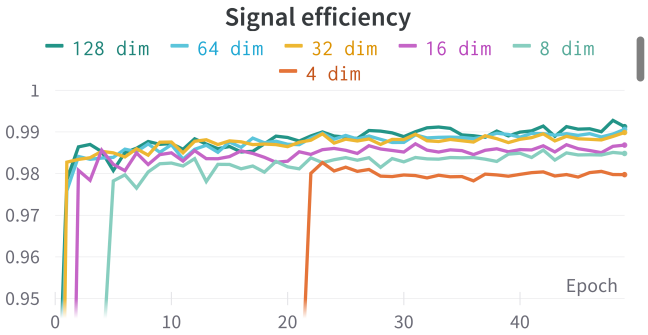
Figure 30: A GNN with 16 hidden dimensions and 2 message passing steps pruned by increasing amounts.

7.3 Performance studies (on GPU)

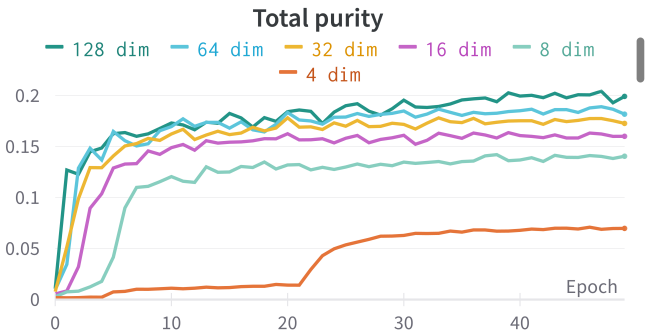
As well as keeping the models small for FPGA implementation, it is also important that models perform well. Given the constraints on model size presented in the previous section, I will now explore the effect that size and pruning has on efficiency and purity. I will present my findings for individual stages of the pipeline, and in Section 7.4, I will use these findings along with results on resource usage to explore how well the full pipeline performs.

7.3.1 MLP efficiency

To study how the MLP's size influences its efficiency, I trained models with a varying number of hidden dimensions for 50 epochs. The signal efficiency and the total purity were recorded. Fig. 31a shows a plot of the signal efficiency during training of these models. Although the improvements in efficiency are small, there is a correlation between model size and its efficiency. Only models with 32 hidden dimensions or more performed above 99%. For the purity, there are larger differences in performance between the models. This is shown in Fig. 31b where it becomes clear that model size affects its purity upon training. The increase in performance for larger models can be attributed to the amount of information stored in the model. For each parameter contained in the model, one more number, and hence one more piece of information, is used to make the predictions. Therefore, the more parameters in a model, the better its predictions will be.



(a) MLP signal efficiency



(b) MLP total purity

Figure 31: Signal efficiency and Purity of MLPs after training for 50 epochs.

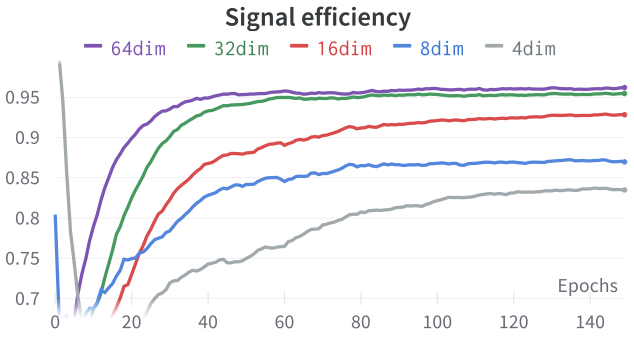
Since these plots clearly demonstrate larger models perform best, we are interested in using as large a model as possible for the final pipeline. In Section 7.1 I found that the upper limit for unpruned models was 16 dimensions. I however also found that pruning leads to a decrease in resource usage. In section 7.3.3, I will explore the possibility of fitting larger models onto the target device by applying pruning.

7.3.2 GNN efficiency

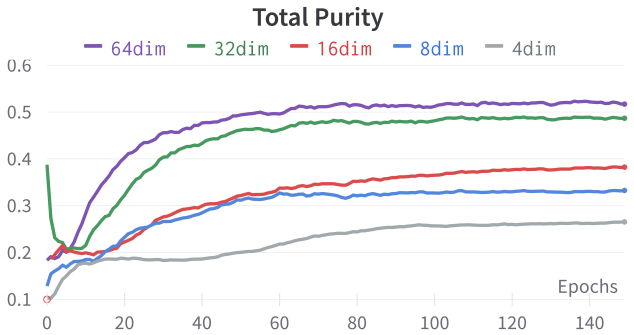
For the GNN, I also recorded the efficiency and purity as a consequence of varying the model size and number of message passing steps. Here, the models were trained for 150 epochs.

Fig. 32 shows the efficiency and purity of the GNN for different numbers of hidden dimensions. In this case, there is also a clear increase in performance when the model size is increased. For example, the 8-dimensional model has a final signal efficiency of 86.4%, the 16-dimensional model 92.7% and the 32-dimensional model 95.5%. Based on the plots, and considering the size constraints found in Section 7.1.2, I consider the 16-dimensional

model to be a good compromise between model size and performance.



