Project Report

Abhinav M

<abnvm1@gmail.com>

<23pmmt11@uohyd.ac.in>

MV487 - Semester Project II

Center for Advanced Studies in Electronic Science and Technology

May 13, 2024

Contents

1 Introduction **4**
1.1 The Muon Collider . . . 5
1.2 Beam Induced Background (BIB) . . . 5
1.2.1 Mitigating BIB . . . 5
1.3 Applied ML for Science . . . 6

2 Objectives **6**

3 Literature Review **6**
3.1 Opportunities for ML in BIB subtraction . . . 7
3.2 Hardware Implementation of Deep Learning Inference . . . 9
3.2.1 Neural Network Training and Optimization . . . 9
3.2.2 hls4ml . . . 10

4 Methodology **11**
4.1 Data Collection . . . 12
4.2 Data pre-processing . . . 12
4.3 Feature Selection . . . 13
4.4 Neural Network Architecture . . . 16
4.5 RTL Synthesis . . . 17

5 Flowchart: GEANT4 Simulation to Vivado Project **18**

6 Results and Discussion **19**
6.1 HLS Code . . . 19
6.1.1 Timing . . . 19
6.1.2 Utilization . . . 19
6.1.3 Timing and Utilization of Converted Verilog code . . . 20
6.2 Vivado Project on part xcu250-figd2104-2L-e . . . 21
6.2.1 Timing . . . 21
6.2.2 Utilization . . . 21
6.2.3 Synthesized design . . . 21
6.2.4 Implemented design . . . 22
6.2.5 Power . . . 22
6.3 Key Findings . . . 23

7 Conclusion **23**

8 Future Work **23**

9 Appendix A **25**
9.1 Usage Instructions for Code . . . 25
9.1.1 Parsing the data into a pandas dataframe . . . 25
9.1.2 Training models, plotting and hls4ml usage . . . 25
9.2 Converting .slcio into .txt files . . . 26
9.3 Parsing .txt files . . . 26
9.4 Code to train the model . . . 37
9.5 Using hls4ml . . . 45

Information

- **Name**: Abhinav M
- **Enrollment No.**: 23pmmt11
- **Programme**: M.Tech Microelectronics and VLSI Design
- **Course**: MV487 - Semester Project II
- **Project Title**: Development of Low-latency Hardware Implementations of Deep Learning Models for Beam-induced Background Subtraction in Frontend Electronics of Muon Collider
- **Supervisor**: Dr. Bhawna Gomber
  - **Institution**: Center for Advanced Studies in Electronic Science and Technology (CASEST), School of Physics, University of Hyderabad
- **Start Date**: 16 October 2023

Development of Low-latency Hardware Implementations of Deep Learning Models for Beam-induced Background Subtraction in Frontend Electronics of Muon Collider

Executive Summary

Due to the challenges in scaling up proton colliders like the LHC for future high energy high luminosity particle colliders, muon colliders are seen as a promising alternative due to their ability to fulfil the energy and luminosity requirements while being relatively compact and the absence of any significant technological roadblocks for their conception. However there are several challenges that are being actively researched to improve the performance of future muon colliders, subtracting Beam Induced Background (BIB) from the signal data is one such challenge. BIB is the ensemble of decay products that result from the interaction of muons with the surrounding matter that makes up the collider. Traditional approaches to filter BIB using hard filters may inadvertently filter out unknown physics data. BIB subtraction is a fertile field to apply an ML based approach. However this requires to comply with on detector processing with timescales of a few nanoseconds to microseconds.These timing constraints push us towards a highly optimised implementation of a deep learning model on hardware which is the focus of this project. Starting with GEANT4 simulation files, entire flow of FPGA implementation of a deep learning model using TensorFlow and hls4ml is demonstrated in this project.

1 Introduction

The main aim of particle colliders is to unearth radically new science and explore unknown areas in our understanding of physics rather than adding incremental knowledge to inevitable discoveries. Researchers aim to attain two crucial characteristics in a particle collider: high luminosity and high collision energy. Luminosity refers to the frequency of collisions happening in the collider. A high luminosity gives us more data points and hence we will have more data to work with and in addition maximises our chances to observe and study statistically rare phenomenon. Higher energies help us probe at smaller length scales and also results in generation of heavy exotic particles that only exist in extreme conditions like during the big bang. However there are challenges that come up when we try to increase either of these parameters in present day proton-proton colliders like the LHC. Some of these issues are discussed below:

- Because protons are charged particles, when they accelerate, they loose some energy in the form of synchrotron radiation. This is very significant as beam energy is increased. A linear accelerator circumvents synchrotron radiation but there each beam can only be used once.
- To increase the energy of the beams further, the accelerating loops of the future colliders will need to get bigger and bigger which is an expensive affair.
- Since protons are composite particles, during collision the energy of the constituent particles will be much lower.
- The beam-beam effects due to electromagnetic forces between protons at high energies make it difficult to control the beam.

These issues among others create technological and economic barriers to envision a larger proton collider based on the LHC for future high energy - high luminosity studies without significant technological breakthroughs in the very near future. Hence we will see how a Muon collider is a very promising alternative in the next section.

1.1 The Muon Collider

Muons are chargeless point particles with a mass over 206 times that of a proton. Immediately it is clear that muons can be accelerated to higher energies and luminosities without much of the issues that proton colliders face, all whilst remaining relatively compact. Muon colliders have been proposed long back and are not a new idea by any means, but the current flurry of interest stems mainly from three factors: one is the earlier mentioned difficulty in envisioning any major breakthroughs that make a larger proton collider feasible, another factor is the technological advancements over the past half century that have inspired confidence in the technical feasibility of a muon collider, and the last is the advantages of using muons over protons.

However there are still some challenges that are actively being researched to overcome in order to develop a muon collider, Beam Induced Background radiation is one such issue that is a focal point of this project and is elaborated in the next section.

1.2 Beam Induced Background (BIB)

Muons are unstable particles, meaning they decay into other products spontaneously with time as well as through interactions with matter. Therefore, it is impossible to study their collision interactions without being exposed to decay products forming as a result of in-flight decays as well interaction of the muon beam with the surrounding matter that make up the collider. The decay products pollute the collider environment. Beam Induced Background (BIB) refers to the collective of these undesired decay products. The amount of BIB generated significantly outweighs the interactions that we wish to observe by many orders of magnitude. Therefore without subtracting or eliminating this effect it is impossible to conceive a muon collider. More specifically the ideal hit occupancy in the tracking system and calorimeter should be a few percent at most, meaning of the total cells at any time only around 1% should fire which corresponds to the interactions as a result of the Muon collisions we wish to study. Without subtracting BIB this low hit occupancy is not possible. The same is true for jet reconstruction at offline stages of processing where it is crucial to differentiate between true signals of collision products and fake jets due to BIB.

Nonetheless, BIB has two differentiating characteristics from the original muon beam and its collision products that we wish to observe: BIB is composed of low energy particles with a broad range of arrival times at the detector. These two characteristics will play an important role in helping us differentiating the true signals from BIB. Predominantly, BIB affects three components of the experiment: the muon beam, the tracking system and the calorimeter. We will see some strategies that are put forth for minimizing BIB at each of these junctures in sec 3.1. It is important to note that on-detector filtering is our priority as it is not physically possible to record or process after the fact the vast quantities of data that is generated at each collision event.

1.2.1 Mitigating BIB

As eluded to earlier, the target beams are polluted by BIB due to interactions with the surrounding accelerator walls and other matter around the beam. To minimize this decay, tungsten cone structures can be used leading up to the collision center to guide the beams which help to minimize BIB concentration in the incident beams especially in the high energy range. Similarly, careful selection of materials used at different parts of the collider play a vital role in the amount and characteristics of the BIB generated. Even with best material selection, a large enough BIB still exists, and hence other filtering techniques have to be employed in conjunction. These filtering techniques rely on the characteristic differences between behavior of the BIB and the target signal. More details on the characteristics of BIB that filtering techniques can leverage are discussed in sec. 3.1.

That being said, it is of equal concern that we do not aggressively over filter the data. This will result in missing out on new undiscovered physics like interactions at low energies, those with long lifetimes and anything that gets filtered out as BIB due to blanket filtering. This is where intelligent sensors and systems can be employed such as those based on on-chip Machine Learning. Ultimately it is a balance between complexity, power consumption, cost, quality of data and a reduced data rate that we are left to consider.

1.3 Applied ML for Science

Machine Learning has found its place in many scientific disciplines in various capacities including making predictions using historical data, accelerating signal processing, anomaly detection, event classification etc. It is worth mentioning that of all scientific disciplines, physics has seen the most widespread use of applied ML. The complexity of the models employed also range from simple clustering and classification algorithms to complex deep learning networks.

Generally, Machine learning(ML) is a subset of Artificial Intelligence which comprises Deep Learning (DL) and ML although the later terms get used interchangeably. The basis of ML are algorithms that can be used to fit a function corresponding to the underlying data we wish to study, these algorithms leverage mathematical and statistical tools to achieve this. Some of the standard ML algorithms include: SVM, SVR, Random Forest, Linear Regression, Logistic Regression etc. In comparison, Deep Learning is build upon the concept of Neural Networks which were inspired from the functioning of biological neurons. Neural Networks(NN) are powerful tools as they serve as universal function approximators, i.e., with the right kind of NN architecture, any data can be fitted, provided the sample space is sufficiently large to describe the target function. Deep Learning is mostly used for image recognition tasks, however with advances in time and technology it has found applications in many other fields. RNN, CNN, LSTM etc are some types of Neural Networks that are widely used in Deep Learning.

However, DL and ML in general require a large subset of data, substantial processing power and time to train the model depending on the complexity of the algorithm and the nature of the data itself. Inference on a trained model is faster than training by orders of magnitude but there is still a significant delay that will be a hurdle for near real-time, low latency applications. This means it is not generally feasible to train and run models as-is for real-time or resource constrained applications. To take advantage of DL in low latency, low power applications such as in particle colliders, extensive modifications have to be made in the way that we train the model as well as the way we perform inference on it. Another aspect to be noted that is in this case we are motivated to implement the DL models directly in custom hardware via ASICs or FPGAs to minimize hardware latency by taking advantage of maximum parallel processing capabilities. More details are presented in section 3.2.

2 Objectives

The main objectives of this project are as follows:

1. Develop quantized DL models trained to filter out BIB without loss of signal data.
2. Develop low latency hardware implementations of quantized DL models to run on FPGAs.

3 Literature Review

A brief review of literature is presented in this section in two parts concerning BIB subtraction in Muon collider in the context of possible ML applications and Deep Learning for particle colliders.

3.1 Opportunities for ML in BIB subtraction

Different characteristic signals produced by electromagnetic, neutron, and charged hadron backgrounds, along with signals, can be harnessed to develop "smart" sensors capable of distinguishing Between Interaction Background (BIB) and the actual signal. An instance of this is the existing 2-layer track trigger design for the CMS at the LHC, which filters out low transverse momentum (pT) tracks by comparing hits on separate sensor layers. While multi-layer designs face limitations due to complex interconnections and communication overhead, devices with a sufficiently large thickness/pixel pitch ratio enable leveraging distinct pixel pulse shapes and cluster patterns for signal and BIB hits. This information can be employed for an immediate local filter to reject BIB, and there's potential for exploring concepts incorporating on-chip machine-learning techniques to exploit the unique pulse shapes and patterns. Another consideration involves filtering BIB during jet reconstruction to prevent the creation of fake jets, which is a resource-intensive task during offline processing. Track multiplicity can be used as one method to filter out fake jets, given that BIB-induced fake jets typically lack associated tracks.

Some of the factors through which BIB can be characterized and separated in the tracking and calorimetry systems are as follows:

At the tracking system, which is responsible for measuring the trajectories of particles, several strategies can be enforced to mitigate BIB:

- **Time sensitive detectors**
  Each time a bunch of Muons collide, a significant amount of BIB is accompanied. It was found through simulations that removing hits incompatible with the main bunch crossing time could reduce the data load by about a factor of 3.
- **Longitudinal displacement based filtering**
  Most BIB particles enter the detector with a large longitudinal displacement from the collision region. Detectors with excellent directional can be used for filtering the BIB.
- **Clustering**
  Reducing the number of individual hits due to the large amount of BIB and instead using on detector processing to cluster individual hits and fire the detector can reduce the hits readout. Further selection criteria based on cluster shape an also be applied. However this comes at the cost of higher requirement for on detector processing and power consumption.
- **Energy Deposition**
  Each of the backgrounds has a characteristic energy deposition signature. For example neutrons have low, localised energy deposits. On-detector filters could efficiently exploit this property.
- **Correlation between layers**
  This is a powerful handle for background rejection that works by comparing the signatures between layers to select data. But this requires more communication between layers and results in high complexity, more overhead and power consumption.
- **Pulse shape**
  Signals from BIB can come with a variety of angles and may not give the deposit profile and pulse shape of a typical Minimum Ionising Particle (MIP). Appropriate pulse processing can be used to filter out BIB.

The calorimeter is responsible for measuring the energy of the incident particles, main strategies of BIB mitigation at the calorimeter include:

- **High spatial granularity**
  The overlap of BIB particles can produce hits with an energy similar to the signal, making harder to distinguish it from the BIB.
- **Timing based exclusion**
  It was found through simulations that an acquisition window of about 300ps could be applied to remove most of the BIB, while preserving most of the signal. This means a temporal resolution of 100ps.
- **Longitudinal segmentation**
  A fine segmentation of the calorimeter can help in distinguishing the signal showers from the fake showers produced by the BIB by looking at their different energy profiles.

These are all promising avenues to train and implement an on chip machine learning model if accuracy and latency requirements can be met.

