frontiers in Big Data

ORIGINAL RESEARCH
published: 23 March 2022
doi: 10.3389/fdata.2022.828666

Graph Neural Networks for Charged Particle Tracking on FPGAs

Abdelrahman Elabd[1], Vesal Razavimaleki[2], Shi-Yu Huang[3], Javier Duarte[2], Markus Atkinson[4], Gage DeZoort[5], Peter Elmer[5], Scott Hauck[6], Jin-Xuan Hu[3], Shih-Chieh Hsu[6,7], Bo-Cheng Lai[3], Mark Neubauer[4]*, Isobel Ojalvo[5], Savannah Thais[5] and Matthew Trahms[6]*

[1] Department of Physics and Astronomy, University of Pennsylvania, Philadelphia, PA, United States, [2] Department of Physics, University of California, San Diego, La Jolla, CA, United States, [3] Department of Electrical and Computer Engineering, National Yang Ming Chiao Tung University, Hsinchu, Taiwan, [4] Department of Physics, University of Illinois at Urbana-Champaign, Champaign, IL, United States, [5] Department of Physics, Princeton University, Princeton, NJ, United States, [6] Department of Electrical and Computer Engineering, University of Washington, Seattle, WA, United States, [7] Department of Physics, University of Washington, Seattle, WA, United States

The determination of charged particle trajectories in collisions at the CERN Large Hadron Collider (LHC) is an important but challenging problem, especially in the high interaction density conditions expected during the future high-luminosity phase of the LHC (HL-LHC). Graph neural networks (GNNs) are a type of geometric deep learning algorithm that has successfully been applied to this task by embedding tracker data as a graph—nodes represent hits, while edges represent possible track segments—and classifying the edges as true or fake track segments. However, their study in hardware- or software-based trigger applications has been limited due to their large computational cost. In this paper, we introduce an automated translation workflow, integrated into a broader tool called `hls4ml`, for converting GNNs into firmware for field-programmable gate arrays (FPGAs). We use this translation tool to implement GNNs for charged particle tracking, trained using the TrackML challenge dataset, on FPGAs with designs targeting different graph sizes, task complexites, and latency/throughput requirements. This work could enable the inclusion of charged particle tracking GNNs at the trigger level for HL-LHC experiments.

Keywords: graph neural networks, FPGAs, tracking, LHC, trigger

OPEN ACCESS

Edited by:
Bo Jayatilaka, Fermilab (DOE), United States

Reviewed by:
Matteo Cremonesi, Fermi Research Alliance, United States
Chase Shimmin, Yale University, United States

****Correspondence:***
Javier Duarte
jduarte@ucsd.edu
Mark Neubauer
msn@illinois.edu

Specialty section:
This article was submitted to Big Data and AI in High Energy Physics, a section of the journal Frontiers in Big Data

Received: *03 December 2021*
Accepted: *26 January 2022*
Published: *23 March 2022*

Citation:
Elabd A, Razavimaleki V, Huang S-Y, Duarte J, Atkinson M, DeZoort G, Elmer P, Hauck S, Hu J-X, Hsu S-C, Lai B-C, Neubauer M, Ojalvo I, Thais S and Trahms M (2022) Graph Neural Networks for Charged Particle Tracking on FPGAs. Front. Big Data 5:828666. doi: 10.3389/fdata.2022.828666

1. INTRODUCTION

In high energy physics (HEP), charged particle tracking (Strandlie and Frühwirth, 2010; Amrouche et al., 2020) is a crucial task necessary for the accurate determination of the kinematics of the particles produced in a collision event, including the position, direction, and momentum of the particles at their production points. This task leverages specialized detectors positioned close to the beam collision area in a strong magnetic field. When charged particles are created in the collisions, their trajectories bend in the magnetic field and they ionize the material of these detectors as they move outwards from the production point, providing position measurements along the trajectory of each particle. The objective of tracking algorithms is to identify the individual trajectories of these charged particles and extract relevant particle kinematics. Current tracking algorithms (Frühwirth, 1987; Billoir, 1989; Billoir and Qian, 1990; Mankel, 1997; Chatrchyan et al., 2014; Aaboud et al., 2017) scale worse than quadratically with the number of hits, which is expected to increase dramatically at higher beam intensities due to the presence of simultaneous proton-proton interactions (or *pileup*) and for more granular, more sensitive detectors. This motivates

the study of alternative algorithms with different computational scaling. Another important consideration is the ability to accelerate these algorithms using highly-parallel heterogeneous computing resources like graphics processing units (GPUs) and field-programmable gate arrays (FPGAs) as further improvements in single-core CPU performance may be limited (Dennard et al., 1974; Esmaeilzadeh et al., 2011). Recent efforts (Farrell et al., 2018; Ju et al., 2019; DeZoort et al., 2021) have demonstrated the effectiveness of graph neural networks (GNNs) to correctly classify "segments" belonging to tracks. Graph-based approaches are well suited to this task because tracking data can be naturally encoded as a graph structure (Shlomi et al., 2020) and GNNs consider local information between pairs of hits to learn relationships between them in order to "connect the dots" and infer tracks. In other words, track finding is an example of edge classification on a graph data structure.

In this paper, we develop an automatic translation toolflow to convert GNNs (DeZoort et al., 2021) for segment classification, based on the interaction network (IN) architecture (Battaglia et al., 2016, 2018), to FPGA firmware. FPGA implementations enable efficient processing, in both speed and energy consumption for large HEP datasets. They may also enable the use of GNNs in the high-throughput, FPGA-based data filter system, known as the level-1 trigger (L1T) (ATLAS Collaboration, 2017; Aad et al., 2020; CMS Collaboration, 2020; Sirunyan et al., 2020), which has strict sub-microsecond latency requirements that only FPGAs or application-specific integrated circuits (ASICs) can meet. For instance, in the upgraded CMS design, a latency of 4 μs (plus 1 μs for data transfer) and an event throughput of 2.22 MHz are required for the L1 track trigger (CMS Collaboration, 2020). Conversely, in situations when a longer latency is permissible such as the high-level trigger (HLT) (Trocino, 2014) or offline processing, they may be used in coprocessing applications and scale to larger tasks. Our automatic translation code is integrated with `hls4ml` (Duarte et al., 2018; Loncar et al., 2021), a more general compiler for converting machine learning (ML) algorithms into FPGA firmware. We evaluate the resource usage, latency, and tracking performance of a variety of different implementations based on the benchmark TrackML dataset (Amrouche et al., 2020).

This paper is structured as follows. In section 2, we briefly recapitulate related work. Section 3 defines the benchmark TrackML dataset and task, including the data preprocessing and graph encoding. Section 4 describes the IN model used for the track segment classification task. Section 5 describes the `hls4ml` user interface, while section 6 describes the FPGA designs. Section 7 summarizes the results of the FPGA firmware synthesis, including measurements of performance, timing, and FPGA resources. Finally, a summary and outlook are given in section 8.

2. RELATED WORK

GNNs have been explored for particle physics applications (Duarte and Vlimant, 2020; Shlomi et al., 2020), including jet identification (Moreno et al., 2020a,b; Qu and Gouskos, 2020), pileup mitigation (Arjona Martínez et al., 2019; Li et al., 2021), calorimeter energy measurements (Qasim et al., 2019), particle-flow reconstruction (Kieseler, 2020; Pata et al., 2021), and charged particle tracking (Farrell et al., 2018; DeZoort et al., 2021; Ju et al., 2021). Automatic translation of ML algorithms into FPGA firmware has also been studied for HEP tasks. Using `hls4ml`, several implementations for HEP-specific tasks have been provided for fully connected neural networks (NNs), autoencoders, boosted decision trees, and convolutional NNs (Duarte et al., 2018; Loncar et al., 2020; Summers et al., 2020; Aarrestad et al., 2021; Coelho et al., 2021; Govorkova et al., 2021). This tool has also been applied extensively for tasks in the HL-LHC upgrade of the CMS L1T system, including anomaly detection, muon energy regression and identification, tau lepton identification, and vector boson fusion event classification (CMS Collaboration, 2020). Moreover, a GNN model known as a GarNet was studied for calorimeter energy regression and deployed on FPGAs using `hls4ml` in Iiyama et al. (2021). Our current work extends this by allowing more generic IN architectures to be converted with `hls4ml`, permitting the specification of a variable graph adjacency matrix as input, and supporting per-node or per-edge outputs as well as per-graph outputs. This paper also supersedes an earlier version of this work in Heintz et al. (2020).

Hardware acceleration of GNN inference, and graph processing in general, has been an active area of study (Besta et al., 2019; Gui et al., 2019; Ming Xiong, 2020). Nurvitadhi et al. (2014), Ozdal et al. (2016), Auten et al. (2020), Geng et al. (2020), Kiningham et al. (2020), Yan et al. (2020), and Zeng and Prasanna (2020) describe various other examples of GNN acceleration architectures. While these frameworks are applicable to various graph processing tasks, they may not apply to the strict latency requirements of the LHC trigger and they typically require the user to specify the design in a highly specialized format.

3. TRACKML DATA

The results presented in this paper are based on the TrackML dataset. The TrackML dataset is a simulated set of proton-proton collision events originally developed for the TrackML Particle Tracking Challenge (Amrouche et al., 2020). Each event is generated with 200 pileup interactions on average, simulating the high-pileup conditions expected at the HL-LHC. Each event data record contains 3D hit positions, additional "cell" information (e.g., directional information), hit truth information, including the true location of each hit, and the features of the true charged particle that generated each hit. In particular, truth particles are specified by particle IDs (p_{ID}) and three-momentum vectors ($\mathbf{p}$).

The TrackML detector emulates a generic LHC silicon-based tracker, containing discrete layers of silicon sensors immersed in a strong, inhomogeneous magnetic field. We focus specifically on the innermost silicon pixel layers, a highly-granular set of 4 barrel and 14 endcap layers in the innermost tracker regions with a spatial resolution of 50 $\times$ 50 μm. These pixel layers are illustrated in **Figure 1**.

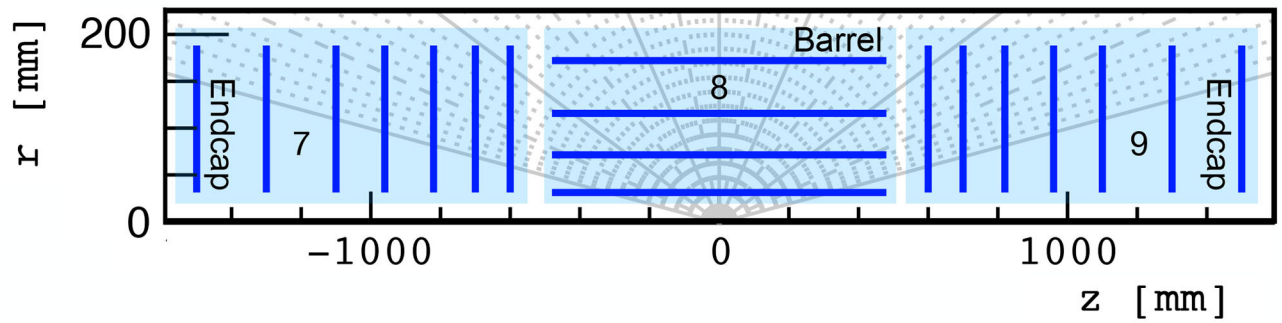

FIGURE 1 | TrackML pixel detector regions labeled 7, 8, and 9, used in this study, consisting of 7, 4, and 7 layers, respectively.

The dataset assumes a right-handed Cartesian coordinate system is defined with the z axis oriented along the beam axis, the x axis toward the center of the collider, and the y axis oriented upward. The x and y axes define the transverse plane, while the z axis identifies the longitudinal direction. The radial coordinate is defined as $r = \sqrt{x^2 + y^2}$. The azimuth angle ϕ is computed with respect to the x axis in radians in the $[-\pi, \pi]$ range. The polar angle θ is measured from the positive z axis and is used to compute the pseudorapidity $\eta = -\log(\tan(\theta/2))$ and the pseudoangular separation $\Delta R = \sqrt{\Delta\phi^2 + \Delta\eta^2}$. The transverse momentum (p_T) is the projection of momentum on the (x, y) plane. We use natural units such that $c = 1$ and express energy and momentum in units of electronvolt (eV).