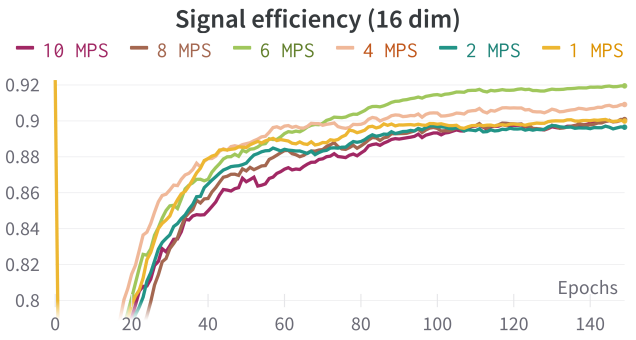
(a) GNN signal efficiency (1 message passing step)



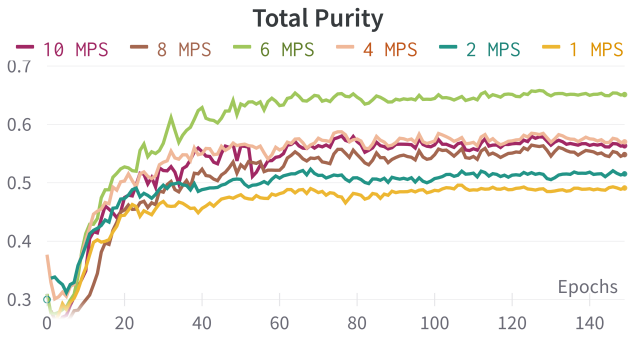
(b) GNN total purity (1 message passing step)

Figure 32: Signal efficiency and Purity of GNNs after training for 150 epochs. The lines have been smoothed for better visibility.

When varying the number of message passing steps, there is not a clear relationship between them and performance results. Fig. 33 shows the efficiency and purity when training the models. The model with 6 message passing steps performs the best when looking at both efficiency and purity. For the purposes of my project, however, I only used 1 message passing step for the models I translated. This is due to the fact that the reuse factor could not be increased, as also mentioned in Section 7.1.2.



(a) GNN signal efficiency



(b) GNN total purity

Figure 33: Training GNNs with an increasing number of message passing steps for 50 epochs. The lines have been smoothed for better visibility.

When adding a message passing step to the GNN algorithm, the node properties are updated once more, and each node will contain information from one further degree away. This also updates the parameters, and intuitively, would make the model more accurate. This is not the behaviour seen in the plots above. If too many message passing steps are added, the nodes end up containing unnecessary information from other nodes very far away, and the model will learn features that are not relevant to the given node. This explains why the models with 8 and 10 message passing steps perform worse than those with 4 and 6.

7.3.3 Pruning: MLP efficiency

As discussed in Section 5.2, pruning leads to a small decrease in efficiency and purity. The efficiency can often be restored through training. However if enough weights are removed, the performance will suffer indefinitely. I therefore tested the maintenance of efficiency and purity for different models. The models I tested were trained for 150 epochs.

As a first step, I looked at the models that were used to investigate the effect of pruning on resource usage (as presented in Fig. 28). In Fig. 34, I plot the efficiency of the MLP with 8 hidden dimensions for increasing pruning amounts. The efficiency was recorded at the end of training. We see here that efficiency is maintained until just before 50% pruning. Purity, on the other hand, drops already at 30% pruning.

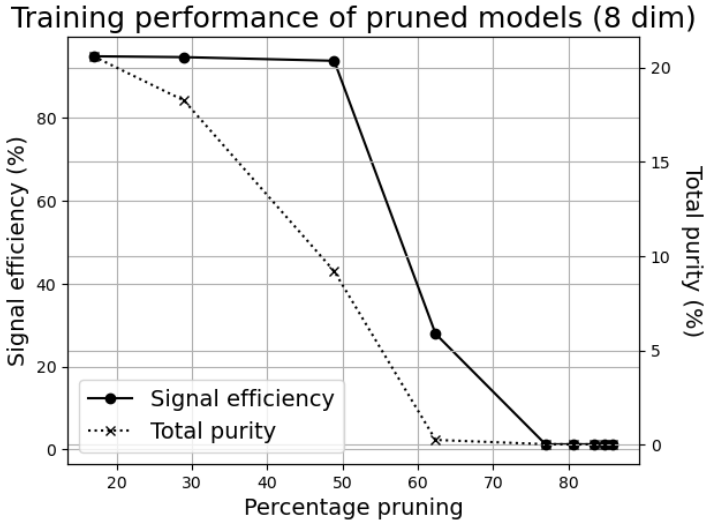


Figure 34: Pruning an 8 dimensional MLP by more than 50% leads to a significant drop in efficiency.

To see how pruning affects a larger model, I did the same test for an MLP with 32 hidden dimensions. The outcomes are plotted in Fig. 35. Here, efficiency suffered only after 78% pruning. Purity decreased slightly after 32% pruning, and more drastically after 78% pruning. From this we can conclude that the larger the model, the higher the percentage can be pruned whilst maintaining performance.

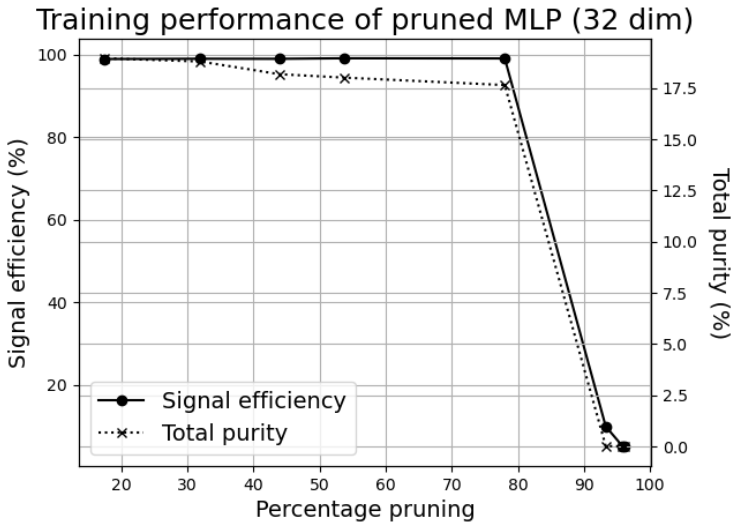


Figure 35: Pruning an MLP with 32 hidden dimensions (3,692 parameters) by up to 98%. Purity starts to slightly drop after 32% pruning, and efficiency after 78%. At 78% pruning, the efficiency is 99.0% and the purity is 17.7%.