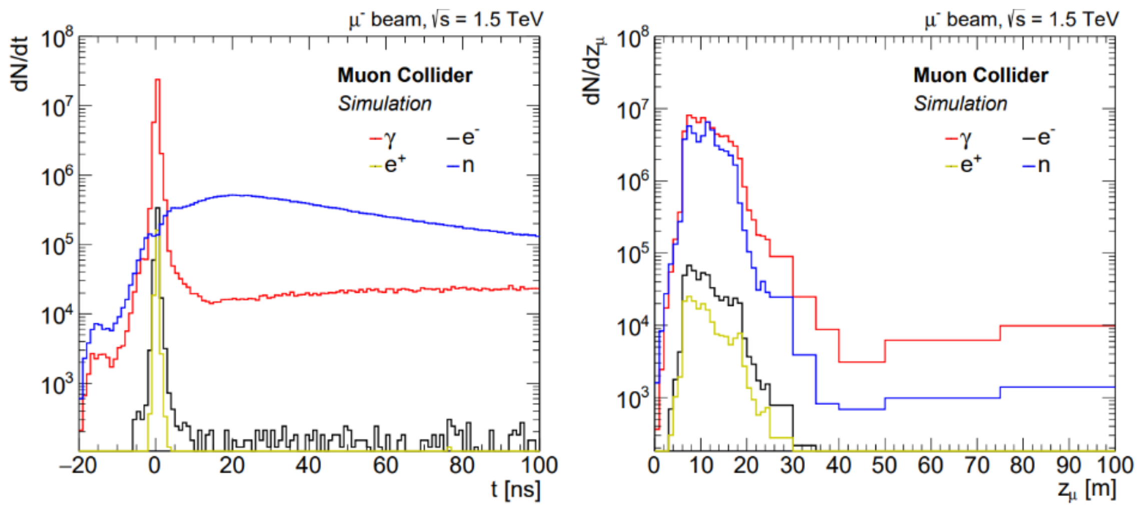


Figure 1: *Time distribution of BIB particles exiting the machine (left) and longitudinal distribution of primary μ-decay generating BIB particles exiting the machine (right). These results are from simulations.*

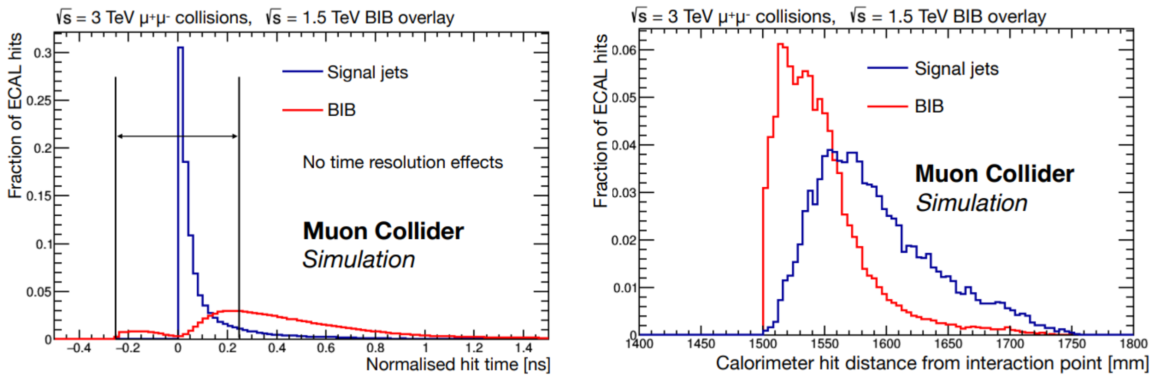


Figure 2: *Normalised hit time in calorimeter barrel, for a signal jet and BIB (left) and the distribution of the barrel hits distance from the interaction point (right).*

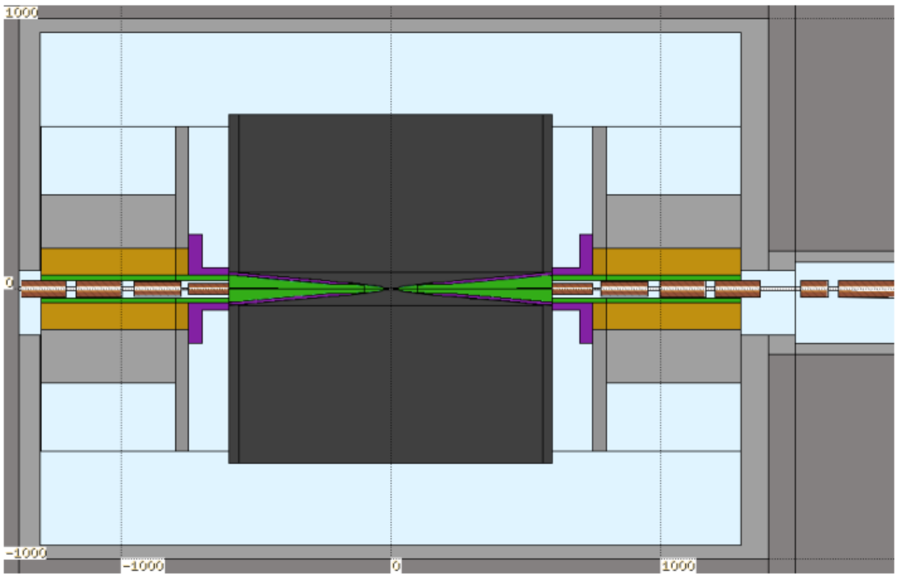


Figure 3: *Cross-sectional view of the Machine–Detector Interface as designed by the Muon Accelerator Program (MAP) collaboration. The tungsten cones leading up to the center of collision are highlighted in green.*

3.2 Hardware Implementation of Deep Learning Inference

Since the processes in particle colliders happen in picosecond to sub microsecond temporal range, any ML/DL based solution that is proposed to be implemented at the frontend systems of the collider will have to be implemented in high-end FPGAs or custom made ASICs optimized for very low latency inference. Hence we have to consider the low latency requirements in every stage of the development and implementation of the DL model. At a high level, FPGA and ASIC algorithm design is different from programming a CPU in that independent operations may run concurrently or fully in parallel. Moreover, independent operations can be pipelined such that the algorithm can accept new inputs while it is still operating on previous inputs.

Development of DL frameworks that can be implemented in resource constrained hardware and finding optimization techniques to achieve low latency are a very active area of research. Below we will see techniques by which a deep learning model can be optimised for fast inference.

3.2.1 Neural Network Training and Optimization

In order to optimize a DL model or NN to achieve low latency inference without loss of accuracy, many strategies exist in literature that are useful for us to employ. Here we will briefly touch upon the main concepts.

- **Quantization-Aware Training**
  Quantization is a way to compress NNs by reducing the number of bits required to represent each weight. For example, by using fixed-point arithmetic, which requires less resources and has a lower latency than floating-point arithmetic. By representing all the inputs, weights, biases, sums, and outputs of each layer as fixed-point numbers, we can speed up operations. It was found that the precision can be reduced significantly without causing a loss in performance. One simple way to reduce a model's size is through post-training quantization (PTQ) where pre-trained model parameters are clipped or rounded to lower precision. However, this process is lossy and sacrifices model performance. To solve this, QAT has been proposed where the quantization is already present at the training phase.

- **Quantization-Aware Pruning**
  Network compression is a common technique to reduce the size, energy consumption, and overtraining of deep NNs. Several approaches have been successfully deployed to compress networks. Parameter pruning is one such approach where some weights are selectively removed based on a particular ranking. This along with QAT can result in significant performance gains.
- **Parallelization and Sparsity**
  Paralellization: The core component of NN layer implementations involves matrix-vector multiplication kernels. In addition to the precision at which these kernels are executed as in QAT there are further configurations that can be used to tune the digital design for low latency. A matrix-vector multiplication kernel requires a number of multiplication operations based on the dimensions of the matrix. The trade off between latency, throughput and FPGA resource usage is determined by the parallelization of the inference calculation and the number of multiplications performed in parallel. A reuse factor of 1 means the operations will be fully parallel, i.e., each multiplier is used once but will require more resources. By adjusting the reuse factor of the multipliers we can further tune and simplify the hardware.

  Sparse operations: By creating a network implementation where the matrix-vector kernel has a large fraction of zero-weights, the computation resources can be greatly reduced.

There are a few python libraries that are specifically used to train quantized NNs, models trained with such strategies include the optimizations mentioned above as well as many other techniques to improve performance in constrained environments. QKeras (an extension of TensorFlow), Pytorch, and (Q)ONNX are some examples of python libraries that support quantized NNs.

3.2.2 hls4ml

Once a DL model is optimized using a quantization framework such as Q-Keras, the next step is implementing the inference of the model in hardware. For transferring into FPGA/ASIC hardware, the mathematical processes required for the inference of the model have to be translated into a suitable hardware description language(HDL). This is a challenging and time consuming process if done manually due to the expertise required to work with HDLs and due to the non-trivial nature of mathematical operations required to perform DL inference.

To address this challenge, researchers at Fermilab and CERN have developed hls4ml, an open-source software-hardware co-design workflow to interpret and translate machine learning algorithms for implementation with both FPGA and ASIC technologies. hls4ml package in combination with TensorFlow and Q-Keras can be used to convert the quantized model into HDL that can be synthesised with HLS software such as HLS Vivado, HLS Vitis and Intel HLS. hls4ml natively supports most NN architectures including Fully connected NN, MLP, LSTM, CNN and GarNet with support for other configurations under development.

4 Methodology

Implementing a low latency hardware based deep learning solution for BIB subtraction involves the following steps in chronological order:

1. **Data collection**
A sufficiently large dataset containing both signal and hits due to BIB has to be collected from experimental data from MAP, simulation studies using GEANT4 or both. Here we use GEANT4.

↓

2. **Data pre-processing**
The dataset should be cleaned and normalized. Any encoding techniques if required need to be done before proceeding.

↓

3. **Feature selection**
Feature selection has to be done to select the best features that describe the target function. This can be done through educated guesses based on theoretical and experimental information and analysing the correlation matrix of the data.

↓

4. **Algorithm selection**
Once appropriate features are decided upon, an appropriate deep learning algorithm such as CNN, RNN, LSTM etc. have to be chosen to train the model. As there is no exact way to choose the best algorithm, experimentation has to be done via trial and error. For simplicity we use a feed forward network in this project.

↓

5. **Training**
The data has to be trained on the selected ML algorithm using TensorFlow and Q-Keras libraries to obtain a quantized model. Cross validation and other techniques should be used to avoid overfitting.

↓

6. **Hyperparameter tuning**
Once the model is trained, further tuning has to be done to find the best set of hyperparameters of the model that gives us the least error.

↓

7. **Translation to HDL**
Once we have the optimized Q-Keras model, using HLS4ML package it is translated into HDL and imported into EDA software, i.e., HLS Vivado.

↓

8. **Synthesis**
The HDL code is now synthesized to get the layout for our target hardware.

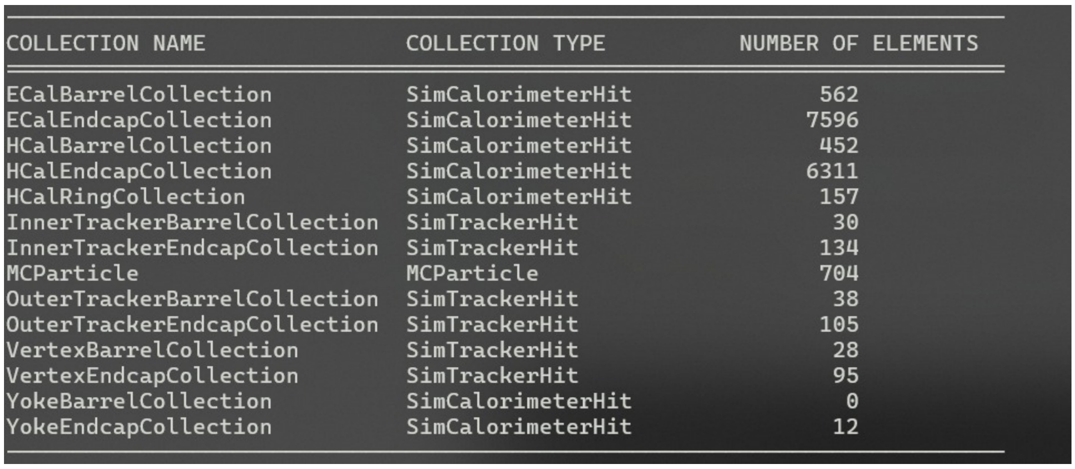
↓

9. **Implementation in FPGA/ASIC**
The synthesized layout is then implemented in a suitable target hardware that meet the performance and latency specifications.

4.1 Data Collection

The data was collected as a result of simulation studies of the Muon collider particle collisions using the GEANT4 montecarlo simulation software. Essentially for this project two sets of data were necessary to train our deep learning model. One set contains only Hard interactions and the other set contains only BIB.

These two datasets were obtained totalling around 500GB of data for 1000 beam collision events. The files for the BIB interactions were in .slcio format, whereas the Hard interaction files were in .stdhep format which were then converted into .slcio.. .slcio files store data in the form of collections. In the case of this particular simulation, there were 13 collections. Each collection corresponds to a different part of the particle colliders structure.

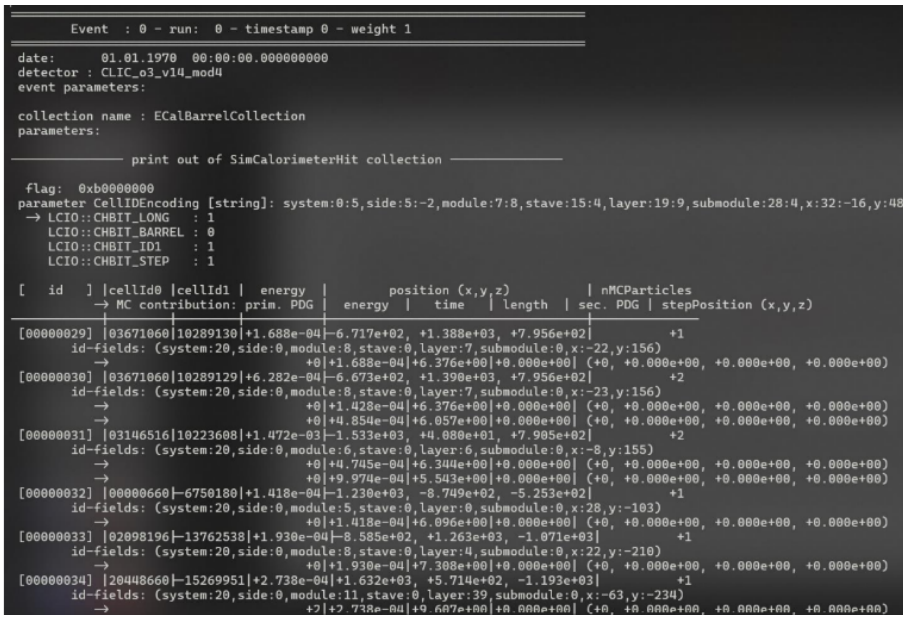


```
COLLECTION NAME                COLLECTION TYPE          NUMBER OF ELEMENTS

ECalBarrelCollection           SimCalorimeterHit           562
ECalEndcapCollection           SimCalorimeterHit          7596
HCalBarrelCollection           SimCalorimeterHit           452
HCalEndcapCollection           SimCalorimeterHit          6311
HCalRingCollection             SimCalorimeterHit           157
InnerTrackerBarrelCollection   SimTrackerHit                30
InnerTrackerEndcapCollection   SimTrackerHit               134
MCParticle                     MCParticle                  704
OuterTrackerBarrelCollection   SimTrackerHit                38
OuterTrackerEndcapCollection   SimTrackerHit               105
VertexBarrelCollection         SimTrackerHit                28
VertexEndcapCollection         SimTrackerHit                95
YokeBarrelCollection           SimCalorimeterHit             0
YokeEndcapCollection           SimCalorimeterHit            12
```

Figure 4: The 13 collections in the GEANT simulation files. Image shows the data in each collection in one of the 1000 hard interaction files.