3.1. Graph Construction

Each event's tracker hits are encoded into a *hitgraph* based on a set of selection criteria applied to the candidate nodes and edges. A set of truth filters are applied to hits before they are assigned to graph nodes. In particular, a p_T *filter* rejects hits generated by truth particles with p_T below some minimum threshold p_T^{min}, a *noise filter* rejects hits generated by noise (no associated truth particle), and a *same-layer filter* rejects all but one hit per layer for each truth particle. After this initial hit filtering yields a reduced set of nodes, we choose to connect certain nodes with edges which are most likely to represent true track segments based on geometric considerations. These chosen edges are a superset of all possible edges that can be classified, therefore it's important to accept as many true track segments as possible, i.e., ensure high *efficiency*, defined as the ratio of true track segment edges contained in the graph to the total number of possible true track segments. Conversely, permitting too many edges at this stage, such as a fully-connected hitgraph with $\frac{1}{2}n_{\text{nodes}}(n_{\text{nodes}} - 1)$ edges, can result in a highly inefficient or intractable computation problem, due to low *purity*, defined as the number of true track segment edges divided by the total number of edges in the graph. This represents a fundamental trade-off between different edge construction algorithms: they must simultaneously maximize efficiency without minimizing purity.

In this work, we use a purely geometric graph construction algorithm that determines whether or not to create an edge with features a_{ij} between hits i and j. Only pixel detector hits are considered, pseudorapidity is restricted to $\eta \in [-4, 4]$, and the noise and same-layer hit filters are applied. The same-layer filter introduces an ambiguity in defining edges between the barrel and innermost endcap layers. Specifically, barrel hits generated by the same particle could produce multiple true edges incoming to a single endcap hit. The resulting triangular edge pattern conflicts with the main assumption of the same-layer filter, that only one true track segment exists between each subsequent layer. Thus, a *barrel intersection* cut is applied, in which edges between a barrel layer and an innermost endcap layer are rejected if they intersect with any intermediate barrel layers. In addition to the barrel intersection cut, edges must also satisfy constraints on the geometric quantities $z_0 = z_i - r_i\frac{z_j - z_i}{r_j - r_i}$ and $\Delta\phi/\Delta r = \frac{\phi_j - \phi_i}{r_j - r_i}$. The restrictions are $z_0 < 15\,\text{cm}$ and $\Delta\phi/\Delta r < 0.0006$ and $p_T^{\text{min}} = 2\,\text{GeV}$.

Even after these geometric and truth particle p_T restrictions, a single TrackML event can feature thousands of nodes and edges. In order to keep the graph sizes manageable for resource-constrained environments, we can subdivide each event graph into a certain number of ϕ and η sectors depending on the latency and resource constraints of the system and the range of applicability of the given implementation. Example graphs for 2 different sectors in one event when subdividing an event into 8 ϕ sectors and 2 η sectors are shown in **Figure 2**.

Based on these graph construction criteria, we can measure the efficiency and purity of the resulting graphs. These are shown for different choices for the number of ϕ and η sectors in **Figure 3** based on 50 events in `train_1` TrackML sample. In particular for 8 ϕ sectors and 2 η sectors, the graphs retain an efficiency of 98% and a purity of 57%.

In addition to the need to subdivide the graphs to reduce size, for a fixed-latency FPGA implementation, hardware resources cannot be dynamically reallocated to accept variable-size input arrays, so the graph sizes must be made uniform. Thus, we typically consider a fixed maximum input graph size. One common choice is to truncate the graphs based on the 95th percentile graph size for each sector.[1] Thus, only 5% of the

[1]We note that for up to 2 η sectors, each of the sectors has the same hit and track multiplicity distribution because the multiplicity depends on $|\eta|$ but not on ϕ. In general, for a larger number of η sectors, a variable maximum graph size could be chosen depending on the $|\eta|$ range of the sector.

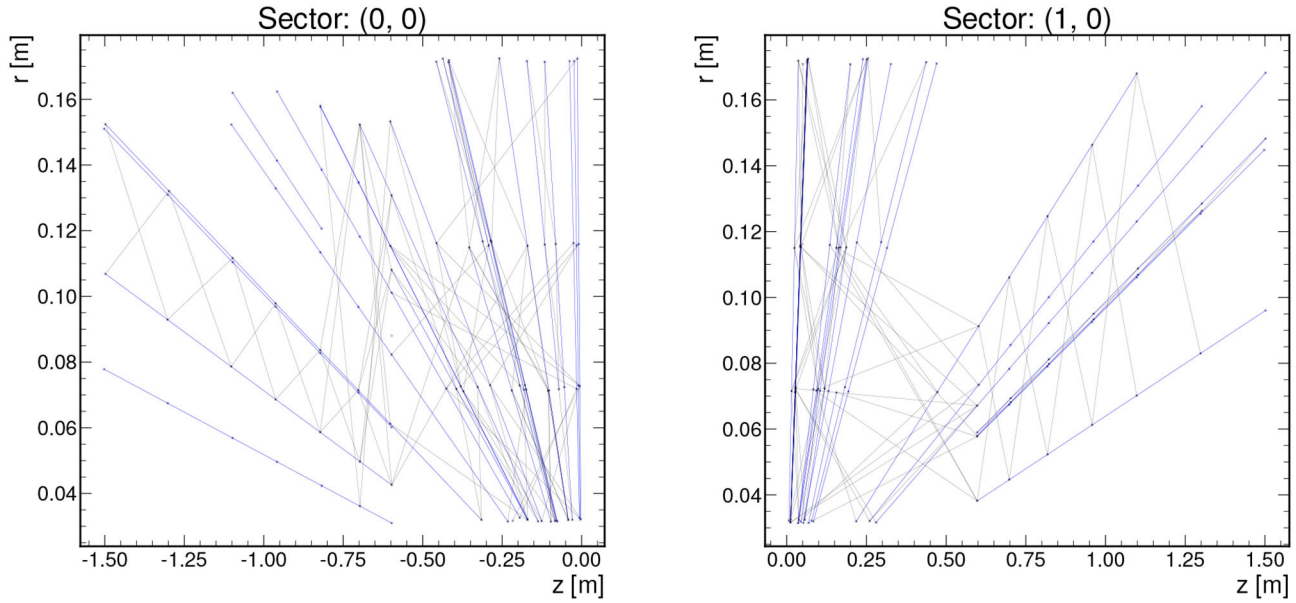

FIGURE 2 | Example graphs showing 2 of the sectors for one event with $p_T^{min} = 2$ GeV, $z_0 < 15$ cm, $\Delta\phi/\Delta r < 0.0006$, 8 ϕ sectors, and 2 η sectors. True track segments are denoted by blue edges, while false track segments are denoted by gray.

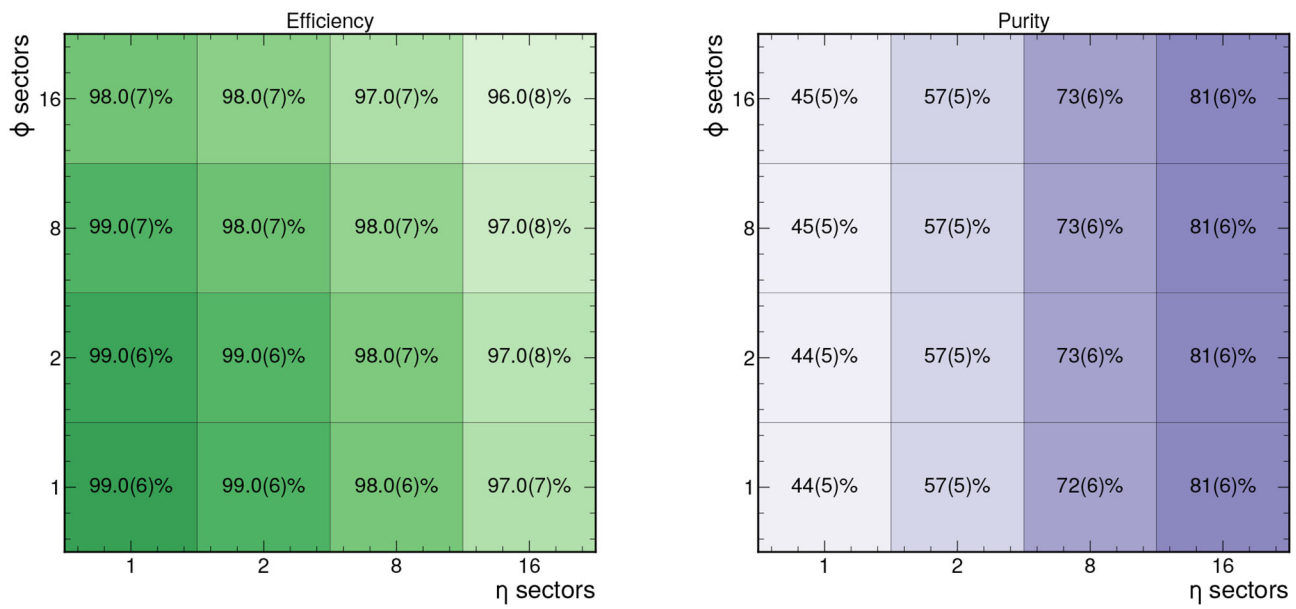

FIGURE 3 | Efficiency **(left)** and purity **(right)** of the hitgraphs studied for different numbers of η and ϕ sectors based on 50 events in `train_1`.

input graphs will be truncated. For smaller graphs that have fewer nodes or edges, we zero-pad the feature matrices to create "null" nodes and add connections between "null" nodes to create "null" edges. We discuss the effects of the graph truncation and zero-padding on the network performance in section 7.

Figure 4 shows the 95th percentile for the number of nodes and edges in each sector depending on the number of sectors chosen. For example, the 95th percentile graph size for 8 ϕ sectors and 2 η sectors is 113 nodes and 196 edges for this 2 GeV graph construction. Depending on the range of applicability for a given FPGA implementation, a different graph construction and segmentation strategy can be adopted. In particular, if a more relaxed set of graph construction criteria is adopted, a greater number of nodes and edges will be included per event. In the **Supplementary Material**, we demonstrate how the number of nodes and edges vary when considering 1 GeV graphs. In

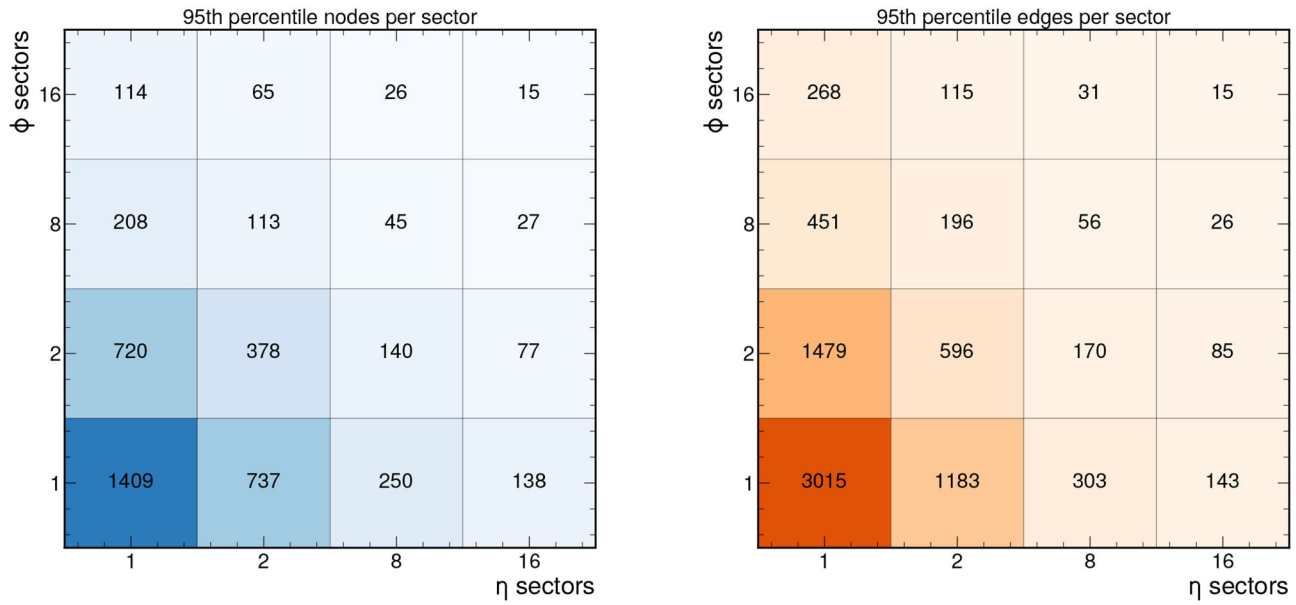

FIGURE 4 | 95th percentile of the number of nodes and edges in each sector for the 2 GeV graphs as a function of the number of η and ϕ sectors, based on 50 events in `train_1`.

particular, the 95th percentile for the number of nodes and edges per event is nearly 6,500 and 20,000, respectively. To accommodate these denser hitgraphs in resource-constrained implementations, a finer segmentation is required.