In Section 7.1.4, I found that unpruned MLPs could only be compiled in Quartus up to 28 hidden dimensions. Since pruning reduces resource usage, I tested whether a pruned 32 dimensional MLP could be compiled. The model I compiled was pruned by 78%, i.e. before the performance drop we see in Fig. 35. The resource estimates obtained from this compilation are listed in Table 5.

Resource	DSP blocks	ALMs	RAM
Usage	11.7%	2.9%	15.3%

Table 5: Resource usage of an MLP with 32 dimensional hidden layers, pruned by 78%. The model had 99% efficiency and 17.7% purity.

I conducted a similar test for a 64 dimensional MLP. Here, efficiency remained at 99% until 95% pruning, and purity at 18% until 80% pruning. It was however not possible to compile the well-performing pruned models in Quartus. A model pruned by 90% passed the HLS compilation, and inferring from the resource estimates it produced, the Quartus estimate for RAM would have been above 100%, leading to the compilation failure.

Comparing the resource estimates presented in Table 5 with the plot from Fig. 24, we see that the pruned 32-dimensional model has similar resource usage to the unpruned 16-dimensional model. The performance of the 16-dimensional model is presented in Fig. 31a, where it has a final efficiency of 98.7% and a purity of 16.0%. The resource usage and performance of the two models is compared in Table 6.

	16 dim, unpruned	32 dim, pruned
DSP blocks	17.5%	11.7%
ALMs	2.9%	2.9%
RAM blocks	9.9%	15.3%
Efficiency	98.7%	99.0%
Purity	16.0%	17.7%

Table 6: Comparing FPGA resource usage and performance of two models: one 16-dimensional unpruned, and one 32-dimensional pruned.

For roughly the same resource usage (fewer DSP blocks, but more RAM blocks), we have thus gained an increase in performance by using a slightly larger, but pruned model. Based on this, I conclude that a 32-dimensional MLP, pruned by up to 78% is a good candidate for the Metric Learning algorithm implemented on a Stratix 10 GX FPGA.

7.3.4 Pruning: GNN efficiency

To test how the performance of a GNN is affected by pruning, I trained GNNs with 16 hidden dimensions in the linear layers (2,369 parameters) with various amounts of pruning. We saw in Section 7.1.2 that GNNs with up to 16 dimensions could be implemented onto an FPGA, and in Section 7.3.2, I showed that this model performs better than any of the smaller GNNs I trained. The pruned models were trained for 150 epochs and had 1 message passing step. The result can be seen in Fig. 36. We see here that purity starts to drop after around 42% pruning, and efficiency after around 60%.

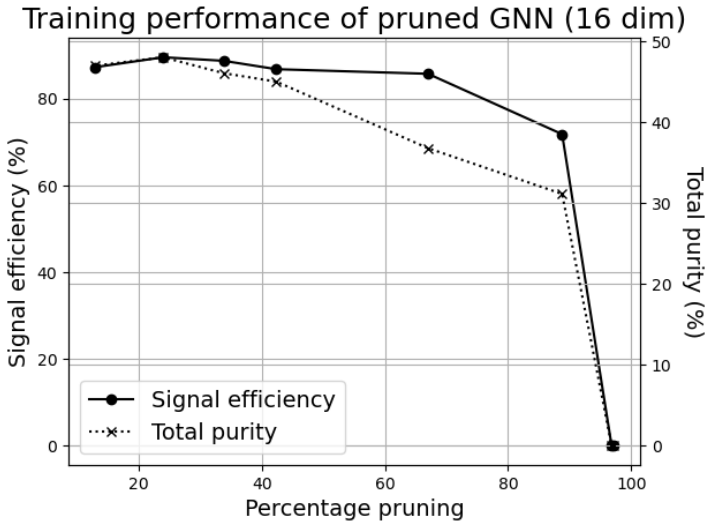


Figure 36: When pruning a GNN (16 dim), the performance is roughly maintained until 42% pruning.

As in the case of the MLP, I checked the resource usage of the 16-dimensional GNN pruned by 42%, i.e. before the performance drops significantly. These estimates are presented in Table 7. In Section 7.3.2 I concluded that a 16-dimensional GNN performs well for its size with 92.7% efficiency and 38.6% purity. To fit the model on the target device, it is not strictly necessary to prune it, but any effort to decrease the model size is desired considering the aim to fit a full pipeline onto an FPGA. Based on these findings, along with the results presented in this section, I conclude that a 16-dimensional GNN with 1 message passing step, pruned by up to 42%, is a good candidate for implementation onto the Stratix 10 GX FPGA.

Resource	DSP blocks	ALMs	RAM
Usage	13.4%	2.0%	6.8%

Table 7: Resource usage of a GNN with 16 dimensional hidden layers, pruned by 42%.