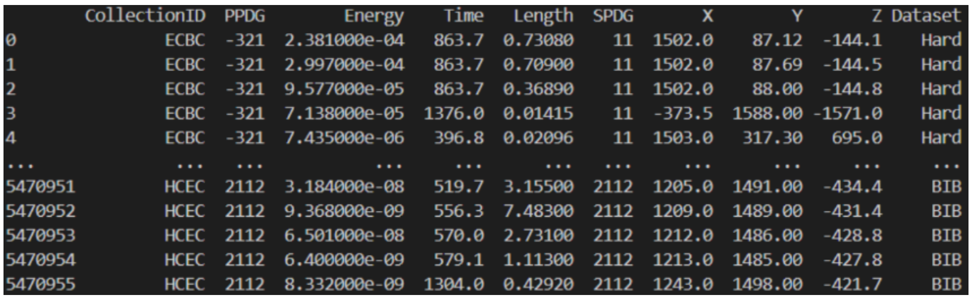
4.2 Data pre-processing

The first task was to extract the information stored in the .slcio files into plain text format so that it can be parsed and loaded into a pandas data frame. The code used for parsing these files and extracting plain-text can be found in Appendix. Once the data was parsed and loaded into the data frame, there were around 5.47 million data points across all collections and half of the data was BIB and the other half was Hard data. Among the 13 collections available, for training the model, it was decided that the ECalBarrelCollection would be used.



```
═══════════════════════════════════════════════════════════════
      Event  : 0 - run:  0 - timestamp 0 - weight 1
═══════════════════════════════════════════════════════════════
date:      01.01.1970  00:00:00.000000000
detector : CLIC_o3_v14_mod4
event parameters:

collection name : ECalBarrelCollection
parameters:

─────────────── print out of SimCalorimeterHit collection ───────────────

 flag:  0xb0000000
parameter CellIDEncoding [string]: system:0:5,side:5:-2,module:7:8,stave:15:4,layer:19:9,submodule:28:4,x:32:-16,y:48
 → LCIO::CHBIT_LONG   : 1
   LCIO::CHBIT_BARREL : 0
   LCIO::CHBIT_ID1    : 1
   LCIO::CHBIT_STEP   : 1

[   id   ] |cellId0 |cellId1 |  energy  |         position (x,y,z)          | nMCParticles
        → MC contribution: prim. PDG |  energy  |   time   | length  | sec. PDG | stepPosition (x,y,z)
───────────┼────────┼────────┼──────────┼───────────────────────────────────┼──────────────
[00000029] |03671060|10289130|+1.688e-04|-6.717e+02, +1.388e+03, +7.956e+02|          +1
       id-fields: (system:20,side:0,module:8,stave:0,layer:7,submodule:0,x:-22,y:156)
        →                            +0|+1.688e-04|+6.376e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
[00000030] |03671060|10289129|+6.282e-04|-6.673e+02, +1.390e+03, +7.956e+02|          +2
       id-fields: (system:20,side:0,module:8,stave:0,layer:7,submodule:0,x:-23,y:156)
        →                            +0|+1.428e-04|+6.376e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
        →                            +0|+4.854e-04|+6.057e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
[00000031] |03146516|10223608|+1.472e-03|-1.533e+03, +4.080e+01, +7.905e+02|          +2
       id-fields: (system:20,side:0,module:6,stave:0,layer:6,submodule:0,x:-8,y:155)
        →                            +0|+4.745e-04|+6.344e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
        →                            +0|+9.974e-04|+5.543e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
[00000032] |00000660|-6750180|+1.418e-04|-1.230e+03, -8.749e+02, -5.253e+02|          +1
       id-fields: (system:20,side:0,module:5,stave:0,layer:0,submodule:0,x:28,y:-103)
        →                            +0|+1.418e-04|+6.096e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
[00000033] |02098196|-13762538|+1.930e-04|-8.585e+02, +1.263e+03, -1.071e+03|          +1
       id-fields: (system:20,side:0,module:8,stave:0,layer:4,submodule:0,x:22,y:-210)
        →                            +0|+1.930e-04|+7.308e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
[00000034] |20448660|-15269951|+2.738e-04|+1.632e+03, +5.714e+02, -1.193e+03|          +1
       id-fields: (system:20,side:0,module:11,stave:0,layer:39,submodule:0,x:-63,y:-234)
        →                            +2|+2.738e-04|+9.607e+00|+0.000e+00| (+0, +0.000e+00, +0.000e+00, +0.000e+00)
```

Figure 5: Contents of one of the .slcio files in plaintext format after conversion using eventdump.



```
        CollectionID  PPDG        Energy    Time    Length  SPDG       X        Y       Z Dataset
0               ECBC  -321  2.381000e-04   863.7  0.73080    11  1502.0    87.12  -144.1    Hard
1               ECBC  -321  2.997000e-04   863.7  0.70900    11  1502.0    87.69  -144.5    Hard
2               ECBC  -321  9.577000e-05   863.7  0.36890    11  1502.0    88.00  -144.8    Hard
3               ECBC  -321  7.138000e-05  1376.0  0.01415    11  -373.5  1588.00 -1571.0    Hard
4               ECBC  -321  7.435000e-06   396.8  0.02096    11  1503.0   317.30   695.0    Hard
...              ...   ...           ...     ...      ...   ...     ...      ...     ...     ...
5470951         HCEC  2112  3.184000e-08   519.7  3.15500  2112  1205.0  1491.00  -434.4     BIB
5470952         HCEC  2112  9.368000e-09   556.3  7.48300  2112  1209.0  1489.00  -431.4     BIB
5470953         HCEC  2112  6.501000e-08   570.0  2.73100  2112  1212.0  1486.00  -428.8     BIB
5470954         HCEC  2112  6.400000e-09   579.1  1.11300  2112  1213.0  1485.00  -427.8     BIB
5470955         HCEC  2112  8.332000e-09  1304.0  0.42920  2112  1243.0  1498.00  -421.7     BIB
```

Figure 6: Text data after parsing, successfully loaded into pandas dataframe. Pandas allows to access and visualize data more conveniently.

4.3 Feature Selection

The available features were: Primary Particle ID, Energy, Time, Length, Secondary Particle ID, X,Y,Z coordinates.

Initially some visualizations were created for the ECalBarrelCollection, the original coordinate representation of one of the events can be seen in Fig.7. Further visualization was made by unrolling the cylinder using a coordinate transformation to project in 2D rectangular coordinates. This was done to better see the distribution of hits and their arrival time. However due to the vast number of events, it was very difficult to consistently identify patterns over the 1000 collision data.

Therefore, next, the data was plotted with respect to different columns to compare the differences between Hard and BIB data. It was expected that there will be a perceivable difference between he two datasets in terms of the energy and/or arrival time characteristics as hinted in literature, however no such obvious pattern was observed from Fig.9. And as expected, due to the lack of distinguishing characteristics a neural network trained on this model was seen to have random guess accuracy (0.5). However, the Primary Particle ID showed that particles present in both BIB and Hard are different as well as exist in different energy ranges which is interesting to note (Fig.10-12), and this feature was highly correlated with whether a given data point is from the BIB dataset or Hard dataset. Training a model on this feature alone gave an accuracy of over .84. However, it was later noted that this particle ID information will not be available in real-time and hence cannot be used to fulfil the objectives of this project.

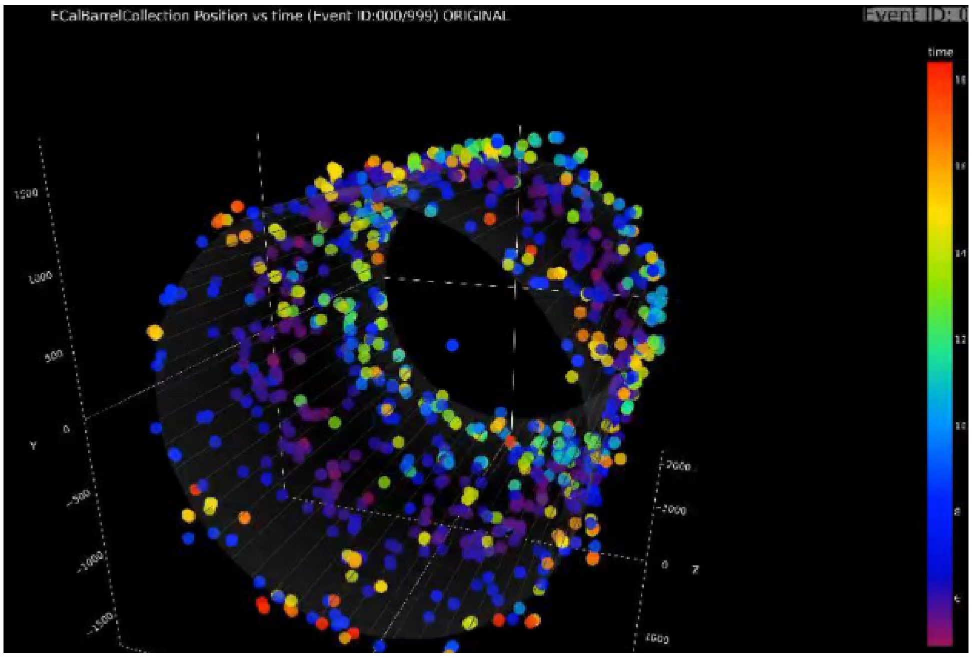


Figure 7: Arrival time as heatmap vs X,Y,Z coordinates for BIB Event 0 out of 1000.

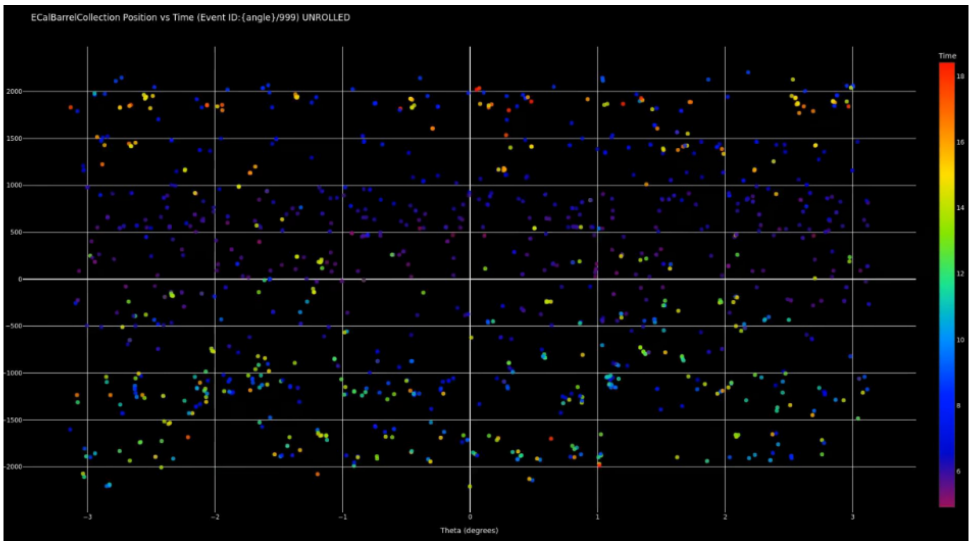


Figure 8: Arrival time as heatmap vs transformed 2D coordinates for BIB Event 0 out of 1000.

Hence to proceed further, a synthetic database representing the noisy BIB characteristic was generated to successfully train a model to distinguish Hard interactions and BIB. This synthetic database was generated by manipulating the coordinates of the original BIB dataset to fit over a displaced Gaussian distribution over the expected range of different features. The model trained to distinguish the Hard database from the synthetic database of BIB gave an accuracy of over .98 as seen in the next section.

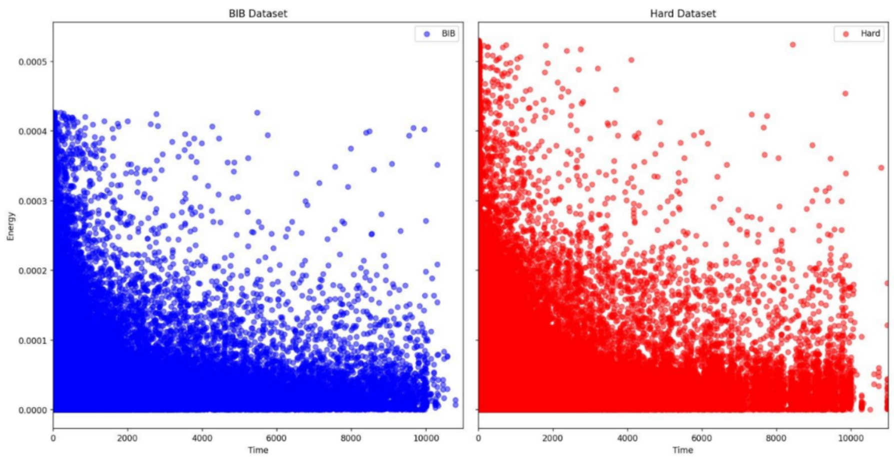


Figure 9: Energy vs Arrival time comparison.