4. INTERACTION NETWORK

Fundamentally, GNNs produce new graph embeddings, leveraging them to produce graph-level, edge-level, or node-level predictions. One iteration of an IN contains two "update" functions, ϕ, and an "aggregation" operation, which, for simplicity, we take to be a simple summation.

The *edge block* computes a four-dimensional output a'_{ij} for each edge, known as the updated edge feature or "message."

$$a'_{ij} = \phi_{R,1}(x_i, x_j, a_{ij}) \,, \tag{1}$$

where x_i and x_j are the input features of node i and j, respectively, which are the (r, ϕ, z) coordinates of the corresponding hit, and a_{ij} is the set of input edge features $(\Delta r, \Delta\phi, \Delta z, \Delta R)$. These messages are subsequently aggregated (summed) over the corresponding connected nodes belonging to the neighborhood $\mathcal{N}(i)$ of node i.

$$\overline{a}'_i = \sum_{j \in \mathcal{N}(i)} a'_{ij} \tag{2}$$

These two steps are known as the message-passing operation. The aggregation function, here taken to be summation, maps edge-specific information to node-specific outputs by compiling information based on the connected node indices. To apply generically to unordered graph-structured data, the chosen aggregation function must be invariant to permutations of their inputs, and should take variable numbers of arguments. Other examples include an element-wise mean, maximum, and minimum. This construction ensures permutation equivariance of the GNN for edge- and node-level outputs.

The *node block* computes a three-dimensional output for each node

$$x'_i = \phi_O(x_i, \overline{a}'_i) \tag{3}$$

that can be thought of as an update of the node features, which takes into account the previous node features, and one round of message passing among neighboring nodes.

An additional partial edge block gives the final one-dimensional *edge weights*

$$a''_{ij} = \phi_{R,2}\left(x'_i, x'_j, a'_{ij}\right), \tag{4}$$

In this way, we produce edge weights from the re-embedded graph with node features and edge features.

The full forward pass, comprised of edge and node blocks used to predict edge weights, is shown in **Figure 5**. The functions $\phi_{R,1}$, $\phi_{R,2}$, and ϕ_O are multilayer perceptrons (MLPs) with rectified linear unit (ReLU) activation functions (Nair and Hinton, 2010; Glorot et al., 2011) on the hidden layers. The ReLU activation function behaves as an identity function for positive inputs and saturates at 0 for negative inputs. Notably, the $\phi_{R,2}$ outputs have sigmoid activations and we optimize the binary cross-entropy (BCE) loss between the truth targets and the outputs, such that the resulting edge weights $a''_{ij} \in [0, 1]$ can be interpreted as independent probabilities that each edge is a track segment.

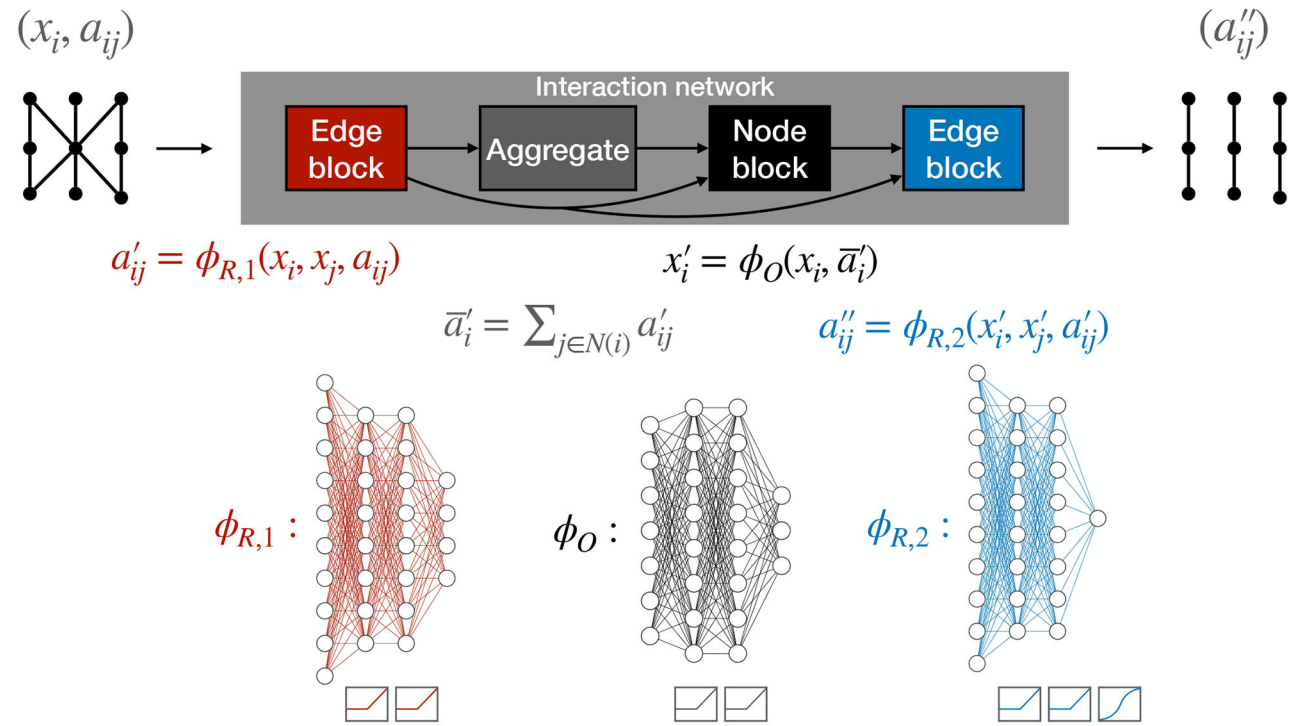

FIGURE 5 | The complete IN forward pass with the edge, edge-aggregation, and node blocks. The functions $\phi_{R,1}$, $\phi_{R,2}$, and ϕ_O are multilayer perceptrons (MLPs) with rectified linear unit (ReLU) activation functions on the hidden layers. In addition, the $\phi_{R,2}$ outputs have a sigmoid activation.

However, we note that this probabilistic interpretation is not required to build a tracking algorithm based on this IN.

Throughout the following studies, the architecture in **Figure 5** is held at a constant size of 528 trainable parameters, corresponding to 8 hidden units (h.u.) per layer in each of the MLPs. Assuming every MLP layer has the same number of h.u., 8 h.u. per layer is sufficient to recover the maximum classification accuracy with models trained on $p_{\mathrm{T}}^{\mathrm{min}} = 2\,\mathrm{GeV}$ segmented graphs. In the following studies, models are trained on graphs built with $p_{\mathrm{T}}^{\mathrm{min}} = 2\,\mathrm{GeV}$. A total of about 28 k graph sectors (corresponding to 1770 total events and 16 sectors per event) belonging to the TrackML `train_1` sample are randomly divided into 70% for training, 20% for validation, and 10% for testing. The Adam optimizer is used to facilitate training (Kingma and Ba, 2015). It is configured with a learning rate of 10^{-3} and $\epsilon = 10^{-8}$, and the model is trained for 1 epoch, which we find is enough time for the model to reach convergence.

5. `hls4ml` USER INTERFACE

The `hls4ml` workflow performs automatically the task of translating a trained NN, specified by the model's architecture, weights, and biases, into the specification of a hardware accelerator. Because every application is different, the goal of the `hls4ml` package is to empower the user to perform this optimization through automated NN translation and design iteration. `hls4ml` leverages HLS to generate hardware modules from code written in high-level programming languages like C/C++ (Numan et al., 2020). Although it may lead to slightly less optimal performance than RTL-based design, HLS-based design has significant benefits: it raises the level of abstraction, reduces the iteration time, simplifies the validation phase, and enables greater exploration and evaluation of design alternatives.

Figure 6 shows the schematic of a typical workflow. The first part of the workflow illustrated in red depicts the model training and optimization performed with PyTorch Geometric (PyG) (Fey and Lenssen, 2019). The blue section of the workflow is performed with `hls4ml`, which translates the model into an HLS project that can subsequently be synthesized and implemented on an FPGA or ASIC, as depicted by the black section.

The `hls4ml` workflow is built on the concept of independent NN "layers" that process some input data to produce some output data. Each layer and activation type is implemented as a separate configurable module customized to perform that specific operation. During the `hls4ml` conversion, these modules are composed in the correct way to perform the inference for a full ML model.

Specifically, this conversion step automatically generates a top-level C++ executable that uses predefined C++ templates representing different NN layers provided through the `nnet_tools` library in `hls4ml`. For this GNN application, the edge block, node block, and edge-aggregation block are each considered separate layers. `hls4ml` also provides a number of

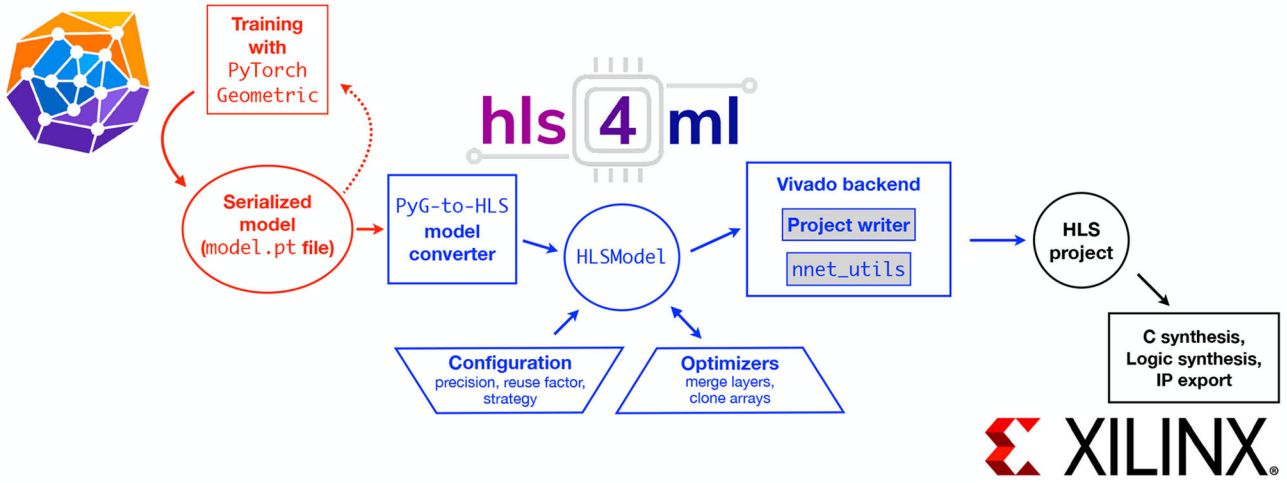

FIGURE 6 | The workflow to translate a GNN model into an FPGA implementation using `hls4ml`. The red boxes **(left)** describe the model training steps performed within ML software frameworks like PYTORCH GEOMETRIC. The `hls4ml` configuration and conversion steps are shown in the blue boxes **(center)**. A model converter translates the model from PYTORCH GEOMETRIC into an intermediate HLSMODEL representation. This representation can be further configured and optimized. Different backend writers can be used to export the model into a given vendor-specific language, such as Vivado HLS. The black boxes **(right)** illustrate possible ways to export and integrate the HLS project into a larger hardware design for example for Xilinx FPGAs.

configurable parameters which can help the user explore and customize the space of latency, throughput, power, and resource usage tradeoffs for their application.