7.4 A sample pipeline

Having obtained results for how various model architectures impact both performance on a GPU and FPGA resource usage, I used these results to construct a full pipeline. The goal here was to maximise the track reconstruction efficiency and purity, while minimising the resources occupied on the target device. While the amount of resources available on the Stratix 10 FPGA puts an upper limit on how much the pipeline can occupy, the limit for the parts of the pipeline I am exploring is, in reality, much smaller. Two factors should be kept in mind here. First of all, I only test the occupancy of the MLP and a simplified GNN. The pipeline consists of many more algorithms including the FRNN search and the track reconstruction algorithm. The full GNN also has more operations than the simplified version. Secondly, as described in Section 7.1.4, the FPGA will not be able to operate in reality if all resources are occupied. Based on these factors, as an estimate, I decided to aim for the MLP and GNN combined to occupy at most 40% of each respective resource.

The MLP chosen for this pipeline had 32 dimensional hidden layers, and was pruned by 74%. I chose these settings based on the results reported in Section. 7.3.3. While efficiency could be maintained until 78% pruning, I decided to leave some room for error and prune it by 74%. At this amount, efficiency is still maintained, whereas purity has only slightly dropped. The GNN I chose for the pipeline had 16 hidden dimensions and was pruned by 42%. As reported in Fig. 36, pruning a GNN of this size by 42% causes only a slight reduction in efficiency and purity.

The performance of both individual steps of the pipeline, as well as the full pipeline are listed in Table 8. The p_T-wise track reconstruction efficiency for signal particles is plotted in Fig. 37. The resources used by the MLP and GNN are listed in Table 9.

Track reco: Efficiency	97.9%
Track reco: Fake rate	15.5%
MLP signal efficiency	98.9%
MLP total purity	16.5%
GNN signal efficiency	89.3%
GNN total purity	47.9%

Table 8: Performance on GPU of the track reconstruction pipeline, following the definitions presented in Section 3.4.

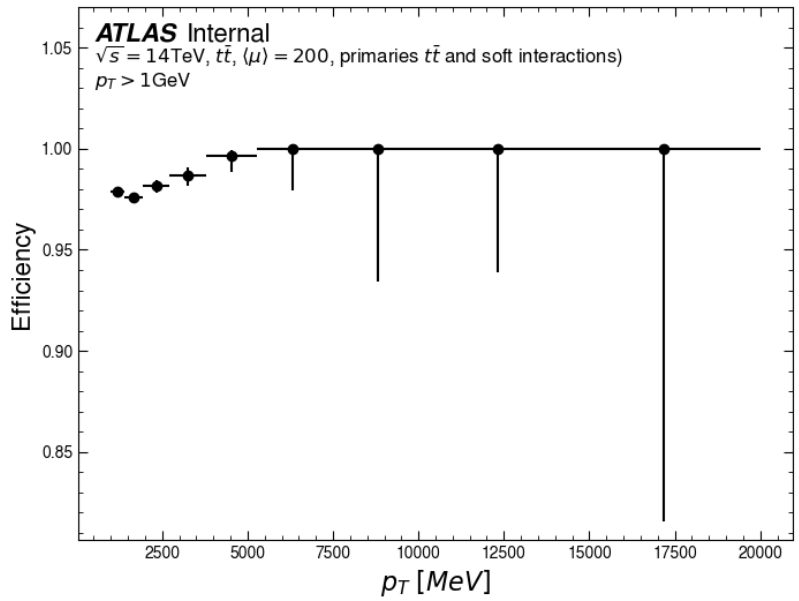


Figure 37: p_T-wise efficiency of the full track reconstruction pipeline. There is a fairly even performance across a range of p_Ts.

	DSP blocks	ALMs	RAM
MLP resources	14.5%	3.4%	26.7%
GNN resources	13.4%	2.0%	6.8%
Total resources	27.9%	5.4%	33.5%

Table 9: Resources used by the MLP and GNN for the full track reconstruction pipeline.

The HLS model can be run on a set of testbench data to generate predictions. I compared

these predictions to the ones from the PyTorch model. The discrepancy (d) between the two sets of output was calculated by:

$$d = mean\Big(\frac{2 \times |P_{HLS} - P_{Torch}|}{|P_{HLS} + P_{Torch}|}\Big) \tag{11}$$

where P_{HLS} and P_{Torch} refer to the predictions from the HLS and PyTorch models, respectively. The discrepancy for the MLP was 0.9%. For the GNN, it was 1.5%.

The graphs used in this pipeline had an average of 18,141 nodes. The number of nodes is determined by the number of hits, and therefore remains constant across the entire pipeline. The number of edges was on average 101,420. I refer here to the edges that were established by the Metric Learning algorithm, and thus also the number of edges receiving labels from the GNN. Had the purity of the Metric Learning stage been higher, there would be fewer edges in the graphs. For the two first stages of the pipeline, graph sizes do not influence the size of the model (and hence occupancy on the FPGA). For the track reconstruction stage, however, the sizes of the graphs are used to determine the architecture of the Walkthrough algorithm. The resource estimates for stage 3 are therefore influenced by the number of nodes and edges in the graphs.

7.4.1 Including Walkthrough estimates

In the sample pipeline I present, I am using the Connected Components method to build track candidates. Since this method has not been implemented on an FPGA, resource estimates for this stage of the pipeline were not included in the table above. We do, however, have resource estimates for the VHDL implementation of the Walkthrough method, as presented in Section 4.6. While there is no measurement of the pipeline's performance using this method, the resource estimates allow us to gauge the occupancy of the full pipeline on an FPGA. Table 10 summarises the resource estimates for the MLP, simplified GNN, and the Walkthrough implementation.