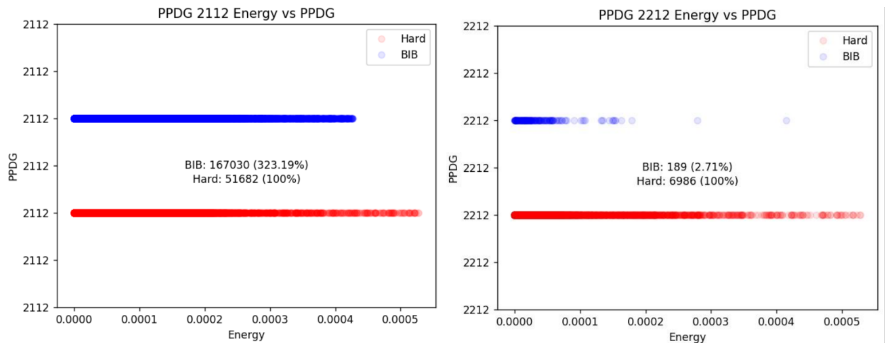


Figure 10: PPDG vs Energy comparison (1)

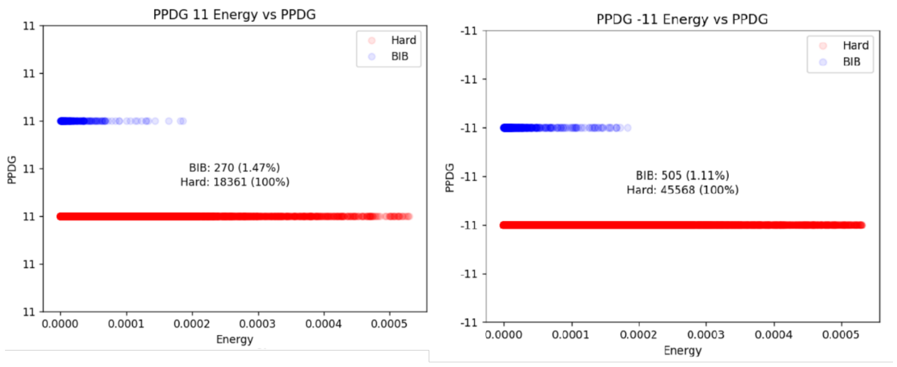


Figure 11: PPDG vs Energy comparison (2)

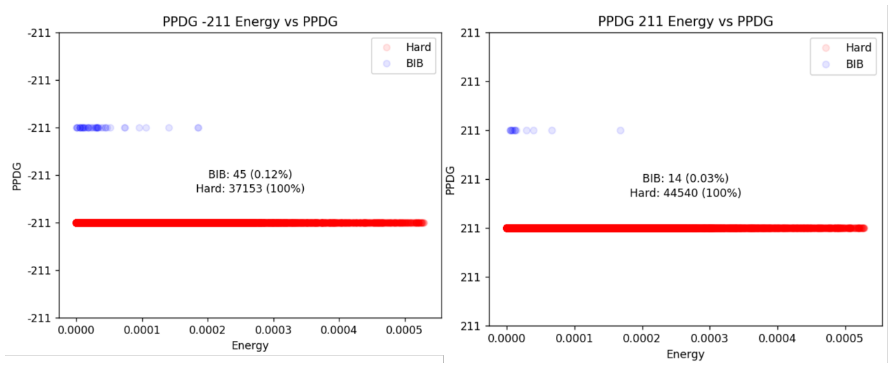


Figure 12: PPDG vs Energy comparison (3)

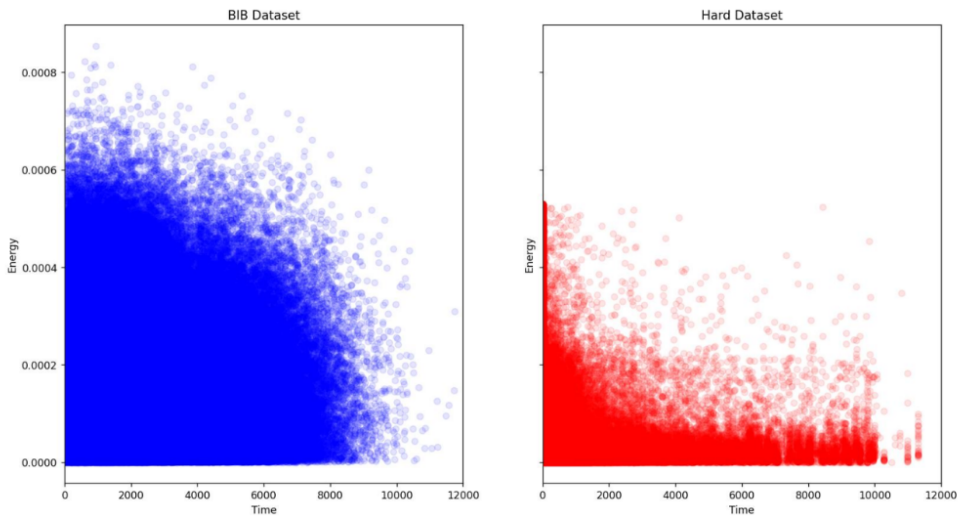
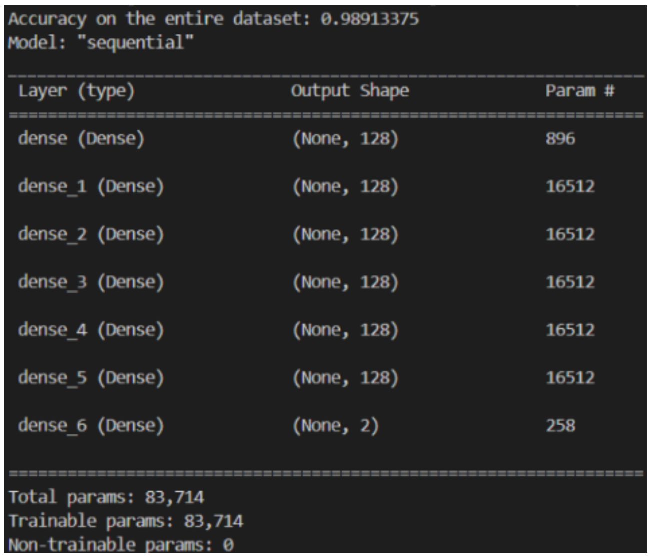


Figure 13: Energy vs Arrival time of Synthetic BIB dataset vs Hard Dataset

4.4 Neural Network Architecture

Once the datasets were seen to have enough distinguishing characteristics, an optimal neural network architecture was to be arrived at. This was done by using Grid-Search technique where neural networks were iteratively trained over a search space of varying number of layers, number of nodes in each layer and the type of activation function used.

The grid search technique yielded an optimum neural network configuration as follows, this Neural Network was then to be synthesize into an RTL project using hls4ml:



```
Accuracy on the entire dataset: 0.98913375
Model: "sequential"
_________________________________________________________________
 Layer (type)                Output Shape              Param #
=================================================================
 dense (Dense)               (None, 128)               896

 dense_1 (Dense)             (None, 128)               16512

 dense_2 (Dense)             (None, 128)               16512

 dense_3 (Dense)             (None, 128)               16512

 dense_4 (Dense)             (None, 128)               16512

 dense_5 (Dense)             (None, 128)               16512

 dense_6 (Dense)             (None, 2)                 258

=================================================================
Total params: 83,714
Trainable params: 83,714
Non-trainable params: 0
```

Figure 14: Parameters of the best NN obtained from GridSearch.

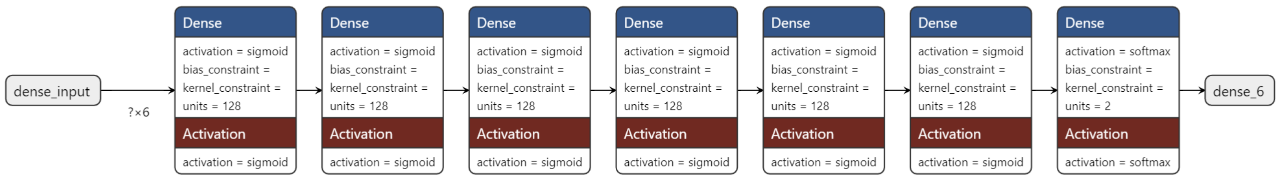


Figure 15: Best NN architecture obtained from GridSearch.

4.5 RTL Synthesis

hls4ml was used to import the TensorFlow model, quantize it and then generate an HLS project for it. The corresponding code is given in Appendix. Once the HLS project was successfully created, RTL synthesis was initiated using Vivado backend and the project was exported into Vivado. Synthesis and Implementation of the RTL was done on Vivado and it's power and timing reports were obtained as follows for a 10ns clock.

5 Flowchart: GEANT4 Simulation to Vivado Project

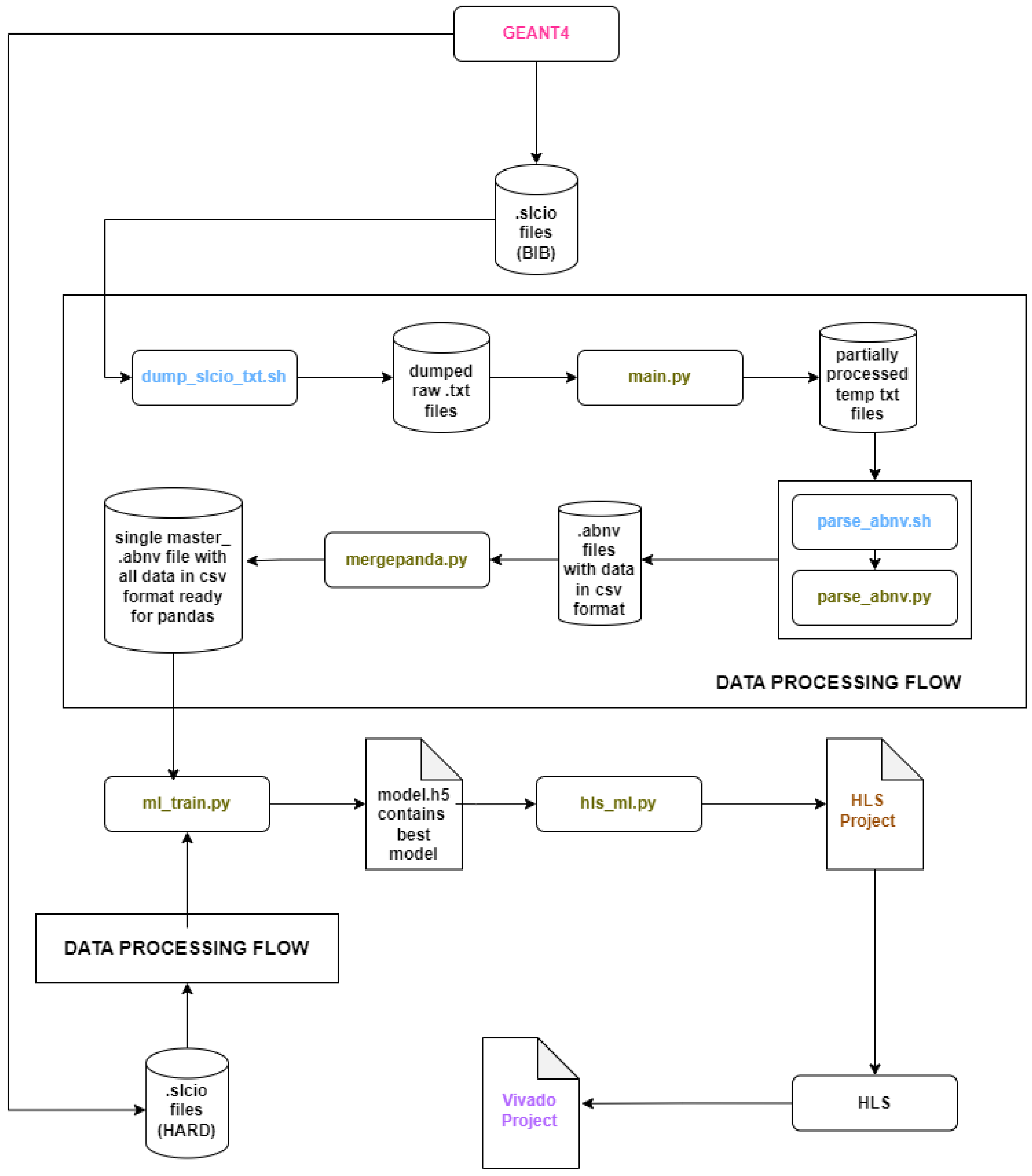
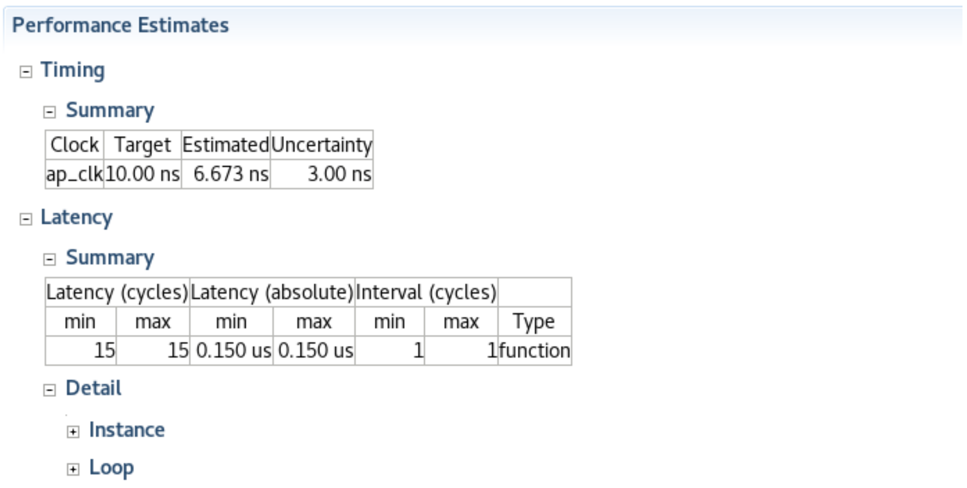


Figure 16: Flowchart showing process flow from GEANT4 simulation to Vivado project. Yellow: Python files, Blue: bash scripts. More details on the code functionality are presented in Appendix.