To provide flexibility and ease-of-use, `hls4ml` provides both a programming API and visualization capabilities. **Figure 6** also illustrates the internal structure of the `hls4ml` PYTHON package. The package first converts the user-specified model into a common internal representation of the network graph. Converters exist for KERAS, TENSORFLOW, PYTORCH, and ONNX model formats. For this work, a new converter for the GNN model provided as a PYTORCH GEOMETRIC model was added. At the conversion step, the user-provided configuration is also attached to the model. We note additional user-provided information is required: the order of the modules.

One key feature of the programming API is the capability to execute the bit-accurate emulation of the generated HLS-synthesizable code in the PYTHON environment, for example as a Jupyter Notebook. The `hls4ml` API enables a workflow where inference can be performed on NUMPY (Harris et al., 2020) array data in PYTHON. In addition to evaluating the `hls4ml` model output, users can access the detailed output of any layer of the network, which can aid in debugging and performing hyperparameter optimization for quantized models. For this work, the capability of running inference on models with multiple inputs (node attributes, edge attributes, and the edge index) was added.

6. FPGA HLS IMPLEMENTATIONS

To meet the strict latency and throughput requirements of the L1T, we developed a *throughput-optimized* design, in which the processing of multiple input graphs overlaps in time.

Conversely, we also consider a *resource-optimized* design, which requires more clock cycles but also consumes far fewer resources. This design has the advantage that it scales to larger input graphs.

Tests of the `hls4ml` implementation target a Xilinx Virtex UltraScale+ VU9P FPGA (part number `xcvu9p-flga2104-2L-e`) with a 5 ns clock period. The target FPGA has 6,840 digital signal processing slices (DSPs), 1,182,240 look-up tables (LUTs), 2,364,480 flip-flops (FFs), and 75.9 Mb of block random access memory (BRAM) (Xilinx, Inc., 2021). As discussed in section 3.1, the input graph size can be adjusted by segmenting the detector more finely in η and ϕ sectors. We find that large input graph sizes prohibit the use of the throughput-optimized design. Therefore, to make scans of resources and timing tractable, we further synthesize designs for graphs of up to 28 nodes and 56 edges. For the resource-optimized design, we also demonstrate how the design scales to larger graphs by presenting results for up to 1,344 nodes and 2,688 edges. For all results shown, we fix the ratio of the number of edges to nodes to 2, as we empirically find that this is close to the observed ratio for the 2 GeV graphs we consider.

6.1. HLS Preprocessor Directives

The FPGA HLS implementation is written using Vivado HLS (Xilinx, Inc., 2020) and integrated with the transpiler `hls4ml`. Vivado HLS makes use of preprocessor directives, called *pragmas*, in order to specify certain high-level design choices. A full description of the two different designs requires an understanding of some of these pragmas, namely `ARRAY_PARTITION`, `UNROLL`, and `PIPELINE`.

Applying `ARRAY_PARTITION` to a given array will partition it into several smaller arrays by one of three different methods.

Block partitioning with a factor of N, will partition the array into N consecutive blocks. Cyclic partitioning with a factor of N, will create N smaller arrays by interleaving elements from the original array such that the ith smaller array contains elements $i, i+N, i+2N, \ldots$ from the original array. Complete partitioning will fully decompose an array into its individual elements. The benefit of partitioning is that different functions can concurrently access and operate on the separate, smaller arrays, but this comes at the cost of utilizing more memory registers.

The UNROLL pragma is used within for-loops. For a for-loop that normally has M iterations, fully unrolling this loop will create M physical copies of the for-loop logic, so that each iteration can run concurrently. Partially unrolling this loop with a factor of $N < M$ will create N copies of the for-loop, each of which will iterate $\lceil M/N \rceil$ times; Vivado HLS does not require that M is an integer multiple of N. Like partitioning, unrolling sacrifices some resources in exchange for lower latency.

Applying the PIPELINE pragma on a function will minimize the function's initiation interval (II), which is the wait time required before a new input can be processed. Pipelining a function allows it to accept a new input before it has finished operating on an earlier input. For example, consider a simple three-layer MLP, and two different inputs for the MLP: A and B. Typically, one would send input A into the MLP, wait for it to pass through all three layers, and then send input B into the MLP. With pipelining, one can send input A through the first layer of the MLP, and then send input B through the first layer as soon as input A reaches the second layer. Likewise, the second layer can accept input B once input A has made it to the third layer. With pipelining, idle resources are used in order to minimize II without requiring any additional resources. The amount of pipelining is limited by the logic and timing of the relevant design. For example, it is possible to pipeline a for-loop as long as each iteration is independent.

6.2. Throughput-Optimized Design

In the throughput-optimized design, pipelining is performed at the level of the IN edge block, node block, and edge-aggregation block, and the amount of pipelining is tunable through the *reuse factor* (RF) parameter, which controls the II of each block. In conjunction with block-level pipelining, all loops are fully unrolled to decrease latency and all arrays are completely partitioned to allow concurrent access to each element. These are implemented through the HLS pragmas described above. Fully unrolling the arrays places a constraint on how large the input graphs can be, given the limitations imposed by the Vivado HLS compiler. In practice, we find graphs of 28 nodes and 56 edges are near the maximum we can consider with this design.

The edge block receives as input three fully partitioned arrays: the node feature matrix with arbitrary fixed-point precision, the edge feature matrix with arbitrary fixed-point precision, and the edge index matrix with an arbitrary integer precision. The output of the edge block is the updated matrix of edge features. The first step is to retrieve the appropriate pair of node features for each edge through the edge index. This is done through a non-static array index. This pair of node features is concatenated with the corresponding edge features in a temporary array. Subsequently, a fully-connected NN (of up to four layers in depth) is applied to compute the updated edge features. HLS pseudocode illustrating the main structure and pragmas of the edge block is shown in **Algorithm 1**.

Algorithm 1: Throughput-optimized edge block.

Input: node_attr[n_{nodes}][node_dim],
edge_attr[n_{edges}][edge_dim], edge_index[n_{edges}][2]
Output: edge_update[n_{edges}][edge_dim]
pipeline algorithm with factor RF ▷ HLS pragma
procedure PARTITION ARRAYS(node_attr, edge_attr, edge_update)
completely partition node_attr ▷ HLS pragma
completely partition edge_attr ▷ HLS pragma
completely partition edge_update ▷ HLS pragma
end procedure
procedure CREATE NN INPUT(node_attr, edge_attr, edge_index)
initialize phi_input[n_{edges}][edge_dim+2×node_dim]
completely partition phi_input ▷ HLS pragma
for $i \leftarrow 1, n_{\text{edges}}$ **do**
completely unroll loop ▷ HLS pragma
receiver_index ← edge_index[i][0]
sender_index ← edge_index[i][1]
receiver ← node_attr[receiver_index]
sender ← node_attr[sender_index]
edge ← edge_attr[i]
phi_input[i] ← ⟨receiver, sender, edge⟩
end for
return phi_input
end procedure
procedure COMPUTE EDGE UPDATE(phi_input)
for $i \leftarrow 1, n_{\text{edges}}$ **do**
completely unroll loop ▷ HLS pragma
edge_update[i] ← NN(phi_input[i])
end for
return edge_update
end procedure

The edge-aggregation block takes as input the updated edge feature matrix and the edge index matrix, and returns the aggregated updated edge features gathered along the receiver nodes. To ensure a greater design flexibility, three aggregation methods are supported: sum, average, and maximum. Summation is the simplest aggregation algorithm, and simply requires appropriately summing the corresponding edge features for each receiver node. Average requires an additional step of counting the number of edges connected to each receiver node, and subsequently dividing by this number. The division is implemented through a predefined look-up table. In maximum aggregation, we initially set the output matrix to the most negative value allowed by the datatype. Subsequently, we use a tournament method to find the maximum value. We note that maximum aggregation requires the longest latency

and the most resources. Therefore, summation and average are choices better suited for this FPGA implementation.

Finally, the node block receives the current node features and the aggregated updated edge features, concatenates them, processes the concatenated product with a NN and returns the updated node features. It does not require any non-static array access.

6.3. Resource-Optimized Design

In the resource-optimized design, pipelining is performed at both the task-level of the whole design and the level of each functional block, using the RF to determine each loop's II and a *parallelization factor* (PF) to determine the degree of loop-unrolling. The task-level pipelining allows every functional block to execute concurrently, reducing II of the total design. Within each functional block, the PF is used to adjust the number of unrolled loops. The resource-optimized design can handle large graph sizes by adjusting PF to fit FPGA resource capacity (higher PF means more unrolled loops and greater resource usage). However, the latency will increase in proportion to the graph size and violates the 4 μs latency constraint when the graph size is larger than about 896 nodes and 1,792 edges.

The resource-optimized edge block has the same inputs and outputs as the throughput-optimized edge block. **Algorithm 2** shows the pseudocode of this design, which adopts different explicit memory access methods to handle arrays that require static or non-static access. Static-access arrays (edge features, edge index, and updated edge features) are cyclically partitioned with a factor equal to the PF multiplied by the array's feature dimension (e.g., the edge index array's feature dimension is 2, the edge feature array's feature dimension is the length of the vector used to represent each edge). This way, all the parallel computing elements can access these arrays concurrently. The same practice with the node features array, which requires non-static access, would cause latency degradation due to stalling cycles when accessing data of different addresses. Therefore, the node features array is first copied into PF-many duplicates, and each parallel computing element then accesses node data from its dedicated duplicate.

The resource-optimized edge-aggregation block has the same inputs and outputs as the throughput-optimized edge-aggregation block. This block adopts different explicit memory access methods to handle arrays that require static or non-static access. Static-access arrays (edge index and updated edge features) are partitioned in the same way as those in the resource-optimized edge block. The aggregated edge features array, which requires non-static access, must be handled differently. Stalling cycles may arise if there is an attempt to concurrently aggregate two or more edges to the same receiver node. Therefore, this block first creates PF-many aggregated edge feature "duplicates," and each parallel computing element then aggregates to its dedicated duplicate. Then, these duplicates are themselves aggregated into the final aggregated edge features array. The corresponding HLS pseudocode is shown in **Algorithm 3**.

The resource-optimized node block also has the same inputs and outputs as the throughput-optimized node block. It does not require any non-static array access, so all arrays are partitioned in the same manner way that the edge block and edge-aggregation block partition the static-access arrays.

Finally, in the resource-optimized design, arrays which are used by more than one functional block, or `hls4ml` layer, are cloned in order to allow concurrent access. For example, it can be seen in **Figure 5** that the output of the first edge block is used in the edge-aggregation block, the node block, and the second edge block. Therefore, three clones are created from the output of this first edge block, one for each function that uses it. This way, no single array is ever used by more than one function. To perform this cloning, we developed an `hls4ml` cloning layer and a frontend `hls4ml` optimizer which searches a design for reused

Algorithm 2: Resource-optimized edge block.