	DSP blocks	ALMs/ALUTs	RAM
MLP + GNN	27.9%	5.4%	33.5%
Walkthrough	0%	69.7%	0.3%
Total resources	27.9%	75.1%	33.8%

Table 10: Total resources used when including the VHDL implementation of the Walkthrough algorithm. DSP block and RAM block usage remain under the 40% target. Due to the Walkthrough's heavy use of ALUTs (and hence ALMs), this resource is used beyond the 40% target.

7.5 Removing the p_T cut

To establish a track reconstruction pipeline that can be implemented in the new Event Filter, it is ultimately necessary that it performs well on entire datasets without making

cuts on the data. To be able to process such large datasets, the models need to be trained on large GPUs. The GPU I used was the Quadro P2000 [48], which has a capacity of 4 Gb. To be able to train the pipeline on datasets with no p_T cut on this GPU, I reduced the sizes of the Metric Learning and GNN algorithms significantly. The Metric Learning algorithm of the pipeline contained an MLP with 16 hidden dimensions, and the model was pruned by 24%. The pruning frequency was increased to 10, allowing models to re-gain more of their performance before the next pruning iteration. The "KNN value", which specifies the maximum amount of neighbouring nodes the FRNN algorithm can construct edges with, was reduced from 50 to 10 during training, and the KNN value for inference was reduced from 800 to 50. This means that the algorithm could only build edges between nodes and their closest 50 neighbours in latent space, even if more than 50 nodes were within the FRNN radius. The GNN had 8 hidden dimensions, 1 message passing step, and was not pruned. Both the GNN and MLP were trained for 500 epochs.

While it was possible to remove the p_T cut, the GPU could not handle datasets when noise was added, even when reducing model sizes to their absolute minimum. I trained the size-reduced pipeline on data both with and without a p_T cut. This was thus a test to gauge whether we can expect performance and/or resource estimates to change when removing data cuts.

In Table 11, I summarise the efficiency and fake rate of the track reconstruction pipeline, along with the resource usage for training on data with and without cuts. It is evident from these results that removing the cut on the p_T in the datasets causes a decrease in the efficiency of the model, and an increase in fake rate. Since the model is very small, it can have a hard time learning the characteristics of a more diverse dataset, i.e. a dataset with a wider range of p_T values. The gap in efficiency is therefore to be expected. Whether this gap closes for larger models has yet to be explored.

The resource estimates, on the other hand, remain roughly the same. This is also to be expected, since the amount of data processed by the model does not influence the size of the model. A larger dataset merely affects the values of the model's parameters, while the number of parameters affects the occupancy on an FPGA.

Cut	0 GeV	1 GeV
Efficiency	75.5%	98.2%
Fake rate	44.8%	17.9%
DSP usage	14.8%	14.8%
ALM usage	2.6%	2.6%
RAM usage	9.1%	9.0%

Table 11: A comparison of performance and resource usage between pipelines trained on data with and without a p_T cut. Here, the resource usage reported is combined for the MLP and GNN.

7.6 ITk pipeline: Summary and discussion

The ITk pipeline was trained on 100 simulated ITk events. The events were from $pp \rightarrow t\bar{t}$ collisions with a pileup of 200 to mimic the data we expect to see in the HL-LHC. It was possible to construct a full pipeline with a track reconstruction efficiency of 97.9% when trained on the events with a 1 GeV cut applied. The machine learning parts of this pipeline occupied 27.9% of DSP blocks, 5.4% of ALMs and 33.5% of RAM.

7.6.1 Resource studies

Behind the construction of this pipeline were studies on how model architectures influence its resource usage on the Intel Stratix10 GX FPGA and its performance on a GPU. First of all, I found that the size of the model, adjusted by increasing the width of its hidden layers, led to a linear increase of FPGA resources used. This was the case for both the MLP and the GNN. For the initial studies, before pruning was implemented, DSP blocks were the most used resource. This is because the parameters of machine learning models are stored in tensors, which are multiplied when using the model to run inference. These types of operations are typically handled by DSP blocks.

7.6.2 Pruning studies

When I implemented pruning into the training of the models, the pattern changed. First of all, I saw a close-to-linear decrease in the amount of resources used when the pruning amount was increased. This was especially the case for DSP blocks, which saw a bigger decrease in resources compared to RAM and ALM usage. Since pruning sets a given percentage of a model's parameters to zero, the amount of tensor multiplication operations also decreases. For the larger models, the decrease in DSP blocks meant that RAM became the most predominantly used resource.

7.6.3 Performance studies

There is a trade-off between model size (and hence compatibility with FPGAs) and its performance. For both the MLP and the GNN, I found there to be a correlation between their sizes, and their efficiency and purity. I however also found that performance can, to a certain point, be maintained during pruning - for example, a 32-dimensional model maintains its performance until 78% pruning, as presented in Fig. 35. In general, the larger the model, the more it can be pruned before efficiency and purity decrease. This can be attributed to the fact that a larger model will have more non-zero parameters remaining after pruning, which can be re-trained to still give accurate predictions. The GNN was more sensitive to pruning, meaning that it could be pruned less than a similar-sized Metric Learning MLP before performance decreased.

7.6.4 The full pipeline

The resource and performance studies were then coupled by using the results to propose the architecture of a full track reconstruction pipeline, which I deem possible to implement on the target device. Here, the model sizes and their amount of pruning were balanced, such that neither DSP blocks nor RAM blocks were over-used compared to the other.

Since I have focused on implementing the pipeline's machine learning components onto an FPGA, there are still studies to be done regarding the size and resource usage of the remaining pipeline. A first estimate of its 3rd stage, track reconstruction, was presented in Section 4.6. The current implementation of stage 3 uses primarily ALMs, whereas ALM usage remains low for the first two stages of the pipeline. In Section 8, I give my suggestions for further studies on the pipeline's FPGA implementation.