6 Results and Discussion

6.1 HLS Code

6.1.1 Timing


Performance Estimates

⊟ Timing

⊟ Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	6.673 ns	3.00 ns

⊟ Latency

⊟ Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
15	15	0.150 us	0.150 us	1	1	function

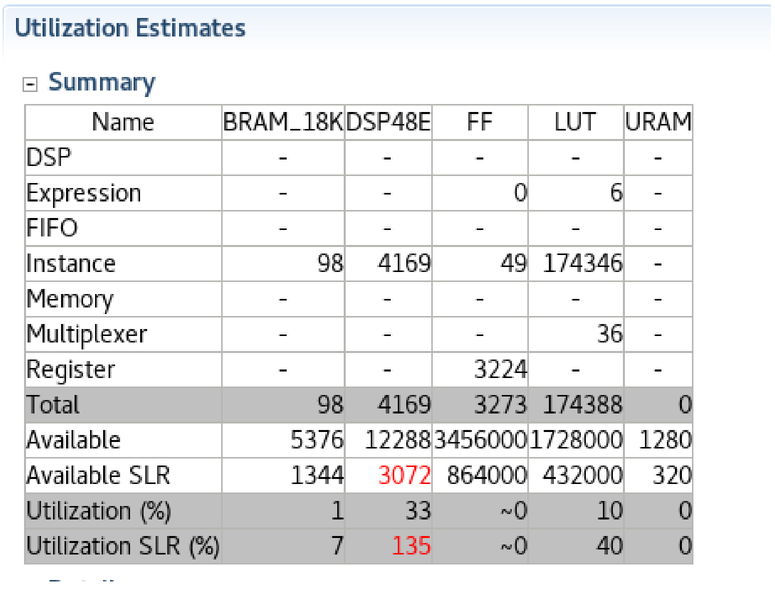
⊟ Detail

⊞ Instance

⊞ Loop

Figure 17: HLS timing summary.

6.1.2 Utilization

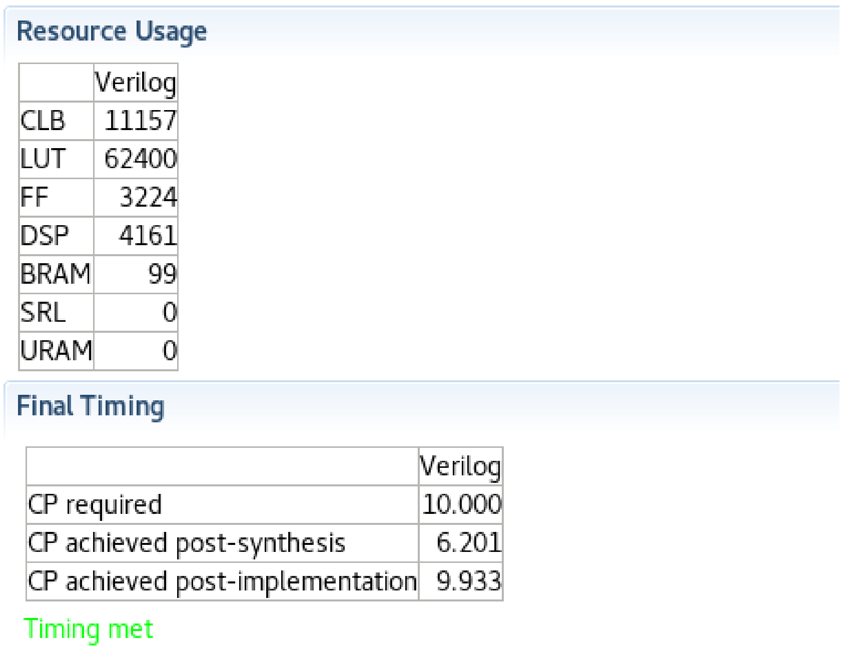

Utilization Estimates

⊟ Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	6	-
FIFO	-	-	-	-	-
Instance	98	4169	49	174346	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	36	-
Register	-	-	3224	-	-
Total	98	4169	3273	174388	0
Available	5376	12288	3456000	1728000	1280
Available SLR	1344	3072	864000	432000	320
Utilization (%)	1	33	~0	10	0
Utilization SLR (%)	7	135	~0	40	0

Figure 18: HLS utilization summary.

6.1.3 Timing and Utilization of Converted Verilog code



Resource Usage

	Verilog
CLB	11157
LUT	62400
FF	3224
DSP	4161
BRAM	99
SRL	0
URAM	0

Final Timing

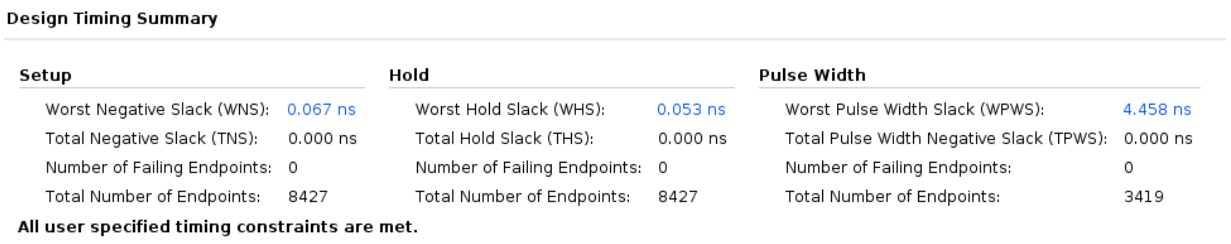
	Verilog
CP required	10.000
CP achieved post-synthesis	6.201
CP achieved post-implementation	9.933

Timing met

Figure 19: Timing summary and utilization estimate of converted Verilog code in HLS using Vivado backend showing target clock of 10ns is achieved. 30% clock uncertainty was included.

6.2 Vivado Project on part xcu250-figd2104-2L-e

6.2.1 Timing



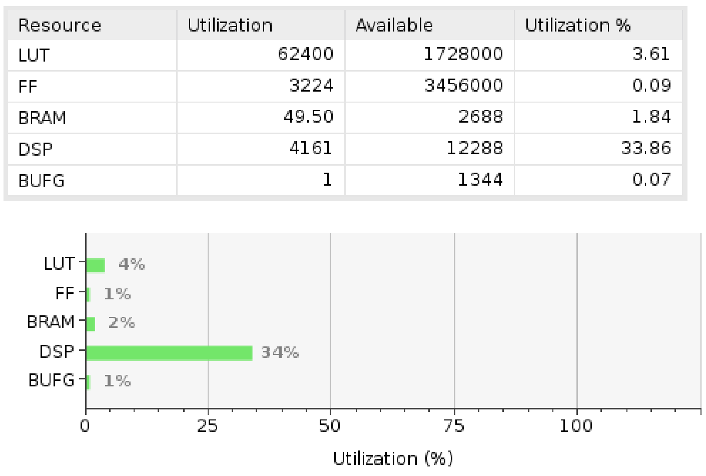
Design Timing Summary

Setup		Hold		Pulse Width	
Worst Negative Slack (WNS):	0.067 ns	Worst Hold Slack (WHS):	0.053 ns	Worst Pulse Width Slack (WPWS):	4.458 ns
Total Negative Slack (TNS):	0.000 ns	Total Hold Slack (THS):	0.000 ns	Total Pulse Width Negative Slack (TPWS):	0.000 ns
Number of Failing Endpoints:	0	Number of Failing Endpoints:	0	Number of Failing Endpoints:	0
Total Number of Endpoints:	8427	Total Number of Endpoints:	8427	Total Number of Endpoints:	3419

All user specified timing constraints are met.

Figure 20: Detailed timing summary in Vivado after synthesis. Target clock was set at 10ns.

6.2.2 Utilization



Resource	Utilization	Available	Utilization %
LUT	62400	1728000	3.61
FF	3224	3456000	0.09
BRAM	49.50	2688	1.84
DSP	4161	12288	33.86
BUFG	1	1344	0.07

Figure 21: Utilization report in Vivado Post synthesis.

6.2.3 Synthesized design

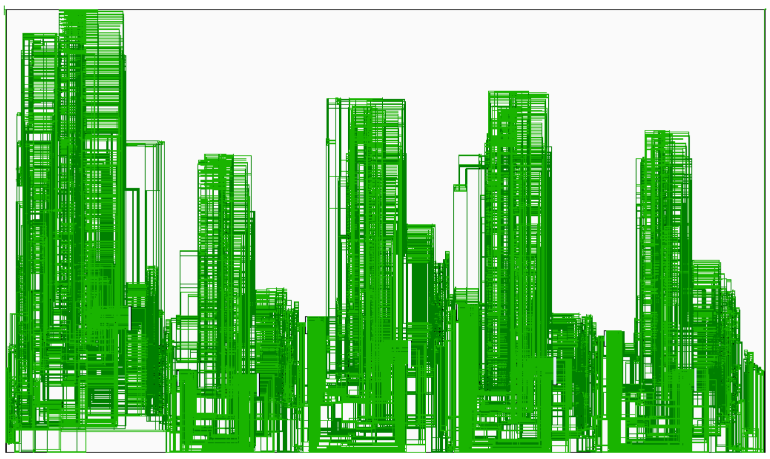


Figure 22: Synthesized design in Vivado

6.2.4 Implemented design

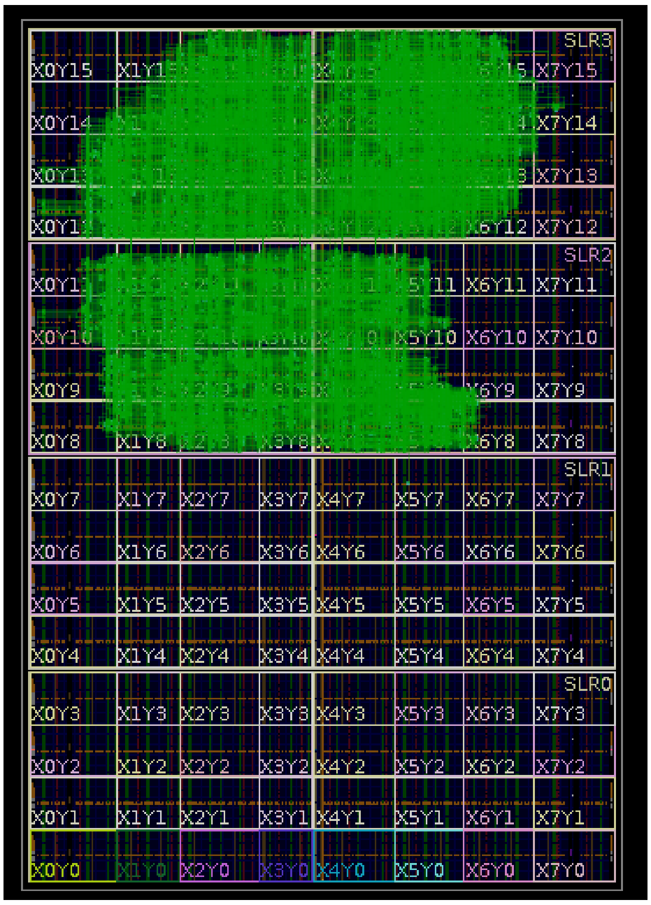
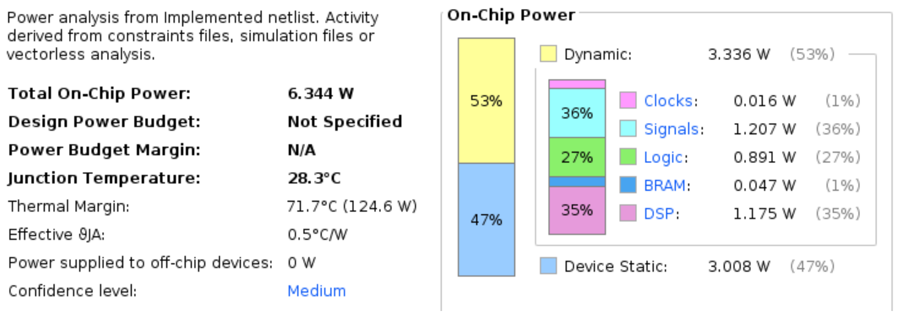


Figure 23: Implemented Design in Vivado

6.2.5 Power



Power analysis from Implemented netlist. Activity derived from constraints files, simulation files or vectorless analysis.

Total On-Chip Power:	**6.344 W**
Design Power Budget:	**Not Specified**
Power Budget Margin:	**N/A**
Junction Temperature:	**28.3°C**
Thermal Margin:	71.7°C (124.6 W)
Effective ϑJA:	0.5°C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

On-Chip Power

Dynamic:		3.336 W	(53%)
	Clocks:	0.016 W	(1%)
	Signals:	1.207 W	(36%)
	Logic:	0.891 W	(27%)
	BRAM:	0.047 W	(1%)
	DSP:	1.175 W	(35%)
Device Static:		3.008 W	(47%)

Figure 24: Power report in Vivado. Note that the switching file was not included, hence, confidence level of this power report is Medium and not High.

6.3 Key Findings

1. Training deep learning models and their hardware implementation via hls4ml, starting with GEANT4 simulation data was demonstrated.
2. It was seen that no trivial relationships exist between the Energy, arrival time and position data between the BIB and Hard data for the ECalBarrelCollection. A model trained directly on this data was shown to not work.
3. It was seen that there is a clear difference in the datasets in terms of particle composition although this data is not available in real-time, it goes to show that the BIB and Hard datasets are not identical.
4. In the case that there are distinguishing characteristics between the datasets, as in the case with synthetic BIB vs Hard, it was shown that the trained NNs could predict the data with a very high degree of accuracy and with <10ns latency for inference.

7 Conclusion

This project demonstrates the implementation of deep learning models on FPGA and the specific methodology to follow when dealing with GEANT4 simulation files. Data was successfully processed from simulation files and models were successfully trained and synthesised in HLS and Vivado, corresponding timing and utilization reports were also obtained. However a synthetic BIB dataset had to be used due to the lack of enough distinguishing features between the original Hard and BIB datasets.

8 Future Work

1. Bring out the distinguishing characteristics between Hard and BIB dataset using some mathematical techniques.
2. Test the other 12 collections to see if there are distinguishing features between Hard and BIB.
3. Inspect the integrity of simulation data.
4. Try this methodology with other GEANT4 simulations.
5. Develop a standard python package to convert or parse .slcio files into .csv format and integrate it into the LCIO python package.
6. Restructure the data to be compatible for other deep-learning architectures such as CNNs, RNNs etc. and implement those.

References

[1] C. Accettura and et al., “Towards a muon collider,” *arXiv preprint arXiv:2303.08533*, pp. 1–119, 2023. [Online]. Available: https://doi.org/10.48550/arxiv.2303.08533

[2] F. Fahim and et al., “hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices,” *arXiv preprint arXiv:2103.05579*, pp. 1–10, 2021. [Online]. Available: https://arxiv.org/abs/2103.05579

[3] O. Calin, *Deep Learning Architectures: A Mathematical Approach*, ser. Springer Series in the Data Sciences. Springer Nature Switzerland AG, 2020.

[4] M. f. t. C. C. Lorusso, "Implementing machine learning inference on fpgas from software to hardware using hls4ml," The Compact Muon Solenoid Experiment, CMS CERN, CH-1211 GENEVA 23, Switzerland, Conference Report CMS CR-2023/019, February 2023, v2, 22 March 2023.