Input: node_attr[n_{nodes}][node_dim],
edge_attr[n_{edges}][edge_dim], edge_index[n_{edges}][2]
Output: edge_update[n_{edges}][edge_dim]
procedure PARTITION ARRAYS(node_attr, edge_attr, edge_index, edge_update)
cyclically partition node_attr with factor node_dim×PF ▷ HLS pragma
cyclically partition edge_attr with factor edge_dim×PF ▷ HLS pragma
cyclically partition edge_index with factor 2×PF ▷ HLS pragma
cyclically partition edge_update with factor edge_dim×PF ▷ HLS pragma
end procedure
procedure COPY NODE ATTRIBUTES(node_attr)
initialize node_attr_copies[PF][n_{nodes}][node_dim]
for $i \leftarrow 1, n_{\text{nodes}}$ **do**
unroll loop with factor PF ▷ HLS pragma
pipeline loop with factor RF ▷ HLS pragma
for $j \leftarrow 1$, PF **do**
completely unroll loop ▷ HLS pragma
node_attr_copies[j][i] ← node_attr[i]
end for
end for
return node_attr_copies
end procedure
procedure COMPUTE EDGE UPDATE(edge_attr, node_attr_copies, edge_index)
for $i \leftarrow 1, n_{\text{edges}}$ **do**
unroll loop with factor PF ▷ HLS pragma
pipeline loop with factor RF ▷ HLS pragma
receiver_index ← edge_index[i][0]
sender_index ← edge_index[i][1]
receiver ← node_attr_copies[i%PF][receiver_index]
sender ← node_attr_copies[i%PF][sender_index]
edge ← edge_attr[i]
phi_input ← ⟨receiver, sender, edge⟩
edge_update[i] ← NN(phi_input)
end for
return edge_update
end procedure

Algorithm 3: Resource-optimized aggregation block.

Input: edge_update[n_{edges}][edge_dim], edge_index[n_{edges}][2]
Output: node_aggr[n_{nodes}][node_dim]

```
procedure PARTITION ARRAYS(edge_update, edge_index, node_aggr)
    cyclically partition edge_update with factor edge_dim×PF ▷ HLS pragma
    cyclically partition edge_index with factor 2×PF ▷ HLS pragma
    cyclically partition node_aggr with factor edge_dim×PF ▷ HLS pragma
end procedure
procedure RESET INTERMEDIATE NODE FEATURES(node_interim)
    initialize node_interim[PF][n_nodes][edge_dim]
    for i ← 1, n_nodes do
        unroll loop with factor PF ▷ HLS pragma
        pipeline loop with factor RF ▷ HLS pragma
        for j ← 1, PF do
            completely unroll loop ▷ HLS pragma
            node_interim[j][i] ← 0
        end for
    end for
    return node_interim
end procedure
procedure AGGREGATE EDGE UPDATE TO RECEIVER(edge_update, node_interim, edge_index)
    for i ← 1, n_edges do
        unroll loop with factor PF ▷ HLS pragma
        pipeline loop with factor RF ▷ HLS pragma
        receiver_index ← edge_index[i][0]
        node_interim[i%PF][receiver_index] ← node_interim[i%PF][receiver_index] + edge_update[i]
    end for
    return edge_update
end procedure
procedure SUM INTERMEDIATE NODE FEATURES(node_interim)
    for i ← 1, n_nodes do
        unroll loop with factor PF ▷ HLS pragma
        pipeline loop with factor RF ▷ HLS pragma
        for j ← 1, PF do
            completely unroll loop ▷ HLS pragma
            node_aggr[i] ← node_interim[j][i] + node_aggr[i]
        end for
    end for
    return node_aggr
end procedure
```

arrays and appropriately inserts instances of this cloning layer wherever necessary.

In summary, the main differences between the resource-optimized and throughput-optimized design are (1) pipelining both at the task-level and within each functional block and (2) duplicating arrays for parallel, concurrent access. Although (2) demands more storage space for data, the memory resources of the FPGA used are more than sufficient to support this design choice for all the graph sizes discussed here.

7. RESULTS

For the `hls4ml` implementation, we scan the fixed-point precision to determine its impact on the physics performance of the algorithm as well as the latency and resource usage on the FPGA. We evaluate the receiver operating characteristic (ROC) curve for the segment classifier, and use the area under the curve (AUC) as a performance metric. In particular, we first scan the number of integer bits `Y` when using the Vivado arbitrary precision data type `ap_fixed<X,Y>`, i.e., `X` total bits are used and `Y` bits are used for the integer part (including sign). From this, we determine that the minimum number of integer bits required to reach the full AUC is 7. Next, we scan the number of total bits, holding the number of integer bits fixed to 7. **Figure 7** shows the AUC as a function of the number of integer bits (left) and total bits (right). We see that with 12 total bits and 7 integer bits, we effectively reproduce the 32-bit floating point model.

With `hls4ml`, we employ post-training quantization (PTQ), meaning the model training does not take into account the expected reduced precision. The required bit width for full performance can be reduced further through techniques like *quantization-aware training* (QAT) (Coelho, 2019; Coelho et al., 2021; Hawks et al., 2021), in which the effects of reduced precision operations are accounted for during training. **Figure 7** (right) shows that with a QAT library called BREVITAS (Pappalardo, 2020), only 7 total bits are needed for full performance. Details of the QAT procedure can be found in the **Supplementary Material**.

Figure 7 also shows the effect of graph truncation and zero-padding on the edge classification performance. Graph zero-padding, in which null nodes and edges are appended to a graph with too few nodes or edges, does not usually affect the classification performance. With the exception of a few rare cases, it is possible to pad a graph such that the null nodes and edges form a completely disconnected graph from the original graph. In this case, no messages are passed between the original and null subgraphs, and results for the original subgraph are the same. Graph truncation, on the other hand, has a twofold effect on the IN's performance. The first effect is that truncated nodes and edges are no longer factored into the creation or passing of messages, which clearly affects the final output. The second effect is that truncated edges can no longer be classified by the IN. To account for this second effect in our performance metrics, we consider each truncated edge as classified as false. **Figure 7** demonstrates that net result of the above effects is a drop in the optimal AUC from 99.9% to about 95%.

All resource estimates are computed using Vivado HLS 2019.2 using logic synthesis targeting a Xilinx Virtex UltraScale+ VU9P FPGA with part number `xcvu9p-flga2104-2L-e`. The latency and II estimates are from C synthesis. For simplicity, when scanning the total bit width `X` in the following results, we use a fixed-point precision of `ap_fixed<X,X/2>`, i.e., `X/2`

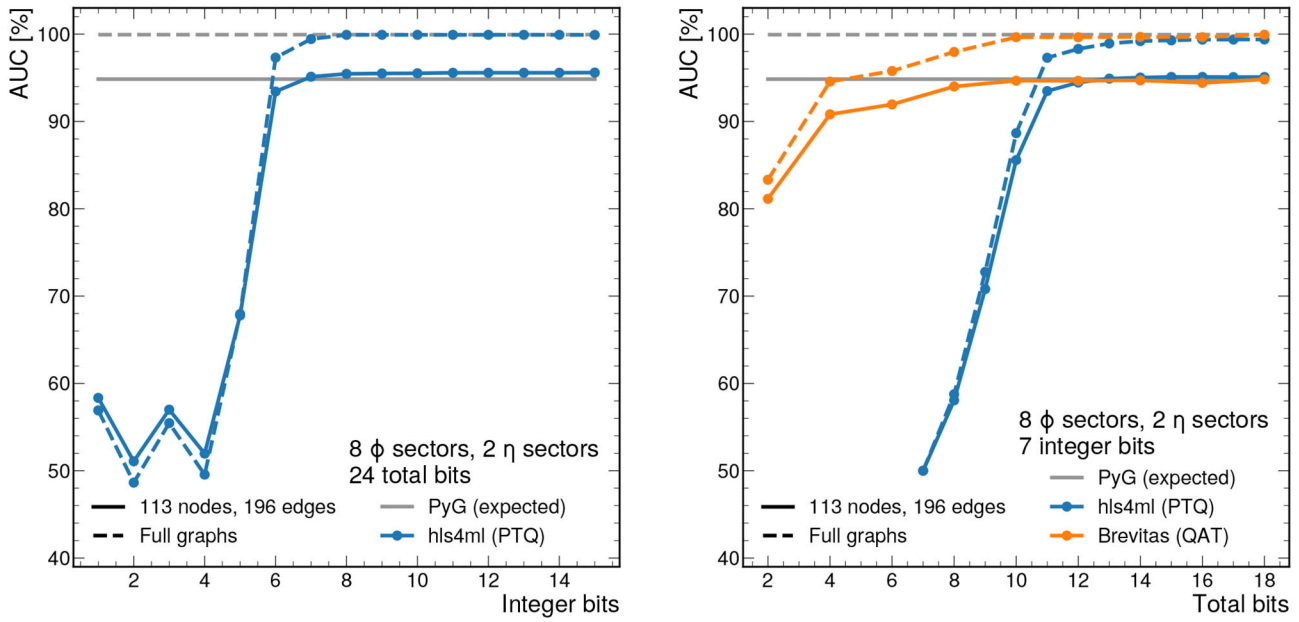

FIGURE 7 | AUC values as a function of the integer bit width with total bit width fixed to 24 **(left)** and total bit width with integer bit width fixed to 7 **(right)**. Either the full sectorized 2 GeV graphs (dashed line) or those truncated at 113 nodes and 196 edges (solid line), corresponding to the 95% percentile graph size, are used as input. The performance is evaluated with 1,000 graphs from `train_2`. With precision greater than `ap_fixed<12,7>`, the AUC closely approximates the full floating point model for the same graphs.

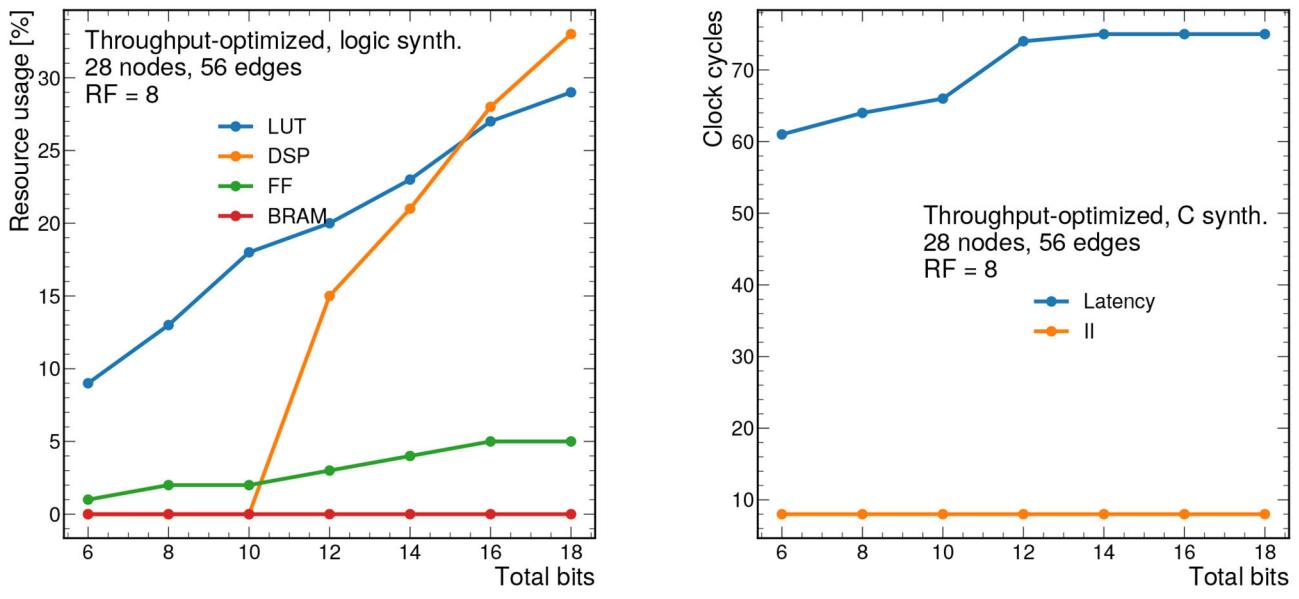

FIGURE 8 | Resource usage estimates as a percentage after logic synthesis **(left)** and latency and II in clock cycles **(right)** for a constant reuse factor of 8 as a function of the total bit width.

bits are used for each the integer and fractional parts, but the results generalize to other quantization schemes. The `hls4ml` version used is a custom branch available at Elabd et al. (2021).