As a final step, I began the work of adjusting the pipeline to run on full events, i.e. without cuts being made to the data. While it was not possible to train the pipeline on events with their, on average, 55% noise, I trained a small pipeline without a cut on the transverse momentum. This led to a clear decrease in efficiency when comparing to the same (small) pipeline trained on data with a cut. I did not make any exhaustive attempts at increasing this efficiency; however training larger models, and for longer, could potentially yield higher track reconstruction performance on full events. The FPGA resource usage, on the other hand, did not change significantly between the two pipelines.

8 Future work

This section contains a list of tests I suggest be conducted in continuation of this project.

- **Training on full events.** In this project, I trained a small version of the track reconstruction pipeline on 100 events without a p_T cut. For an even more realistic study on the pipeline's performance, a bigger pipeline (within the limits of the FPGA) should be trained on more data. Here more data refers both to more events in the datasets, as well as adding noise back in. This would require access to a large GPU, and ability to train for many more epochs. The pipeline running on full events could also benefit from further optimisation studies to make it suited for a more diverse dataset.

- **Full GNN resource estimates.** Once support for aggregation functions and indexing operations is added to HLS4ML, the resource estimates of the full GNN can be tested. Until then, one could test on a GPU the occupancy of the full GNN and compare it to the simplified GNN.

- **Resource usage in Event Filter.** In this project, I have worked with data in the form of hits. As mentioned in Section 1.1, the Event Filter uses a three-step procedure to perform tracking. The resource usage of the first and the final steps has not been explored in this thesis. To gain a better understanding of the resource usage for the entire Event Filter tracking process, this will be necessary.

- **Sectioning.** The events used in the pipeline contained hits from the full ITk detector. In reality, breaking the data into smaller detector sections, as was done for the TrackML pipeline, could be a way to reduce the stress on the FPGAs in use.

- **Impacts on track reconstruction efficiency.** The concrete effects of efficiency and purity of the two first stages in the pipeline on track reconstruction efficiency could be tested. Whether the track reconstruction efficiency is more sensitive to efficiency or purity of the first pipeline stages is currently not known.

- **Quantisation aware training.** Along with pruning, quantisation aware training (QAT) could be implemented to reduce model sizes. Since model parameters are quantised when implemented onto an FPGA device, training can be adapted to make sure that quantisation favours a size reduction while maintaining efficiency.

- **Fixed point precision.** The precision of the fixed points, as discussed in Section 4.3, influences the accuracy of the HLS model's predictions when compared to the PyTorch predictions. Its influence on the model's resource usage on the FPGA is however not known and could be studied further.

- **Accuracy on FPGA.** While the HLS compilation ran a simulation on some test-bench data, this has not been done in Quartus. Whether the model implemented in an FPGA can produce predictions identical or similar to the PyTorch model running on the GPU should be tested.

- **Integration with Intel OneAPI.** During this project, I was in contact with Intel regarding running the HLS code as a kernel in OneAPI [38]. This proved not to be possible currently, however there are ongoing discussions with Intel engineers. One recommended solution is to rewrite the pipeline code with SYCL for easier integration with OneAPI.

9 Conclusion

In this project, I tested the FPGA implementation of a three-stage GNN-based track reconstruction pipeline for tracking in the ATLAS TDAQ system for the HL-LHC upgrade. I developed pipelines for two different datasets: TrackML for an initial assessment of the track reconstruction pipeline, and simulated ITk data for more realistic estimates of the pipeline's performance in the new ITk detector. With the goal of implementing the pipeline's machine learning components onto an Intel FPGA, I translated the Python code to high level synthesis, and compiled the code with Intel Quartus to obtain resource estimates. I found there to be a direct correlation between the size of the models and their FPGA resource usage. This imposes a limit on the size of a track reconstruction pipeline that can run on an FPGA.

Simultaneously, I studied the performance of the track reconstruction pipeline, and implemented pruning methods for decreasing model sizes while maintaining their accuracy.

Depending on the size of the model, it is possible to prune models by up to around 80% while both preserving efficiency, and being able to fit them onto the target device. While unpruned models occupy mainly DSP blocks, pruned models tend to use more RAM blocks than DSP blocks.

Based on efficiency and resource studies, I constructed a sample pipeline, which was developed to fit onto the target device Intel Stratix 10 GX FPGA. It was trained on data subject to a 1 GeV cut on the transverse momentum. The track reconstruction efficiency presented in the sample pipeline is 97.9%. In Section 3.4.5 I defined a goal of improving the efficiency compared to a CPU-based demonstrator algorithm, which had an average efficiency of around 90%. The sample pipeline achieves this goal. When removing the p_T cut, however, the efficiency dropped to 75.5%, which is below that of the base-line algorithm. To improve on this result, further studies into both the architecture of the pipeline and methods for minimising its size are required.

10 Acknowledgements

There are a number of people who have helped greatly in making this project possible. First of all, I would like to extend a general thank you to the "GNN for EF tracking" community for their inputs and very useful discussions. To Haider Abidi for his support and dedication to help solve any problem that came along during the project. My deepest gratitude to Daniel Murnane for continuously providing his insights and perspectives that have significantly broadened my understanding of GNNs and high-energy physics in general. And lastly, a special thank you to my advisor, Alessandra Camplani, who has been exceptionally supportive and enthusiastic during the entire project.

References

[1] CERN. The large hadron collider. https://home.web.cern.ch/science/accelerators/large-hadron-collider. Accessed on 10th August 2023.