[5] T. Aarrestad and et al., "Fast convolutional neural networks on fpgas with hls4ml," *Machine Learning: Science and Technology*, vol. 2, no. 4, p. 045015, 2021.

[6] F. Barbosa, L. Belfore, N. Branson, C. Dickover, C. Fanelli, D. Furletov, S. Furletov, L. Jokhovets, D. Lawrence, and D. Romanov, "Development of ml fpga filter for particle identification and tracking in real time," *IEEE Transactions on Nuclear Science*, vol. 70, no. 6, June 2023.

[7] A. Sznajder, "Nanosecond jet classification at lhc," *Journal of Physics: Conference Series*, vol. 2438, no. 1, p. 012047, 2023.

[8] B. Ramhorst, G. A. Constantinides, and V. Loncar, "Fpga resource-aware structured pruning for real-time neural networks," *arXiv preprint arXiv:2308.05170*, August 2023.

9 Appendix A

9.1 Usage Instructions for Code

9.1.1 Parsing the data into a pandas dataframe

First, all .slcio files for **either** BIB or Hard have to be put in one directory. Ensure **anajob** command is working, if not, from your current directory, **source setup.sh** from **LCIO/build**. The following code should be executed in this order and same to be repeated for the other dataset (refer flowchart in section 5):

1. **dump_slcio_txt.sh** should be run in the directory containing all .slcio files. This will dump all .slcio files into .txt format. For every .slcio file there will be a corresponding .txt file assuming each file has only one event.

2. **main.py** parses the raw text data to some extent by only grabbing the useful lines of the .txt files and saving them in new temporary text files. It has command line arguments that are useful if you want to plot figures like Fig.7 and Fig. 8.

3. **parse_abnv.py** removes unwanted characters and spaces in the temporary .txt files and loads each temporary text file information into a pandas dataframe and then creates a new .abnv file where the data is saved in csv format. Following this the temporary file is deleted

4. **parse_abnv.sh** is a bash script that runs **parse_abnv.py** over all .txt files one by one in the current directory.

5. **mergepanda.py** merges all .abnv files into a single master .abnv file that will contain the BIB or Hard dataset depending on what .slcio files you were working with.

9.1.2 Training models, plotting and hls4ml usage

1. **ml_train.py** loads both master .abnv files for BIB and hard into a single dataframe and performs the following functions: Plotting figures such as Fig.9 -13, generating the synthetic BIB dataset, using tensorflow to train models and finding the best model ands aving it to a file.

2. **hls_ml.py** loads the saved model file and initiates code conversion to an HLS project and from there we can use Vivado or HLS to proceed.

9.2 Converting .slcio into .txt files

Listing 1: Bash script for converting .slcio files into .txt `dump_lcio_txt.sh`

```
#!/bin/bash

# Counter for the output file names
count=0

# Loop through all .slcio files in the current directory
for file in *.slcio; do
    # Check if the file exists
    if [ -e "$file" ]; then
        # Run the command for each file and redirect the output to a text file
        dumpevent "$file" 1 > "eventdump$count.txt"
        # Increment the counter
        ((count++))
    fi
done
```

9.3 Parsing .txt files

Listing 2: This code can parse the.slcio files and also has some functions for visualization for creating plots. `main.py`

```
import re
import numpy as np
import pandas as pd
import plotly.graph_objs as go
import plotly.graph_objects as go
import numpy as np
import re
import pandas as pd

def process_file(input_file, output_file):
    def process_line(line):
        line = re.sub(r'[^\w\s.,+-]', ',', line)
        line = '␣'.join(line.split())
        return line

    def remove_extra_commas(lines):
        cleaned_lines = []
        for line in lines:
            if line[:2] == '-,':
                line = line[2:]
            parts = line.split(',')
            cleaned_parts = [part.strip() for part in parts if part.strip()]
            cleaned_line = ','.join(cleaned_parts)
            cleaned_lines.append(cleaned_line)
        return cleaned_lines

    with open(input_file, 'r') as f:
        lines = f.readlines()

    processed_lines = [process_line(line) for line in lines]
    cleaned_lines = remove_extra_commas(processed_lines)

    with open(output_file, 'w') as f:
```

```
    for line in cleaned_lines:
        f.write(line + '\n')

# Print number of lines in final_result.txt
with open(output_file, 'r') as f:
    lines = f.readlines()
    print(len(lines))

# Open final_result.txt and if any line ends with +n, where n greater than 3, calculate sum of
    ↪ all such n = count. Then print % = count / (sum of all n including those less than 3)
with open(output_file, 'r') as f:
    lines = f.readlines()

sum_n = 0
count = 0
for line in lines:
    match = re.search(r'\+(\d+)$', line.strip())  # Check if the line ends with '+n'
    if match:
        n = int(match.group(1))
        if n > 3:  # Ensure n is greater than 3
            sum_n += n
            count += 1

total_sum = sum_n
for line in lines:
    match = re.search(r'\+(\d+)$', line.strip())  # Check if the line ends with '+n'
    if match:
        n = int(match.group(1))
        total_sum += n


#avoid division by zero
if total_sum == 0:
    print("No␣points␣excluded.")
else:
    percentage = (count / total_sum) * 100
    print(f"{percentage:.2f}%␣of␣points(+4␣and␣above␣decays)␣excluded␣due␣to␣difficulties␣in␣
        ↪ parsing.")

with open(output_file, 'r') as f:
    lines = f.readlines()

with open(output_file, 'w') as f:
    skip_next = 0
    for line in lines:
        if skip_next > 0:
            skip_next -= 1
            continue
        match = re.search(r'\+(\d+)$', line.strip())  # Check if the line ends with '+n'
        if match:
            n = int(match.group(1))
            if n > 3:  # Ensure n is greater than 2
                skip_next = n
                continue
        f.write(line)

# Open final_result.txt and if any line ends with +2, duplicate that line below it
with open(output_file, 'r') as f:
    lines = f.readlines()
```

```
with open(output_file, 'w') as f:
    for line in lines:
        f.write(line)
        if line.strip().endswith('+2'):
            f.write(line)

# Print number of lines in final_result.txt
with open(output_file, 'r') as f:
    lines = f.readlines()
    print(len(lines))

# Open final_result.txt, if any line repeats more than once, swap its position with the line
    ↪ below it
def swap_repeated_lines(file_path):
    with open(file_path, 'r') as file:
        lines = file.readlines()

    prev_line = None
    for i in range(len(lines)):
        if prev_line == lines[i]:
            lines[i], lines[i + 1] = lines[i + 1], lines[i]
            prev_line = None
        else:
            prev_line = lines[i]

    with open(file_path, 'w') as file:
        file.writelines(lines)

# Usage example
file_path = output_file
swap_repeated_lines(file_path)

# Print number of lines in final_result.txt
with open(output_file, 'r') as f:
    lines = f.readlines()
    print(len(lines))

# Duplicate lines ending with +3 two times below them
with open(output_file, 'r') as f:
    lines = f.readlines()

with open(output_file + "_modified.txt", 'w') as f:
    for line in lines:
        f.write(line)
        if line.strip().endswith('+3'):
            f.write(line)
            f.write(line)

# Swap lines i+1 and i+4 if line i ends with +3 and start scanning from i+6
with open(output_file + "_modified.txt", 'r') as f:
    lines = f.readlines()

with open(output_file + "_final.txt", 'w') as f:
    skip_next = 0
    for i in range(len(lines)):
        if skip_next > 0:
            skip_next -= 1
            continue
            f.write(lines[i])
            if lines[i].strip().endswith('+3'):
                if i + 4 < len(lines):  # Make sure there are enough lines to swap
                    lines[i + 1], lines[i + 4] = lines[i + 4], lines[i + 1]
                    skip_next = 5
                else:
                    f.write("Error:␣Not␣enough␣lines␣to␣perform␣swap.\n")

    with open(output_file, "r") as file:
        lines = file.readlines()

    combined_lines = []
    for i in range(0, len(lines), 2):
        combined_lines.append(lines[i].strip() + "," + lines[i + 1].strip())

    with open(output_file, "w") as file:
        file.write("\n".join(combined_lines))

    # Print number of lines in final_result.txt
    with open(output_file, 'r') as f:
        lines = f.readlines()
        print(len(lines))

    # Open final_result.txt and load into a pandas dataframe
    df = pd.read_csv(output_file, header=None)

    # Name the columns: id, cellId0, cellId1, energy, position (x,y,z), nMCParticles MC contribution
        ↪ : prim. PDG, energy_part, time, length, sec. PDG and stepPosition (x,y,z)
    df.columns = ['id', 'cellId0', 'cellId1', 'energy', 'position␣x', 'position␣y', 'position␣z', '
        ↪ nMCParticles',
                  'mc␣contri␣prim.␣PDG', 'energy_part', 'time', 'length', 'sec_PDG', 'stepPosition␣x
                      ↪ ',
                  'stepPosition␣y', 'stepPosition␣z']

    # Convert all the columns to the numeric type
    df = df.apply(pd.to_numeric, errors='coerce')

    return df

# Example usage:
# Assuming 'data.txt' is the input file and 'final_result.txt' is the output file


# Create histogram trace
def plot_histogram(data, x_label='Energy', y_label='Count␣(log␣scale)', nbins=20, color='royalblue',
    ↪  title='Energy␣Distribution', plot_bgcolor='rgb(51,␣51,␣51)', paper_bgcolor='rgb(51,␣51,␣51)'
    ↪ , font_color='white'):
    # Create histogram trace
    histogram = go.Histogram(
        x=data,
        nbinsx=nbins,
        marker=dict(color=color)
    )

    # Create layout
    layout = go.Layout(
        title=title,
        xaxis=dict(title=x_label, color=font_color),
        yaxis=dict(title=y_label, type='log', color=font_color),
        bargap=0.05,
        bargroupgap=0.1,
        plot_bgcolor=plot_bgcolor,
        paper_bgcolor=paper_bgcolor,
        font=dict(color=font_color),
        autosize=True,
    )

    # Create figure
    fig = go.Figure(data=[histogram], layout=layout)

    # Update color for dark theme
    fig.update_layout(
        xaxis=dict(linecolor=font_color),
        yaxis=dict(linecolor=font_color),
        bargap=0.05,
        bargroupgap=0.1,
        plot_bgcolor=plot_bgcolor,
        paper_bgcolor=paper_bgcolor,
        font=dict(color=font_color)
    )

    # Show plot
    fig.show()

# Example usage:
# Assuming df['energy'] contains your energy data
#plot_histogram(df['energy'])


import plotly.graph_objects as go
import numpy as np

def plot_unrolled_and_reconstructed(df, column_name, angle):
    # Extract data
    x = df['position␣x']
    y = df['position␣y']
    z = df['position␣z']
    c = df['time']

    # Convert cylindrical coordinates to rectangular coordinates

    # print points with lowest z coordinate value
    min_z = df[df['position␣z'] == df['position␣z'].min()]
    O = [0, 0, min_z['position␣z'].values[0]]
    A = [min_z['position␣x'].values[0], min_z['position␣y'].values[0], min_z['position␣z'].values
        ↪ [0]]

    # find angle between OA and every other point, check cross product of OA and OB, if positive
        ↪ angle = angle otherwise angle = -angle
    def angle_between_points(O, A, B):
        OA = np.array(A) - np.array(O)
        OB = np.array(B) - np.array(O)
        angle = np.arccos(np.dot(OA, OB) / (np.linalg.norm(OA) * np.linalg.norm(OB)))
        cross_product = np.cross(OA, OB)
        if cross_product[2] > 0:
            return angle
        else:
            return -angle
```

```
angles = []
zvalues = []
for i in range(len(df)):
    B = [df['position␣x'].iloc[i], df['position␣y'].iloc[i], -2203.0]
    if (B == O):
        continue
    angles.append(angle_between_points(O, A, B))
    zvalues.append(df['position␣z'].iloc[i])

# For the z-coordinate, it remains the same as the z of the cylinder
z_rect = z

# Create scatter plot trace for unrolled plot
scatter_unrolled = go.Scatter(
    x=angles,
    y=z_rect,
    mode='markers',
    marker=dict(
        size=8,
        color=c,
        colorscale='Rainbow',
        colorbar=dict(title='Time'),
        opacity=0.8
    )
)

# Create layout for unrolled plot
layout_unrolled = go.Layout(
    title='ECalBarrelCollection␣Position␣vs␣Time␣(Event␣ID:{angle}/999)␣UNROLLED',
    xaxis=dict(title='Theta␣(degrees)'),
    yaxis=dict(title='Z'),
    plot_bgcolor='black',
    paper_bgcolor='black',
    font=dict(color='white')
)

# Create figure for unrolled plot
fig_unrolled = go.Figure(data=[scatter_unrolled], layout=layout_unrolled)

# Save unrolled plot as PNG with resolution set to 1080p
file_name = f"images/unrolled/plot_{column_name}_{angle}.png"
fig_unrolled.write_image(file_name, width=1920, height=1080)


# Calculate radius
r = np.sqrt(df['position␣x'] ** 2 + df['position␣y'] ** 2)

# Create scatter plot trace for reconstructed plot
scatter_reconstructed = go.Scatter3d(
    x=np.cos(angles) * r[0],
    y=np.sin(angles) * r[0],
    z=df['position␣z'],
    mode='markers',
    marker=dict(
        size=5,
        color=df['time'],
        colorscale='Rainbow',
        colorbar=dict(title='Time'),
            line=dict(color='white', width=0),
            opacity=0.8
        )
    )

    # Create layout for reconstructed plot
    layout_reconstructed = go.Layout(
        title='ECalBarrelCollection␣Position␣vs␣Time␣(Event␣ID:000/999)␣RECONSTRUCTED',
        scene=dict(
            xaxis=dict(showbackground=False, showline=True, linecolor='white', title='X', showgrid=
                ↪ False,
                        showticklabels=True),
            yaxis=dict(showbackground=False, showline=True, linecolor='white', title='Y', showgrid=
                ↪ False,
                        showticklabels=True),
            zaxis=dict(showbackground=False, showline=True, linecolor='white', title='Z', showgrid=
                ↪ False,
                        showticklabels=True),
            bgcolor='black'
        ),
        margin=dict(l=0, r=0, b=0, t=40),
        paper_bgcolor='black',
        font=dict(color='white')
    )

    # Create figure for reconstructed plot
    #fig_reconstructed = go.Figure(data=[scatter_reconstructed], layout=layout_reconstructed)

    # Show reconstructed plot
    #fig_reconstructed.show()
    # Create figure for reconstructed plot
    #fig_reconstructed = go.Figure(data=[scatter_reconstructed], layout=layout_reconstructed)
    #file_name = f"plot_ree_{column_name}_{angle}.png"
    #fig_reconstructed.write_image(file_name)
    # Show reconstructed plot
    #fig_reconstructed.show()

# Example usage
# Assuming df is your DataFrame
#plot_unrolled_and_reconstructed(df)

import plotly.graph_objects as go
import numpy as np
import os

def plot_3d_heatmap_at_angle(df, column_name, angle, save_image=False):
    # Generate data for cylinder surface
    def cylinder(x, y, z, radius, height):
        u = np.linspace(0, 2 * np.pi, 100)
        v = np.linspace(0, height, 10)
        U, V = np.meshgrid(u, v)
        X = x + (radius * np.cos(U))
        Y = y + (radius * np.sin(U))
        Z = z + V
        return X, Y, Z

    # Create scatter plot trace
    scatter = go.Scatter3d(
        x=df['position_x'],
        y=df['position␣y'],
        z=df['position␣z'],
        mode='markers',
        marker=dict(
            size=5,
            color=df[column_name],
            colorscale='Rainbow',
            colorbar=dict(title=column_name),
            line=dict(color='white', width=0),
            opacity=0.8
        )
    )

    # Generate cylinder surface
    cylinder_x, cylinder_y, cylinder_z = cylinder(x=0, y=0, z=-2000, radius=1500, height=4000)

    # Create cylindrical outline trace
    dark_color = 'rgb(0,␣0,␣0)'
    light_color = 'rgb(255,␣255,␣255)'
    light_color2='rgb(45,␣45,␣45)'

    colorscale = [
        [0, light_color],
        [0.5, dark_color],
        [1, light_color]
    ]

    cylinder_outline = go.Surface(
        x=cylinder_x,
        y=cylinder_y,
        z=cylinder_z,
        opacity=0.1,
        showscale=False,
        colorscale=colorscale
    )

    n_points = 50
    theta = np.linspace(0, 2*np.pi, n_points)
    x = 1500 * np.cos(theta)
    y = 1500 * np.sin(theta)
    z = np.linspace(-2000, 2000, n_points)

    lines_circumference = []
    for i in range(n_points):
        lines_circumference.append(go.Scatter3d(x=[x[i], x[i]], y=[y[i], y[i]], z=[-2000, 2000],
            ↪ mode='lines', line=dict(color=light_color2)))

    # Create layout
    layout = go.Layout(
        title=f'ECalBarrelCollection␣Position␣vs␣{column_name}␣(Event␣ID:000/999)␣ORIGINAL',
        scene=dict(
            xaxis=dict(showbackground=False, showline=True, linecolor='white', title='X', showgrid=
                ↪ False, showticklabels=True),
            yaxis=dict(showbackground=False, showline=True, linecolor='white', title='Y', showgrid=
                ↪ False, showticklabels=True),
            zaxis=dict(showbackground=False, showline=True, linecolor='white', title='Z', showgrid=
                ↪ False, showticklabels=True),
            bgcolor='black',
            camera=dict(
                up=dict(x=0, y=1, z=0),  # Adjusted for y-axis rotation
                center=dict(x=0, y=0, z=0),
                eye=dict(x=0.5, y=0.5, z=1.5)  # Zoom out by increasing z value
            )
        ),
        margin=dict(l=0, r=0, b=0, t=40),
        paper_bgcolor='black',
        font=dict(color='white')
    )

    # Create figure
    fig = go.Figure(layout=layout)

    # Update camera viewpoint for rotation and zoom
    eye_x = np.cos(np.deg2rad(angle))
    eye_z = np.sin(np.deg2rad(angle))
    fig.update_layout(scene_camera=dict(eye=dict(x=0.5*eye_x, y=0.5, z=1.5)))  # Adjusted for y-axis
        ↪ rotation and zoom

    # Append scatter plot, cylinder, and lines to figure
    fig.add_trace(scatter)
    fig.add_trace(cylinder_outline)
    fig.add_traces(lines_circumference)
    fig.update_traces(showlegend=False)

    # Save plot as image if save_image is True
    if save_image:
        if not os.path.exists("images"):
            os.makedirs("images")
        #fig.write_image(f"images/plot_{column_name}_{angle}.png", width=1200, height=800, scale=3)

    # Show plot
    #fig.show()

# Example usage:
# Assuming df is your DataFrame and 'time' is the column you want to use for heatmap

import argparse
def main():
    parser = argparse.ArgumentParser(description='Process␣a␣file␣and␣plot␣3D␣heatmap␣at␣a␣specified␣
        ↪ angle.')
    parser.add_argument('input_file', help='Path␣to␣the␣input␣text␣file')
    parser.add_argument('output_file', help='Path␣to␣the␣output␣text␣file')
    parser.add_argument('column_name', help='Name␣of␣the␣column␣to␣be␣plotted')
    parser.add_argument('--angle', type=float, default=0, help='Angle␣for␣the␣3D␣plot␣(default:␣45)'
        ↪ )
    parser.add_argument('--save_image', action='store_true', help='Save␣the␣plot␣as␣an␣image')

    args = parser.parse_args()

    # Process file
    df = process_file(args.input_file, args.output_file)
    plot_unrolled_and_reconstructed(df,args.column_name, args.angle)
    # Plot 3D heatmap
    #plot_3d_heatmap_at_angle(df, args.column_name, args.angle, save_image=args.save_image)
    #os.remove(args.output_file)

if __name__ == "__main__":
    main()
```