7.1. Throughput-Optimized Results

First, we present results for the throughput-optimized implementation. We consider graphs consisting of 28 nodes and 56 edges, which is near the upper limit that can be synthesized with this design. **Figure 8** (left) shows the resource usage as a function of the bit width for a constant RF of 8. As expected, increasing the bit width increases the resource usage especially for LUTs and DSPs. As shown previously, this model has good performance with a total bit width of 12 bits, however the bit width can be reduced down to 7 bits using QAT (Coelho,

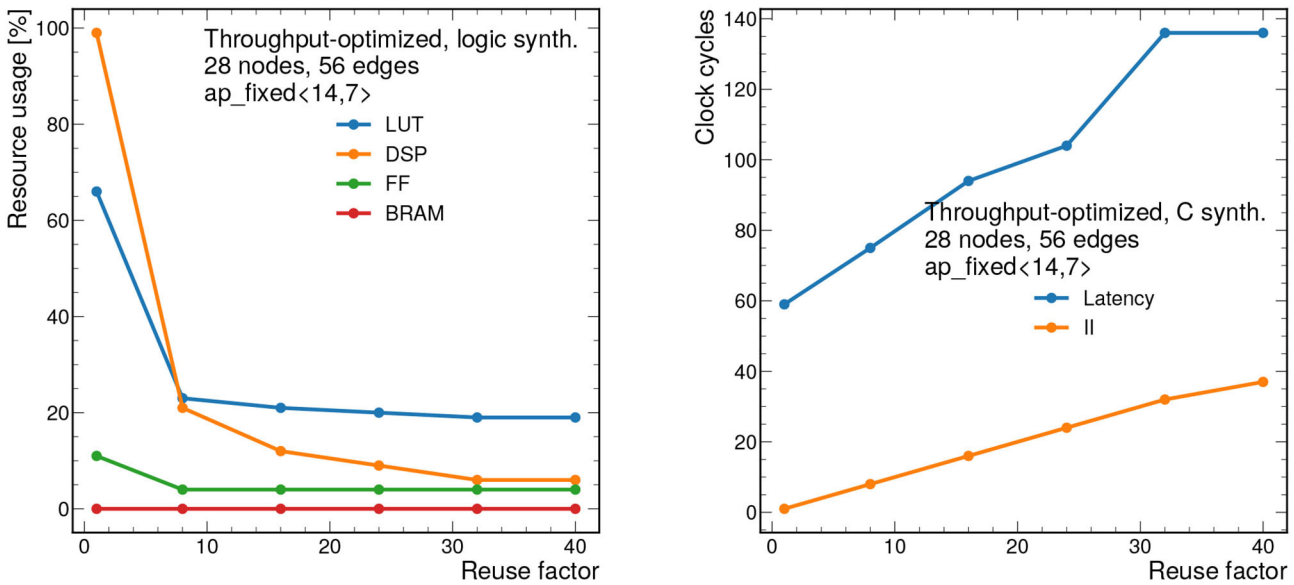

FIGURE 9 | Resource usage estimates as a percentage after logic synthesis **(left)** and latency and II in clock cycles **(right)** for a constant fixed point precision of `ap_fixed<14,7>` as a function of the reuse factor.

2019; Coelho et al., 2021; Hawks et al., 2021), meaning a substantion reduction in resource usage. We also note that Vivado HLS implements multiplications with bit widths of 10 and above using DSPs, and multiplications below 10 bits using LUTs (Xilinx, Inc., 2020). For this reason, we see that the DSP usage drops to zero for 10 bits and below.

Figure 8 (right) shows the latency in clock cycles (for a 5 ns clock period) as a function of the total bit precision, which ranges from about 300 to 370 ns. For simplicity, we consider `ap_fixed<X,X/2>` data types, so the number of integer bits is half of the total bits. By construction, the II for this design should be equal to the RF, although it may be smaller due to optimizations in Vivado HLS. In this case, the II is constant at 40 ns given the constant RF of 8.

We also scan the RF at a constant fixed point precision of `ap_fixed<14,7>`, to study the resources and timing as a function of decreasing concurrency. **Figure 9** shows the resource usage estimates (left) and latency and II (right) vs. RF. In general, increasing the RF, decreases the resources, while increasing the latency and II. For a RF of 1, the algorithm saturates the FPGA (100% DSP usage and 65% LUT usage). However, increasing the reuse factor to 8, makes the algorithm much more feasible, with the same resources taking up about 25% of the available ones. As we scan the RF from 1 to 40, we find the latency ranges from about 400 to 700 ns, while the II ranges from 5 to 200 ns.

Figure 10 also shows how the design scales as a function of the number of nodes n_{nodes} from 7 to 28. The number of edges n_{edges} is fixed to $n_{\text{edges}} = 2n_{\text{nodes}}$, a relationship that is empirically observed for the 2 GeV graphs. To synthesize larger graphs than those shown here, a different approach must be adopted.

7.2. Resource-Optimized Results

To demonstrate how the design scales, results for the resource-optimized implementation are shown for 448 nodes and 896 edges are shown in **Figures 11**, **12**. For all results shown, we take the PF= 16, but this is fully configurable by the user. Despite the large graph size, the resources remain below 90% for all bit widths and reuse factors considered. However, the latency ranges from about 2.4 μs to 40 μs depending on the reuse factor. Similarly the II ranges from about 850 ns to 11 μs. While these latency and II results may be too long for L1 applications, they represent substantial improvements over CPU-based processing and thus may form the basis of a CPU-FPGA coprocessing workflow for a particle tracking application in the high-level trigger or offline processing. For CPU-based inference of the same model (using a 12-core Intel Xeon CPU E5-2650 v4 @ 2.20 GHz), the latency is about 1.06 ms per graph for the same size graphs (448 nodes and 896 edges) in the PyG framework (Fey and Lenssen, 2019).

Figure 13 demonstrates the scalability of this resource-optimized design to larger graphs. We vary the number of nodes from 28 to 1,344, while, as before, we fix $n_{\text{edges}} = 2n_{\text{nodes}}$. Given the dataflow design, the resources stay fairly consistent throughout, always staying below 60% for the dominant resource (DSPs) when $RF = 1$. The latency and II scale linearly with the number of nodes.

Table 1 summarizes the latency, II, and FPGA resource usage metrics of the synthesized firmware, in particular comparing the throughput-optimized design with the resource-optimized one for a few different configuration choices. In particular, the table compares the throughput-optimized design for RF = 1 and RF = 8, noting the large reduction in resources (and relatively small increase in latency). We also show how the resources remain stable for the resource-optimized design even when scaling

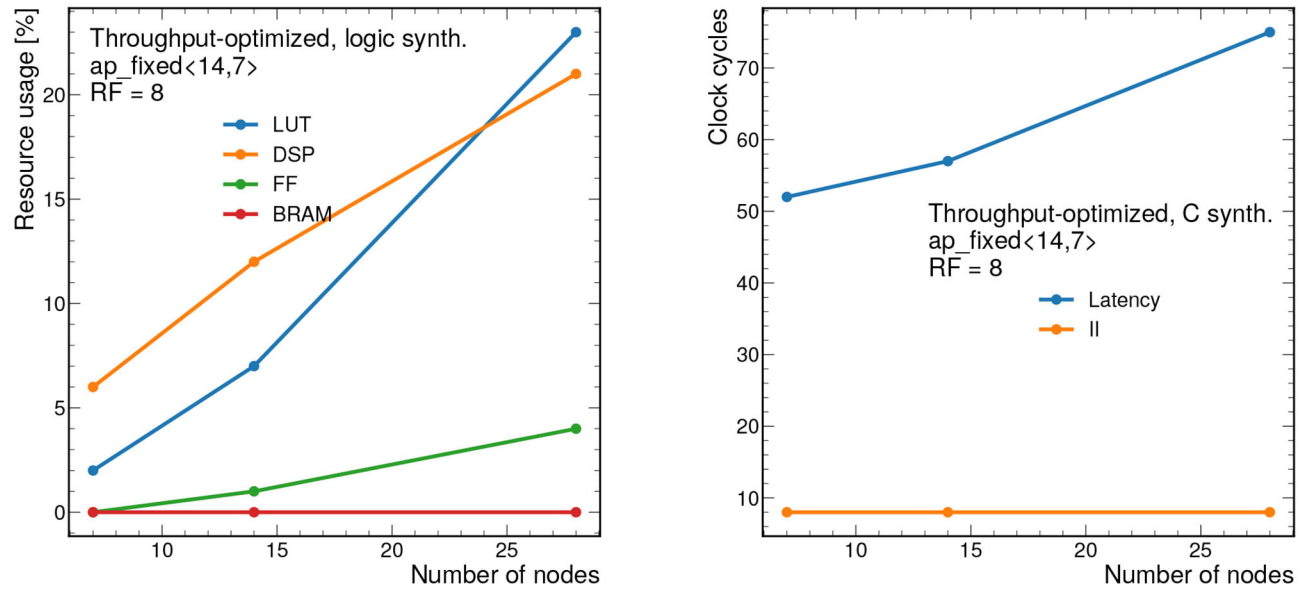

FIGURE 10 | Resource usage estimates as a percentage after logic synthesis **(left)** and latency and II in clock cycles **(right)** for a constant reuse factor of 8 and a bit width of `ap_fixed<14,7>` as a function of the number of nodes n_{nodes}. The number of edges is fixed to $n_{\text{edges}} = 2n_{\text{nodes}}$, as is empirically observed for the 2 GeV graphs. Each clock cycle corresponds to 5 ns.

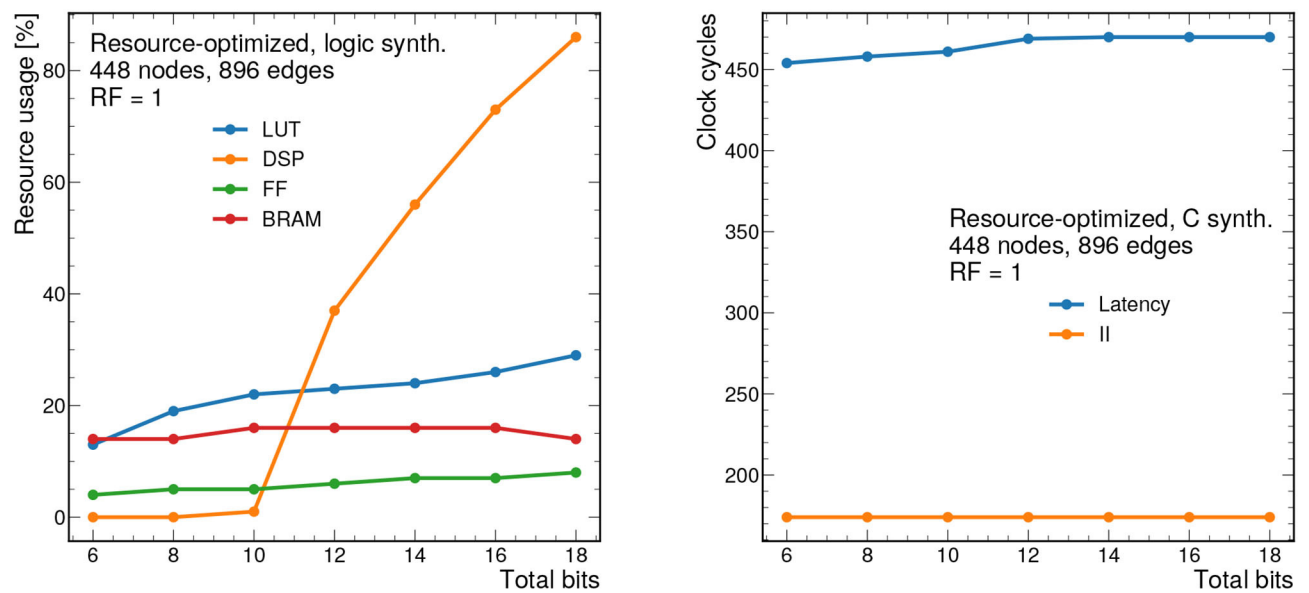

FIGURE 11 | Resource usage estimates as a percentage after logic synthesis **(left)** and latency and II in clock cycles **(right)** for a constant reuse factor of 1 as a function of the total bit width for the resource-optimized implementation. Input graphs consist of 448 nodes and 896 edges. Each clock cycle corresponds to 5 ns.

to much larger graph sizes. Finally, relative to the resource-optimized design, the throughput-optimized design is able to achieve the smallest latency and II for small graphs.