[2] Werner Herr and Bruno Muratori. Concept of luminosity. https://cds.cern.ch/record/941318/files/p361.pdf. Accessed on 23rd August 2023.

[3] CERN. High-luminosity lhc. https://home.cern/science/accelerators/high-luminosity-lhc. Accessed on 14th August 2023.

[4] CERN. Atlas experiment. https://atlas.cern/. Accessed on 14th August 2023.

[5] The ATLAS Collaboration. Clustering and tracking in dense environments with the atlas inner tracker for the high-luminosity lhc. 2023.

[6] Peter Jenni, Marzio Nessi, Markus Nordberg, and Kenway Smith. *ATLAS high-level trigger, data-acquisition and controls: Technical Design Report*. Technical design report. ATLAS. CERN, Geneva, 2003.

[7] Laura Gonella. The atlas itk detector system for the phase-ii lhc upgrade. *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment*, 1045:167597, 2023.

[8] ATLAS TDAQ Collaboration. Technical Design Report for the Phase-II Upgrade of the ATLAS Trigger and Data Acquisition System - EF Tracking Amendment. Technical report, CERN, Geneva, Sep 2021. For internal review by ATLAS TDAQ.

[9] Alessandra Camplani, S. Dittmeier, A. Annovi, K. Axiotis, Roberto Beccherle, N. Biesuz, R. Brenner, S. Débieux, Mattias Ellert, P. Francavilla, P. Giannetti, Konstantinos Kordas, M. Mårtensson, P. Mastrandrea, C. Noulas, J. Oechsle, Marco Piendibene, R. Poggi, A. Schöning, and J. Zinßer. Intel stratix 10 fpga design for track reconstruction for the atlas experiment at the hl-lhc. 02 2023.

[10] Benjamin Sanchez-Lengeling, Emily Reif, Adam Pearce, and Alexander B. Wiltschko. A gentle introduction to graph neural networks. https://distill.pub/2021/gnn-intro/, Sep 2021. Accessed on 14th August 2023.

[11] Intel. 3.1. fpga architecture overview. https://www.intel.com/content/www/us/en/docs/programmable/683152/22-1/fpga-architecture-overview.html. Accessed on 10th August 2023.

[12] Intel. Stratix 10 gx fpga product information. https://www.intel.com/content/www/us/en/products/details/fpga/stratix/10/gx.html. Accessed: July 23, 2023.

[13] Sylvain Caillou, Paolo Calafiura, Charline Rougier, Jan Stark, Daniel Thomas Murnane, Alexis Vallier, Xiangyang Ju, and Steven Andrew Farrell. Atlas itk track reconstruction with a gnn-based pipeline. `https://cds.cern.ch/record/2815578`, Jul 2022.

[14] Kaggle. TrackML Particle Tracking Challenge. `https://www.kaggle.com/c/trackml-particle-identification`, N.D. Accessed: June 7, 2023.

[15] Analysis Software Group. Pileup analysis sequence. `https://atlassoftwaredocs.web.cern.ch/AnalysisSWTutorial/cpalg_prw/`, 2023. Accessed on 23rd August 2023.

[16] Sabrina Amrouche, Laurent Basara, Paolo Calafiura, Victor Estrade, Steven Farrell, Diogo R. Ferreira, Liam Finnie, Nicole Finnie, Cécile Germain, Vladimir Vava Gligorov, and et al. The tracking machine learning challenge: Accuracy phase. *The NeurIPS '18 Competition*, page 231–264, 2019.

[17] Daniel Thomas Murnane. Graph Neural Networks for High Luminosity Track Reconstruction. Graph Neural Networks for High Luminosity Track Reconstruction. 2022.

[18] Abdelrahman Elabd, Vesal Razavimaleki, Shi-Yu Huang, Javier Duarte, Markus Atkinson, Gage DeZoort, Peter Elmer, Scott Hauck, Jin-Xuan Hu, Shih-Chieh Hsu, Bo-Cheng Lai, Mark Neubauer, Isobel Ojalvo, Savannah Thais, and Matthew Trahms. Graph neural networks for charged particle tracking on fpgas. *Frontiers in Big Data*, 5, 2022.

[19] HSF Reconstruction and Software Triggers. Tracking-ml-exa.trkx. `https://github.com/HSF-reco-and-software-triggers/Tracking-ML-Exa.TrkX`, 2023. Accessed: June 7, 2023.

[20] Markus Julian Atkinson, Sylvain Caillou, Paolo Clafiura, Christophe Collard, Steven Andrew Farrell, Benjamin Huth, Xiangyang Ju, Ryan Liu, Tuan Minh Pham, Daniel Murnane (corresponding author), Mark Neubauer, Charline Rougier, Jan Stark, Heberth Torres, and Alexis Vallier. gnn4itk. `https://gitlab.cern.ch/gnn4itkteam/commonframework`. Accessed on 19th May 2023.

[21] Xiangyang Ju, Daniel Murnane, Paolo Calafiura, Nicholas Choma, Sean Conlon, Steve Farrell, Yaoyuan Xu, Maria Spiropulu, Jean-Roch Vlimant, Adam Aurisano, Jeremy Hewes, Giuseppe Cerati, Lindsey Gray, Thomas Klijnsma, Jim Kowalkowski, Markus Atkinson, Mark Neubauer, Gage DeZoort, Savannah Thais, Aditi Chauhan, Alex Schuy, Shih-Chieh Hsu, and Alex Ballow. Performance of a Geometric Deep Learning Pipeline for HL-LHC Particle Tracking. *arXiv e-prints*, page arXiv:2103.06995, March 2021.