Listing 3: Code for converting partially processed .txt files into .abnv intermediary files for further formatting and parsing `parse_abnv.py`

```
import argparse
import pandas as pd
import os

def find_line_number(file_path, target_text):
    with open(file_path, 'r') as file:
        lines = file.readlines()
        for i, line in enumerate(lines):
            if target_text in line:
                return i + 1  # Line numbers start from 1
    return None

def clean_temp_file(temp_file):
    with open(temp_file, 'r') as file:
        content = file.read()

    # Remove unwanted spaces and characters from each line
    content = content.replace('->', '').replace('(', '').replace(')', '').replace('|', ',')

    # Remove leading and trailing spaces from each line
    content = '\n'.join(line.strip() for line in content.split('\n'))

    with open(temp_file, 'w') as file:
        file.write(content)

def copy_lines(source_file, destination_file, start_line, end_line, heading):
    with open(source_file, 'r') as source, open(destination_file, 'a') as dest:
        dest.write(heading + '\n')
        lines = source.readlines()[start_line - 1:end_line - 1]
        for line in lines:
            if line.strip().startswith('->'):
                dest.write(line)

def process_file(filename):
    # Remove spaces from all lines and save the file
    with open(filename, 'r') as file:
        lines = file.readlines()
    lines = [line.replace('␣', '') for line in lines]
    with open(filename, 'w') as file:
        file.writelines(lines)

    # Open the file again to perform further operations
    with open(filename, 'r') as file:
        lines = file.readlines()

    # Find line numbers for different collections
    a = lines.index(next(line for line in lines if 'ECalBarrelCollection' in line))
    b = lines.index(next(line for line in lines if 'ECalEndcapCollection' in line))
    c = lines.index(next(line for line in lines if 'HCalBarrelCollection' in line))
    d = lines.index(next(line for line in lines if 'HCalEndcapCollection' in line))

    # Initialize a pandas dataframe
    df = pd.DataFrame(columns=['CollectionID', 'PPDG', 'Energy', 'Time', 'Length', 'SPDG', 'X', 'Y',
        ↪  'Z'])

    # Read CSV values and append to the dataframe
    data_frames = []
    for start, end, prefix in [(a+3, b-1, 'ECBC'), (c+3, d-1, 'ECEC'), (d+3, len(lines)-1, 'HCBC'),
        ↪ (c+3, d-1, 'HCEC')]:
        data = []
        for line in lines[start:end]:
            values = line.strip().split(',')
            data.append([prefix] + values)
        temp_df = pd.DataFrame(data, columns=df.columns)
        data_frames.append(temp_df)
    df = pd.concat(data_frames, ignore_index=True)

    # Print dataframe information
    print(df.info())
    # Convert dataframe columns to numeric
    df = df.apply(pd.to_numeric, errors='ignore')
    return df



def main():
    parser = argparse.ArgumentParser(description='Process␣input␣file␣and␣generate␣output␣in␣.abnv␣
        ↪ format.')
    parser.add_argument('input_file', metavar='input_file', type=str, help='Path␣to␣input␣file')
    args = parser.parse_args()

    input_file = args.input_file
    temp_file = 'temp.txt'

    ecal_barrel_start = find_line_number(input_file, 'ECalBarrelCollection')
    ecal_endcap_start = find_line_number(input_file, 'ECalEndcapCollection')
    hcal_barrel_start = find_line_number(input_file, 'HCalBarrelCollection')
    hcal_endcap_start = find_line_number(input_file, 'HCalEndcapCollection')
    hcal_ring_start = find_line_number(input_file, 'HCalRingCollection')

    copy_lines(input_file, temp_file, ecal_barrel_start, ecal_endcap_start, 'ECalBarrelCollection')
    copy_lines(input_file, temp_file, ecal_endcap_start, hcal_barrel_start, 'ECalEndcapCollection')
    copy_lines(input_file, temp_file, hcal_barrel_start, hcal_endcap_start, 'HCalBarrelCollection')
    copy_lines(input_file, temp_file, hcal_endcap_start, hcal_ring_start, 'HCalEndcapCollection')

    clean_temp_file(temp_file)  # Clean up the temp file after copying and before finishing

    filename = 'temp.txt'
    df = process_file(filename)
    print(df.head())

    # Print rows where PPDG is negative
    print(df[df['PPDG'] < 0])

    # Save the dataframe to a .abnv file in csv format
    output_file = os.path.splitext(input_file)[0] + '.abnv'
    df.to_csv(output_file, index=False)

    # Remove the temporary file
    os.remove(temp_file)

if __name__ == "__main__":
    main()
```

Listing 4: Bash script to execute parse_abnv.py for all dumped .txt files parse$_{a}bnv.sh$

```
#!/bin/bash

# Directory containing the text files
directory="."

# Loop through each text file in the directory
for file in "$directory"/*.txt; do
    # Check if the file exists
    if [ -e "$file" ]; then
        # Acquire a lock and execute the Python script
        {
            flock -x 200
            python3 parse_abnv.py "$file"
        } 200>"$file.lock"  # Use a separate lock file for each text file
    fi
done
```

Listing 5: Code to load data in the individual .abnv files into a single pandas dataframe and then save the dataframe as a master .abnv file in csv format for easy access `mergepanda.py`

```
import os
import pandas as pd

# Get the current directory
directory = os.getcwd()

# List all .abnv files in the directory
abnv_files = [file for file in os.listdir(directory) if file.endswith('.abnv')]

# Initialize an empty list to store dataframes
dfs = []

# Iterate over each .abnv file
for file in abnv_files:
    # Load the .abnv file into a pandas dataframe
    df = pd.read_csv(os.path.join(directory, file))
    # Append the dataframe to the list
    dfs.append(df)

# Concatenate all dataframes into a single dataframe
combined_df = pd.concat(dfs, ignore_index=True)

# Export the combined dataframe to master_hard.abnv
output_file = 'master_BIB.abnv'
combined_df.to_csv(output_file, index=False)

print(f"Combined␣dataframe␣exported␣to␣{output_file}")
```