8. SUMMARY AND OUTLOOK

In summary, we develop two complementary field-programmable gate array (FPGA) implementations of a graph neural network (GNN) for charged particle tracking at the LHC. Namely, the GNN classifies track segments as true or false based on a graph constructed from the positions of hits in the tracking detector. One of the implementations is optimized for low-latency and high-throughput typical for applications in the FPGA-based level-1 trigger systems at the LHC. The other implementation is optimized to minimize the FPGA resources needed and is capable of scaling to much larger graph sizes

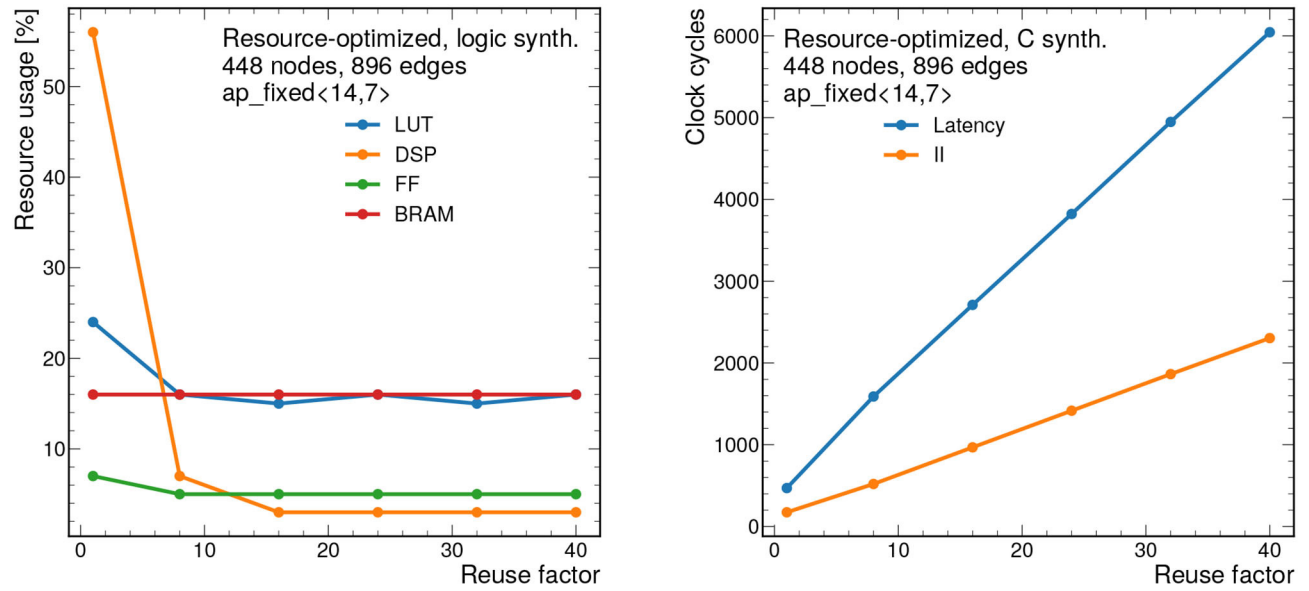

FIGURE 12 | Resource usage estimates as a percentage after logic synthesis **(left)** and latency and II in clock cycles **(right)** for a constant fixed point precision of `ap_fixed<14,7>` as a function of the reuse factor for the resource-optimized implementation. Input graphs consist of 448 nodes and 896 edges. Each clock cycle corresponds to 5 ns.

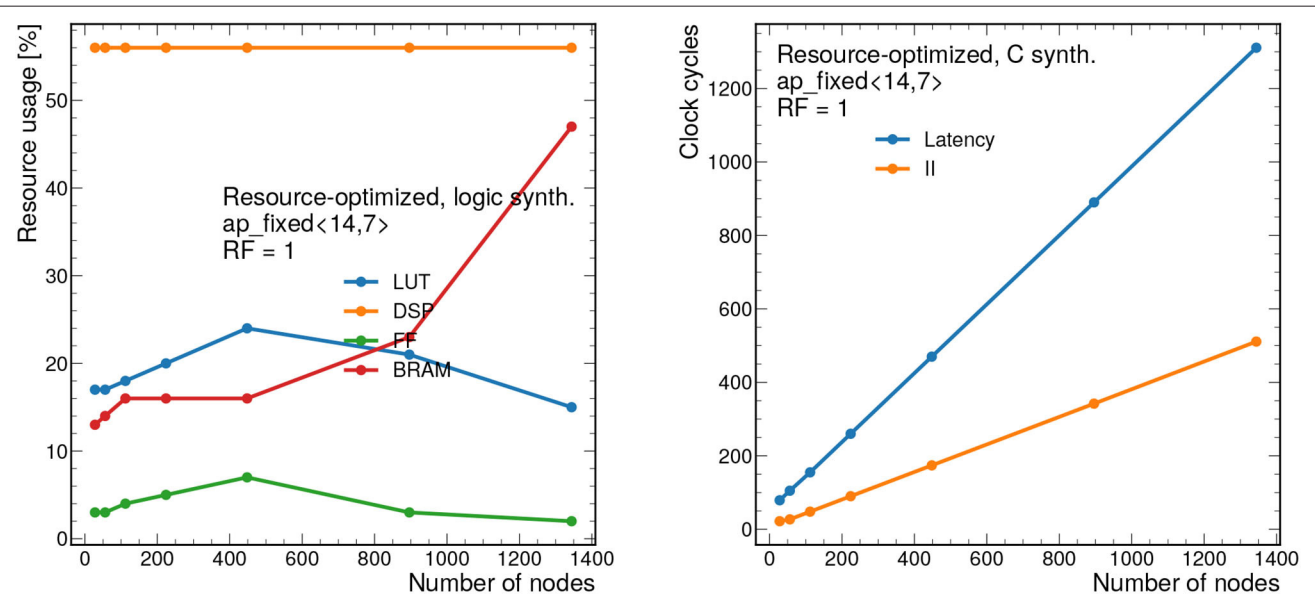

FIGURE 13 | Resource usage estimates as a percentage after logic synthesis **(left)** and latency and II in clock cycles **(right)** for a constant reuse factor of 1 and a bit width of `ap_fixed<14,7>` as a function of the number of nodes n_{nodes}. The number of edges is fixed to $n_{\text{edges}} = 2n_{\text{nodes}}$, as is empirically observed for the 2 GeV graphs. Each clock cycle corresponds to 5 ns.

(thousands of nodes and edges). In order to make this possible, multiple improvements were made including an optimization of the memory access of the input data and the instantiation of multiple parallel processing engines. This implementation is applicable for FPGA-CPU coprocessing workflows in both the software-based high-level trigger and offline computing. The conversion of the trained model, specified using PyTorch Geometric, into high-level synthesis (HLS) code is achieved automatically using a custom converter integrated into `hls4ml`, a source-to-source compiler.

TABLE 1 | Summary of the latency, II, and FPGA resource usage metrics of the synthesized firmware for a variety of design choices.

Design	(n_{nodes}, n_{edges})	RF	Precision	Latency [cycles]	II [cycles]	DSP [%]	LUT [%]	FF [%]	BRAM [%]
Throughput-opt.	(28, 56)	1	`ap_fixed<14,7>`	59	1	99.9	66.0	11.7	0.7
Throughput-opt.	(28, 56)	8	`ap_fixed<14,7>`	75	8	21.9	23.8	4.7	0.7
Resource-opt.	(28, 56)	1	`ap_fixed<14,7>`	79	28	56.6	17.6	3.9	13.1
Resource-opt.	(448, 896)	1	`ap_fixed<14,7>`	470	174	56.6	25.0	7.4	16.5
Resource-opt.	(448, 896)	8	`ap_fixed<14,7>`	1590	520	5.6	25.0	7.4	16.3

The target FPGA is a Xilinx Virtex UltraScale+ VU9P FPGA (part number xcvu9p-flga2104-2L-e), which has 6,840 DSPs, 1,182,240 LUTs, 2,364,480 FFs, and 75.9 Mb of BRAM (Xilinx, Inc., 2021). A 5 ns clock period is used.

Several further improvements can reduce the resource usage or improve the performance. In particular, we showed quantization-aware training (QAT) (Coelho, 2019; Coelho et al., 2021; Hawks et al., 2021) can significantly reduce the number of bits required, but further work is needed to fully support QAT GNN models into `hls4ml`. A further detector-motivated optimization is the restriction of which nodes can be accessed simultaneously. Currently, the firmware is generic enough to allow any two nodes in the input graph to be connected, but given the detector geometry and hitgraph construction, certain nodes will never be directly connected (such as those belonging to the same tracker layer or on opposite sides of the detector). Implementing this restriction should further reduce the FPGA resources required.

Other considerations are necessary for implementing such a trigger in real experimental settings. For instance, graph construction, track building, and track fitting are additional steps that need to be applied in addition to the track segment classification performed here. Further, the sectors we use are non-overlapping, which means boundary effects can reduce the accuracy of the track segment classification. To resolve this, a typical technique is to use overlapping regions and define "fiducial regions" as the central portions of each region, where boundary effects are less significant. Further duplicate removal may be necessary as well when building full track candidates.

Despite these caveats and further possible improvements, this study demonstrates that for small graphs with dozens of nodes and edges, inference of GNNs for charged particle tracking is possible within the strict sub-microsecond latency and FPGA resource requirements of the level-1 trigger at the LHC. Further, for CPU-FPGA coprocessing applications, larger graphs with thousands of nodes and edges can also be processed with latencies in the tens of microseconds range, which still represent a considerable speedup with respect to CPU-only inference.

DATA AVAILABILITY STATEMENT

Publicly available datasets were analyzed in this study. This data can be found here: https://kaggle.com/c/trackml-particle-identification.

AUTHOR CONTRIBUTIONS

AE and VR developed the `hls4ml` implementation of the neural networks, especially the throughput-optimized design, with input and supervision from JD, MN, MA, ST, GD, IO, and PE. S-YH improved and developed the resource optimized HLS implementation with input and supervision from B-CL, J-XH, and S-CH. MT developed and trained the quantization-aware models with input and supervision from S-YH, SH, and JD. All authors contributed to the article and approved the submitted version.

FUNDING

This work was supported by IRIS-HEP through the U.S. National Science Foundation (NSF) under Cooperative Agreement OAC-1836650. JD was supported by the U.S. Department of Energy (DOE), Office of Science, Office of High Energy Physics Early Career Research program under Award No. DE-SC0021187. GD was supported by DOE Award No. DE-SC0007968. B-CL was supported by the Taiwan Ministry of Science and Technology under MOST 110-2224-E-A49-004. S-CH and SH were supported by NSF Award No. OAC-1934360. MA and MN were supported by NSF Award No. OAC-1934757.

ACKNOWLEDGMENTS

We gratefully acknowledge the input and discussion from the Exa.TrkX collaboration. We also acknowledge the Fast Machine Learning collective as an open community of multi-domain experts and collaborators. This community was important for the development of this project. In particular, Vladimir Loncar, Sioni Summers, Duc Hoang, Yutaro Iiyama, Maurizio Pierini, Nhan Tran, Philip Harris, Mia Liu, Sofia Vallecorsa, and Kazi Ahmed Asif Fuad provided valuable input for the `hls4ml` implementation of the graph neural network.