[22] Biscarat, Catherine, Caillou, Sylvain, Rougier, Charline, Stark, Jan, and Zahreddine, Jad. Towards a realistic track reconstruction algorithm based on graph neural networks for the hl-lhc. *EPJ Web Conf.*, 251:03047, 2021.

[23] Sylvain Caillou, Paolo Calafiura, Steven Andrew Farrell, Xiangyang Ju, Daniel Thomas Murnane, Charline Rougier, Jan Stark, and Alexis Vallier. ATLAS ITk Track Reconstruction with a GNN-based pipeline. Technical report, CERN, Geneva, 2022.

[24] ATLAS Collaboration. Athena. https://doi.org/10.5281/zenodo.2641997, April 2019.

[25] Collaboration ATLAS. Technical Design Report for the Phase-II Upgrade of the ATLAS Trigger and Data Acquisition System - Event Filter Tracking Amendment. Technical report, CERN, Geneva, 2022.

[26] Alejandro Villalpando. Track Reconstruction with GNN and CNN, 2022.

[27] Analytics Vidhya. Basic introduction to feed-forward network in deep learning. https://www.analyticsvidhya.com/blog/2022/03/basic-introduction-to-feed-forward-network-in-deep-learning/. Accessed on 9th September 2023.

[28] PyTorch Geometric. pool.radius. https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.pool.radius.html#torch_geometric.nn.pool.radius. Accessed on 14th August 2023.

[29] Peter W. Battaglia, Razvan Pascanu, Matthew Lai, Danilo Rezende, and Koray Kavukcuoglu. Interaction networks for learning about objects, relations and physics, 2016.

[30] Pauli Virtanen, Ralf Gommers, Travis E. Oliphant, Matt Haberland, Tyler Reddy, David Cournapeau, Evgeni Burovski, Pearu Peterson, Warren Weckesser, Jonathan Bright, Stéfan J. van der Walt, Matthew Brett, Joshua Wilson, K. Jarrod Millman, Nikolay Mayorov, Andrew R. J. Nelson, Eric Jones, Robert Kern, Eric Larson, C J Carey, İlhan Polat, Yu Feng, Eric W. Moore, Jake VanderPlas, Denis Laxalde, Josef Perktold, Robert Cimrman, Ian Henriksen, E. A. Quintero, Charles R. Harris, Anne M. Archibald, Antônio H. Ribeiro, Fabian Pedregosa, Paul van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[31] SciPy. scipy.sparse.csgraph.connected_components. https://docs.scipy.org/doc/scipy/reference/generated/scipy.sparse.csgraph.connected_components.html. Accessed on 9th September 2023.

[32] Exa.TrkX Team. Tracking ml with exa.trkx: matching definitions. https://hsf-reco-and-software-triggers.github.io/Tracking-ML-Exa.TrkX/performance/matching_definitions/, 2023. Accessed on 22nd July 2023.

[33] Technical Design Report for the ATLAS Inner Tracker Pixel Detector. Technical report, CERN, Geneva, 2017.

[34] Robert Keim. What is a hardware description language (hdl)? - technical articles. https://www.allaboutcircuits.com/technical-articles/what-is-a-hardware-description-language-hdl/, Mar 2020. Accessed 14th August 2023.

[35] Intel. Fpga design software - intel® quartus® prime. https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html. Accessed 14th August 2023.

[36] Cadence. High-level synthesis (hls) overview — high-level synthesis tools — cadence. https://www.cadence.com/en_US/home/explore/high-level-synthesis.html. Accessed 14th August 2023.

[37] Peter J. Ashenden. *The Designer's Guide to VHDL*. Morgan Kaufmann Publishers, 2011.

[38] Intel. Oneapi: A new era of heterogeneous computing. https://www.intel.com/content/www/us/en/developer/tools/oneapi/overview.html. Accessed on 14th August 2023.

[39] Intel. M20k memory block definition. https://www.intel.com/content/www/us/en/programmable/quartushelp/17.0/reference/glossary/def_m20k.htm. Accessed on 14th August 2023.

[40] Intel. Intel fpga product catalog. https://cdrdv2-public.intel.com/730595/Intel-fpga-product-catalog-23.1.pdf. Accessed on 16th August 2023.

[41] FastML Team. hls4ml. https://github.com/fastmachinelearning/hls4ml.

[42] Javier Duarte et al. Fast inference of deep neural networks in FPGAs for particle physics. *JINST*, 13(07):P07027, 2018.

[43] Junjie Bai, Fang Lu, Ke Zhang, et al. Onnx: Open neural network exchange. https://github.com/onnx/onnx, 2019.

[44] PyTorch. Hingeembeddingloss. https://pytorch.org/docs/stable/generated/torch.nn.HingeEmbeddingLoss.html. Accessed on 14th August 2023.

[45] PyTorch. Bceloss. https://pytorch.org/docs/stable/generated/torch.nn.BCELoss.html. Accessed on 14th August 2023.

[46] PyTorch. Adamw. https://pytorch.org/docs/stable/generated/torch.optim.AdamW.html. Accessed on 14th August 2023.

[47] Intel. Fpga architecture white paper. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wp-01003.pdf, 2006. Accessed on 16th August 2023.

[48] Nvidia. Datasheet quadro p2000 - nvidia. https://www.nvidia.com/content/dam/en-zz/Solutions/design-visualization/productspage/quadro/quadro-desktop/quadro-pascal-p2000-data-sheet-us-nvidia-704443-r2-web.pdf. Accessed on 14th August 2023.