9.4 Code to train the model

Listing 6: This code generates the synthetic database, finds the best NN model using tensorflow and saves it and also has plotting functions `ml_train.py`

```
import pandas as pd

# Load the first dataset
file_path1 = "master_hard.abnv"
```

```
df_hard = pd.read_csv(file_path1, header=0)

# Add a new column to indicate the source dataset
df_hard['Dataset'] = 'Hard'

# Load the second dataset
file_path2 = "master_BIB.abnv"
df_bib = pd.read_csv(file_path2, header=0)

# Add a new column to indicate the source dataset
df_bib['Dataset'] = 'BIB'

# Concatenate the two dataframes
combined_df = pd.concat([df_hard, df_bib], ignore_index=True)

# Print the combined dataframe
print(combined_df)
# Filter rows with CollectionID ECBC
df_ecbc = combined_df[combined_df['CollectionID'] == 'ECBC']

# Print the new dataframe
print(df_ecbc)
#count number of Hard and BIB datasets
print(df_ecbc['Dataset'].value_counts())

#make a new dataframe by selecting 500,000 BIB rows and 500,000 Hard rows at random
df_sample = df_ecbc.groupby('Dataset').apply(lambda x: x.sample(n=500000, random_state=42)).
    ↪ reset_index(drop=True)

# Print the new dataframe
print(df_sample)
#exclude outlier values of Energy column for BIB and Hard datasets using Z-Score Method
from scipy import stats

# Calculate the z-scores for the Energy column in the BIB and Hard datasets
df_sample['ZScore'] = df_sample.groupby('Dataset')['Energy'].transform(lambda x: stats.zscore(x))

# Filter out rows with z-scores greater than 3 or less than -3
df_sample = df_sample[(df_sample['ZScore'] < 2) & (df_sample['ZScore'] > -2)]

# Print the new dataframe
print(df_sample)

#plot Energy distribution for BIB and Hard datasets using a scatter plot for df_sample
import matplotlib.pyplot as plt

# Create a figure with two subplots

#plot the side by side graph of PPDG vs energy for BIB and Hard datasets for each PPDG
#exclude PPDGs that are not present in both datasets
import matplotlib.pyplot as plt

# Filter out PPDGs that are not present in both datasets
common_ppdgs = set(df_sample[df_sample['Dataset'] == 'Hard']['PPDG']).intersection(set(df_sample[
    ↪ df_sample['Dataset'] == 'BIB']['PPDG']))

# Plot the side-by-side graphs, instead of dots, use some type of heatmap to show BIB and Hard
    ↪ values vs PPDG
import seaborn as sns
```

```
# for ppdg in common_ppdgs:
#     fig, ax = plt.subplots()

#     # Filter data for the specific PPDG and dataset
#     data_hard = df_sample[(df_sample['PPDG'] == ppdg) & (df_sample['Dataset'] == 'Hard')]
#     data_bib = df_sample[(df_sample['PPDG'] == ppdg) & (df_sample['Dataset'] == 'BIB')]

#     # Combine data for violin plot
#     data_combined = pd.concat([data_hard, data_bib])

#     # Plot violin plot
#     sns.violinplot(x='Dataset', y='Energy', data=data_combined, ax=ax)

#     # Set title
#     plt.title(f'PPDG {ppdg} Energy Distribution by Dataset')

#     # Show plot
#     plt.show()


import matplotlib.pyplot as plt

# Set the transparency level
alpha = 0.1

# Define the offset
offset = 1  # Adjust this value based on your preference

# Iterate over common_ppdgs
import matplotlib.pyplot as plt

# Set the transparency level
alpha = 0.1

# Define the offset
offset = 0.1  # Adjust this value based on your preference

# Iterate over common_ppdgs
# for ppdg in common_ppdgs:
#     fig, ax = plt.subplots()

#     # Filter data for the specific PPDG and dataset
#     data_hard = df_sample[(df_sample['PPDG'] == ppdg) & (df_sample['Dataset'] == 'Hard')]
#     data_bib = df_sample[(df_sample['PPDG'] == ppdg) & (df_sample['Dataset'] == 'BIB')]

#     # Plot data points with transparency and offset
#     ax.plot(data_hard['Energy'], data_hard['PPDG'], 'o', color='red', alpha=alpha, label='Hard')
#     ax.plot(data_bib['Energy'], data_bib['PPDG'] + offset, 'o', color='blue', alpha=alpha, label='
    ↪ BIB')

#     # Set labels and title
#     ax.set_xlabel('Energy')
#     ax.set_ylabel('PPDG')
#     plt.title(f'PPDG {ppdg} Energy vs PPDG')
#     perc_bib = len(data_bib) / len(data_hard) * 100
#     #label number of bib and hard points used in the plot, in brackets, show percentage, keeping
    ↪ hard points as 100%
```

```
#     plt.text(0.5, 0.5, f'BIB: {len(data_bib)} ({perc_bib:.2f}%)', horizontalalignment='center',
    ↪ verticalalignment='center', transform=ax.transAxes)
#     plt.text(0.5, 0.45, f'Hard: {len(data_hard)} (100%)', horizontalalignment='center',
    ↪ verticalalignment='center', transform=ax.transAxes)

#    # Set y-axis limits
#     min_ppdg = min(data_hard['PPDG'].min(), data_bib['PPDG'].min() + offset)
#     max_ppdg = max(data_hard['PPDG'].max(), data_bib['PPDG'].max() + offset)
#     ax.set_ylim(min_ppdg - 0.1, max_ppdg + 0.1)  # Adjust the padding as needed

#     # Set y-axis tick labels
#     ax.set_yticks(ax.get_yticks())
#     ax.set_yticklabels([ppdg] * len(ax.get_yticks()))  # Set all y-axis tick labels to the same
    ↪ PPDG

#     # Add legend
#     plt.legend()

#     # Show plot
#     #plt.show()


#take 1000 random samples each from BIB and Hard rows of df_sample and make a new dataframe

# #train a ML model to classify BIB and Hard datasets based on PPDG,Energy ,    Time,  Length ,  X
    ↪    ,   Y   ,    Z columns
# from sklearn.model_selection import train_test_split
# from sklearn.ensemble import RandomForestClassifier
# from sklearn.metrics import accuracy_score

# # Define the features and target variable
# X = df_sample_subset[['PPDG', 'Energy', 'Time', 'Length', 'X', 'Y', 'Z']]
# y = df_sample_subset['Dataset']

# # Split the data into training and testing sets
# X_train, X_test, y_train, y_test = train_test_split(X, y, test_size=0.2, random_state=42)

# # Test all classifiers
# #import the classifiers
# from sklearn.ensemble import RandomForestClassifier
# from sklearn.svm import SVC
# from sklearn.linear_model import LogisticRegression
# from sklearn.neighbors import KNeighborsClassifier
# from sklearn.naive_bayes import GaussianNB
# from sklearn.tree import DecisionTreeClassifier
# #more
# from sklearn.metrics import accuracy_score

# # Initialize the classifiers
# rf = RandomForestClassifier(random_state=42)
# svc = SVC(random_state=42)
# lr = LogisticRegression(random_state=42)
# knn = KNeighborsClassifier()
# nb = GaussianNB()
# dt = DecisionTreeClassifier(random_state=42)

# from sklearn.metrics import f1_score
# from sklearn.metrics import confusion_matrix
# # Create a list of classifiers
# classifiers = [rf, svc, lr, knn, nb, dt]
# best_accuracy=0
# # Train and evaluate each classifier
# for clf in classifiers:
#     # Train the classifier
#     clf.fit(X_train, y_train)

#     # Make predictions
#     y_pred = clf.predict(X_test)

#     # Calculate the accuracy
#     accuracy = accuracy_score(y_test, y_pred)

#     # Print the classifier and accuracy
#     print(f'{clf.__class__.__name__} Accuracy: {accuracy:.2f}')
#     #save the best classifier
#     if accuracy > best_accuracy:
#         best_accuracy = accuracy
#         best_classifier = clf

#     #print confusion matrix for the best classifier
#     y_pred = best_classifier.predict(X_test)
#     cm = confusion_matrix(y_test, y_pred)
#     print(f'Confusion Matrix:\n{cm}')


import numpy as np
#train a Deep Learning model to classify BIB and Hard datasets based on PPDG,Energy ,    Time,
    ↪ Length ,   X    ,    Y   ,    Z columns
#use popular neural network architectures like CNN, RNN, LSTM, etc. Use tensorflow
#use dataframe df_sample_subset


#generate a BIB dataframe with 500,000 rows that has random values for PPDG, Energy, Time, Length, X
    ↪ , Y, Z columns, the values should be within the range of the original dataset

# Generate random values for the BIB datasetimport numpy as np

np.random.seed(42)

# Generate random samples for PPDG
bib_ppdg = np.random.choice(df_sample['PPDG'], 500000)

# Define parameters for energy distribution
mean_energy = 0.0000392  # Mean energy value
energy_std = 0.0001767    # Standard deviation for energy

# Define parameters for time distribution
mean_time = 651.30976     # Mean time value
time_std = 1648.124      # Standard deviation for time

# Generate random samples for energy and time
num_samples = 500000
bib_energy = []
bib_time = []

# Generate samples until desired number is reached
while len(bib_energy) < num_samples:
    energy_sample = np.random.normal(mean_energy, energy_std)
    time_sample = np.random.normal(mean_time, time_std)
    if energy_sample > 0 and time_sample > 0:
        bib_energy.append(energy_sample)
        bib_time.append(time_sample)

# Convert lists to arrays
bib_energy = np.array(bib_energy)
bib_time = np.array(bib_time)
import numpy as np

# Assuming bib_time is your existing array
indices = np.random.choice(len(bib_time), 50000)
bib_time[indices] += 4000



bib_length = np.random.uniform(df_sample['Length'].min(), df_sample['Length'].max(), 500000)
bib_x = np.random.uniform(df_sample['X'].min(), df_sample['X'].max(), 500000)
bib_y = np.random.uniform(df_sample['Y'].min(), df_sample['Y'].max(), 500000)
bib_z = np.random.uniform(df_sample['Z'].min(), df_sample['Z'].max(), 500000)

# Create a new BIB dataframe
df_bib_new = pd.DataFrame({
    'PPDG': bib_ppdg,
    'Energy': bib_energy,
    'Time': bib_time,
    'Length': bib_length,
    'X': bib_x,
    'Y': bib_y,
    'Z': bib_z,
    'Dataset': 'BIB'
})

#add Hard rows to the new BIB dataset from df_sample, take same values for hard rows as in the
    ↪ original dataset

# Select 500,000 random Hard rows from the df_sample dataset
df_hard_sample = df_sample[df_sample['Dataset'] == 'Hard'].sample(n=400000, random_state=42)

# Concatenate the BIB and Hard dataframes
df_combined_new = pd.concat([df_bib_new, df_hard_sample])

#plot time vs energy distribution for BIB and Hard datasets using a scatter plot for df_combined_new
    ↪ , use shared axes
import matplotlib.pyplot as plt

# Create a figure with two subplots
fig, ax = plt.subplots(1, 2, figsize=(12, 6), sharex=True, sharey=True)

# Plot the scatter plot for BIB dataset
ax[0].scatter(df_combined_new[df_combined_new['Dataset'] == 'BIB']['Time'], df_combined_new[
    ↪ df_combined_new['Dataset'] == 'BIB']['Energy'], color='blue', label='BIB', alpha=0.1)
ax[0].set_title('BIB␣Dataset')
ax[0].set_xlabel('Time')
ax[0].set_ylabel('Energy')

# Plot the scatter plot for Hard dataset
ax[1].scatter(df_combined_new[df_combined_new['Dataset'] == 'Hard']['Time'], df_combined_new[
    ↪ df_combined_new['Dataset'] == 'Hard']['Energy'], color='red', label='Hard', alpha=0.1)
```

```
ax[1].set_title('Hard␣Dataset')
ax[1].set_xlabel('Time')
ax[1].set_ylabel('Energy')
#set x axis limits to 12000
plt.xlim(0, 12000)
# Show the plot
plt.show()



# Print the new combined dataframe
print(df_combined_new)
import tensorflow as tf
from tensorflow import keras
from keras.models import Sequential, load_model

from sklearn.preprocessing import LabelEncoder
from sklearn.model_selection import train_test_split
from sklearn.metrics import accuracy_score
import tensorflow as tf

from sklearn.preprocessing import LabelEncoder


df_sample_subset = df_combined_new.groupby('Dataset').apply(lambda x: x.sample(n=400000,
    ↪ random_state=42)).reset_index(drop=True)

# Define the features and target variable
import tensorflow as tf

from sklearn.preprocessing import LabelEncoder
from sklearn.metrics import accuracy_score
import numpy as np

# Define the search space for different architectures
# search_space = [
#     {'hidden_layers': [6],
#      'units': [16,32, 64, 128],
#      'activation': ['relu', 'tanh', 'sigmoid','softmax']}
# ]

# # Define a function to create a model based on the given parameters
# def create_model(input_shape, num_classes, hidden_layers, units, activation):
#     model = Sequential()
#     model.add(Dense(units, activation=activation, input_shape=input_shape))
#     for _ in range(hidden_layers - 1):
#         model.add(Dense(units, activation=activation))
#     model.add(Dense(num_classes, activation='softmax'))
#     model.compile(optimizer='adam', loss='sparse_categorical_crossentropy', metrics=['accuracy'])
#     return model

# # Define the features and target variable
X = df_sample_subset[['Energy', 'Time', 'Length', 'X', 'Y', 'Z']]
y = df_sample_subset['Dataset']

# # Encode the target variable
# encoder = LabelEncoder()
# y_encoded = encoder.fit_transform(y)
# # Split the data into training and testing sets
# X_train, X_test, y_train, y_test = train_test_split(X, y_encoded, test_size=0.5, random_state=42)

# best_model = None
# best_accuracy = 0.0

# # Randomly search the architectures
# for _ in range(10):  # Try 10 different random configurations
#     config = np.random.choice(search_space)
#     hidden_layers = np.random.choice(config['hidden_layers'])
#     units = np.random.choice(config['units'])
#     activation = np.random.choice(config['activation'])

#     model = create_model(input_shape=(X_train.shape[1],), num_classes=len(encoder.classes_),
#                          hidden_layers=hidden_layers, units=units, activation=activation)
#     history = model.fit(X_train, y_train, epochs=5, batch_size=32, validation_data=(X_test, y_test
    ↪ ), verbose=0)
#     _, accuracy = model.evaluate(X_test, y_test, verbose=0)

#     print(f'Architecture: hidden layers={hidden_layers}, units={units}, activation={activation},
    ↪ Accuracy: {accuracy}')

#     if accuracy > best_accuracy:
#         best_model = model
#         best_accuracy = accuracy

# print(f'Best Accuracy: {best_accuracy}')

# # Evaluate the best model on the test set
# test_loss, test_accuracy = best_model.evaluate(X_test, y_test)
# print(f'Test Accuracy of Best Model: {test_accuracy}')
# #save best model to a file
# best_model.save('best_model.h5')


#load the best model from the file
# Load the best model from the file
best_model = load_model('best_model.h5')


from keras.models import load_model

# Load the saved best model from the file
best_model = load_model('best_model.h5')
encoder = LabelEncoder()
# y_encoded = encoder.fit_transform(y)
# Preprocess the entire dataset
X = df_sample_subset[['Energy', 'Time', 'Length', 'X', 'Y', 'Z']]
y = encoder.fit_transform(df_sample_subset['Dataset'])

# Make predictions on the entire dataset
predictions = best_model.predict(X)

# Convert the predictions to class labels
predicted_labels = np.argmax(predictions, axis=1)

# Evaluate the performance of the model
accuracy = accuracy_score(y, predicted_labels)
print(f'Accuracy␣on␣the␣entire␣dataset:␣{accuracy}')
```

```
#display the model architecture and summary
# Display the model architecture
best_model.summary()
#plot the model architecture
from keras.utils import plot_model

# Plot the architecture of the best model
plot_model(best_model, to_file='best_model_architecture.png', show_shapes=True, show_layer_names=
    ↪ True)
```

9.5 Using hls4ml

Listing 7: This code loads the best model file and then invokes hls4ml and HLS to create an HLS project
hls_ml.py

```
import tensorflow as tf
import hls4ml
import os

# Set HLS PATH
os.environ['XILINX_VIVADO'] = '/home/vlsilab13/Xilinx/Vivado/2020.1'
os.environ['PATH'] = os.environ['XILINX_VIVADO'] + '/bin:' + os.environ['PATH']

# Create a function to load the Keras model from file
def load_keras_model(file_path):
    return tf.keras.models.load_model(file_path)

# Load the model from file
model_file_path = 'bestmodel.h5'
dummy_model = load_keras_model(model_file_path)

# Convert the TensorFlow model to hls4ml configuration
config = hls4ml.utils.config_from_keras_model(dummy_model, granularity='model')

# Set loop unroll factor in the HLS configuration
config['HLSConfig'] = {'IOType': 'io_stream', 'Optimization': 'Performance', 'UnrollFactor': 2}

# Convert the model to HLS
hls_model = hls4ml.converters.convert_from_keras_model(dummy_model,
                                                       hls_config=config,
                                                       output_dir='test/test_1',
                                                       part='xcu250-figd2104-2L-e')

# Visualize the model
hls4ml.utils.plot_model(hls_model, show_shapes=True, show_precision=True)

# Synthesize using HLS backend
hls_model.compile()
hls_model.build(csim=False)
```