SUPPLEMENTARY MATERIAL

The Supplementary Material for this article can be found online at: https://www.frontiersin.org/articles/10.3389/fdata.2022.828666/full#supplementary-material

REFERENCES

Aaboud, M., et al. (2017). Performance of the ATLAS track reconstruction algorithms in dense environments in LHC Run 2. *Eur. Phys. J. C* 77, 673. doi: 10.1140/epjc/s10052-017-5225-7

Aad, G., et al. (2020). Operation of the ATLAS trigger system in Run 2. *J. Instrum*. 15, P10004. doi: 10.1088/1748-0221/15/10/P10004

Aarrestad, T., Loncar, V., Ghielmetti, N., Pierini, M., Summers, S., Ngadiuba, J., et al. (2021). Fast convolutional neural networks on FPGAs with hls4ml. *Mach. Learn. Sci. Tech*. 2, 045015. doi: 10.1088/2632-2153/ac0ea1

Amrouche, S., Basara, L., Calafiura, P., Estrade, V., Farrell, S., Ferreira, D. R., et al. (2020). "The tracking machine learning challenge: accuracy phase," in *The NeurIPS '18 Competition*, eds S. Escalera and R. Herbrich (Cham: Springer), 231. doi: 10.1007/978-3-030-29135-8_9

Arjona Martínez, J., Cerri, O., Pierini, M., Spiropulu, M., and Vlimant, J.-R. (2019). Pileup mitigation at the Large Hadron Collider with graph neural networks. *Eur. Phys. J.* 134, 333. doi: 10.1140/epjp/i,2019-12710-3

ATLAS Collaboration (2017). *Technical Design Report for the Phase-II Upgrade of the ATLAS TDAQ System*. ATLAS Technical Design Report CERN-LHCC-2017-020. Available online at: https://cds.cern.ch/record/2285584

Auten, A., Tomei, M., and Kumar, R. (2020). "Hardware acceleration of graph neural networks," in *2020 57th ACM/IEEE Design Automation Conference (DAC)* (New York, NY: IEEE). doi: 10.1109/DAC18072.2020.9218751

Battaglia, P., Hamrick, J., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V., Malinowski, M., et al. (2018). Relational inductive biases, deep learning, and graph networks. *arXiv preprint arXiv:1806.01261*.

Battaglia, P. W., Pascanu, R., Lai, M., Rezende, D. J., and Kavukcuoglu, K. (2016). "Interaction networks for learning about objects, relations and physics," in *Advances in Neural Information Processing Systems*, eds D. Lee, M. Sugiyama, U. Luxburg, I. Guyon, and R. Garnett (Red Hook, NY: Curran Associates), 4502.

Besta, M., Stanojevic, D., De Fine Licht, J., Ben-Nun, T., and Hoefler, T. (2019). Graph processing on FPGAs: taxonomy, survey, challenges. *arXiv[Preprint]. arXiv:1903.06697*.

Billoir, P. (1989). Progressive track recognition with a Kalman-like fitting procedure. *Comput. Phys. Commun*. 57, 390. doi: 10.1016/0010-4655(89)90249-X

Billoir, P., and Qian, S. (1990). Simultaneous pattern recognition and track fitting by the Kalman filtering method. *Nucleic Instrum. Methods Phys. Res. A* 294, 219. doi: 10.1016/0168-9002(90)91835-Y

Chatrchyan, S., et al. (2014). Description and performance of track and primary-vertex reconstruction with the CMS tracker. *J. Instrum*. 9, P10009. doi: 10.1088/1748-0221/9/10/P10009

CMS Collaboration (2020). *The Phase-2 Upgrade of the CMS Level-1 Trigger*. CMS Technical Design Report https://cds.cern.ch/record/2714892https://cds.cern.ch/record/2714892

Coelho, C. (2019). *QKeras*.

Coelho, C. N., Kuusela, A., Li, S., Zhuang, H., Ngadiuba, J., Aarrestad, T. K., et al. (2021). Automatic heterogeneous quantization of deep neural networksfor low-latency inference on the edge for particle detectors. *Nat. Mach. Intell.* 3:675 doi: 10.1038/s42256-021-00356-5

Dennard, R. H., Gaensslen, F. H., Yu, H., Rideout, V. L., Bassous, E., and LeBlanc, A. R. (1974). Design of ion-implanted MOSFET's with very small physical dimensions. *IEEE J. Solid State Circ.* 9, 256. doi: 10.1109/JSSC.1974.1050511

DeZoort, G., Thais, S., Ojalvo, I., Elmer, P., Razavimaleki, V., Duarte, J., et al. (2021). Charged particle tracking via edge-classifying interaction networks. *Comput. Softw. Big Sci*. 5, 26. doi: 10.1007/s41781-021-00073-z

Duarte, J., Han, S., Harris, P., Jindariani, S., Kreinar, E., Kreis, B., et al. (2018). Fast inference of deep neural networks in FPGAs for particle physics. *J. Instrum*. 13, P07027. doi: 10.1088/1748-0221/13/07/P07027

Duarte, J., and Vlimant, J.-R. (2020). "Graph neural networks for particle tracking and reconstruction," in *Artificial Intelligence for High Energy Physics*, eds P. Calafiura, D. Rousseau, and K. Terao (World Scientific Publishing). 387. doi: 10.1142/97898112340330012

Elabd, A., Razavimaleki, V., Huang, S. -Y., Duarte, J., Atkinson, M., DeZoort, G., et al. (2021). abdelabd/hls4ml: v0.6.0-pyg. (Geneva: Zenodo). Available online at: https://arxiv.org/pdf/2112.02048.pdf (accessed January 26, 2022).

Esmaeilzadeh, H., Blem, E., St. Amant, R., Sankaralingam, K., and Burger, D. (2011). "Dark silicon and the end of multicore scaling," in *Proceedings of the 38th Annual International Symposium on Computer Architecture* (New York, NY: ACM), 365. doi: 10.1145/2024723.2000108

Farrell, S., Calafiura, P., Mudigonda, M., Prabhat, Anderson, D., Vlimant, J-R., et al. (2018). "Novel deep learning methods for track reconstruction," in *4th International Workshop Connecting The Dots 2018* (Seattle, WA).

Fey, M., and Lenssen, J. E. (2019). "Fast graph representation learning with PyTorch Geometric," in *ICLR Workshop on Representation Learning on Graphs and Manifolds* (New Orleans, LA).

Frühwirth, R. (1987). Application of Kalman filtering to track and vertex fitting. *Nucleic Instrum. Methods Phys. Res. A* 262, 444. doi: 10.1016/0168-9002(87)90887-4

Geng, T., Li, A., Shi, R., Wu, C., Wang, T., Li, Y., et al. (2020). "AWB-GCN: a graph convolutional network accelerator with runtime workload rebalancing," in *53rd IEEE/ACM International Symposium on Microarchitecture* (New York, NY: IEEE). doi: 10.1109/MICRO50266.2020.00079

Glorot, X., Bordes, A., and Bengio, Y. (2011). "Deep sparse rectifier neural networks," in *Proceedings of the 14th International Conference on Artificial Intelligence and Statistics*, eds G. Gordon, D. Dunson, and M. Dudík (Fort Lauderdale, FL: JMLR), 315.

Govorkova, E., Puljak, E., Aarrestad, T., James, T., Loncar, V., Pierini, M., et al. (2021). Autoencoders on FPGAs for real-time, unsupervised new physics detection at 40 MHz at the Large Hadron Collider. *arXiv[Preprint]. arXiv:2108.03986*. Available online at: https://arxiv.org/pdf/2108.03986.pdf (accessed August 12, 2021)

Gui, C.-Y., Zheng, L., He, B., Liu, C., Chen, X.-Y., Liao, X.-F., et al. (2019). A survey on graph processing accelerators: challenges and opportunities. *J. Comput. Sci. Technol*. 34, 339. doi: 10.1007/s11390-019-1914-z

Harris, C. R., Millman, K. J., van der Walt, S. J., Gommers, R., Virtanen, P., Cournapeau, D., et al. (2020). Array programming with NumPy. *Nature* 585, 357. doi: 10.1038/s41586-020-2649-2

Hawks, B., Duarte, J., Fraser, N. J., Pappalardo, A., Tran, N., and Umuroglu, Y. (2021). Ps and Qs: quantization-aware pruning for efficient low latency neural network inference. *Front. AI* 4, 94. doi: 10.3389/frai.2021.676564

Heintz, A., Razavimaleki, V., Duarte, J., DeZoort, G., Ojalvo, I., Thais, S., et al. (2020). "Accelerated charged particle tracking with graph neural networks on FPGAs," in *3rd Machine Learning and the Physical Sciences Workshop at the 34th Conference on Neural Information Processing Systems* (Vancouver, BC).

Iiyama, Y., Cerminara, G., Gupta, A., Kiesler, J., Loncar, V., Pierini, M., et al. (2021). Distance-weighted graph neural networks on FPGAs for real-time particle reconstruction in high energy physics. *Front. Big Data* 3, 44. doi: 10.3389/fdata.2020.598927

Ju, X., Farrell, S., Calafiura, P., Murnane, D., Prabhat, Gray. L., et al. (2019). "Graph neural networks for particle reconstruction in high energy physics detectors," in *Machine Learning and the Physical Sciences Workshop at the 33rd Annual Conference on Neural Information Processing Systems* (Vancouver, BC).

Ju, X., Murnane, D., Calafiura, P., Farrell, S., Xu, Y., Spiropulu, M., et al. (2021). Performance of a geometric deep learning pipeline for HL-LHC particle tracking. *Eur. Phys. J. C* 81, 876. doi: 10.1140/epjc/s10052-021-09675-8

Kieseler, J. (2020). Object condensation: one-stage grid-free multi-object reconstruction in physics detectors, graph and image data. *Eur. Phys. J. C* 80, 886. doi: 10.1140/epjc/s10052-020-08461-2

Kingma, D. P., and Ba, J. (2015). "Adam: a method for stochastic optimization," in *3rd International Conference on Learning Representations*, eds Y. Bengio and Y. LeCun (San Diego, CA).

Kiningham, K., Re, C., and Levis, P. (2020). GRIP: a graph neural network accelerator architecture. *arXiv[Preprint]. arXiv:2007.13828*.

Li, T., Liu, S., Feng, Y., Tran, N., Liu, M., and Li, P. (2021). "Semi-supervised graph neural network for particle-level noise removal," in *NeurIPS 2021 AI for Science Workshop* (Vancouver, BC).

Loncar, V., Summers, S., Duarte, J., Tran, N., Kreis, B., Ngadiuba, J., et al. (2020). Compressing deep neural networks on FPGAs to binary and ternary precision with *hls4ml. Mach. Learn. Sci. Technol*. 2020, 015001. doi: 10.1088/2632-2153/aba042

Loncar, V., Summers, S., Duarte, J., Tran, N., Kreis, B., Ngadiuba, J., et al. (2021). fastmachinelearning/hls4ml: coris (v0.6.0). (Geneva: Zenodo). doi: 10.5281/zenodo.5680908

Mankel, R. (1997). A concurrent track evolution algorithm for pattern recognition in the hera-b main tracking system. *Nucleic Instrum. Methods Phys. Res. A* 395, 169. doi: 10.1016/S0168-9002(97)00705-5

Ming Xiong, Z. (2020). "A survey of FPGA based on graph convolutional neural network accelerator," in *2020 International Conference on Computer Engineering and Intelligent Control (ICCEIC)* (New York, NY, USA: IEEE), 92. doi: 10.1109/ICCEIC51584.2020.00026

Moreno, E. A., Cerri, O., Duarte, J. M., Newman, H. B., Nguyen, T. Q., Periwal, A., et al. (2020a). JEDI-net: a jet identification algorithm based on interaction networks. *Eur. Phys. J. C* 80, 58. doi: 10.1140/epjc/s10052-020-7608-4

Moreno, E. A., Nguyen, T. Q., Vlimant, J.-R., Cerri, O., Newman, H. B., Periwal, A., et al. (2020b). Interaction networks for the identification of boosted $H \rightarrow b\bar{b}$ decays. *Phys. Rev. D* 102, 012010. doi: 10.1103/PhysRevD.102.012010

Nair, V., and Hinton, G. E. (2010). "Rectified linear units improve restricted Boltzmann machines," in *Proceedings of the 27th International Conference on International Conference on Machine Learning, ICML'10* (Madison, WI: Omnipress), 807.

Numan, M. W., Phillips, B. J., Puddy, G. S., and Falkner, K. (2020). Towards automatic high-level code deployment on reconfigurable platforms: a survey of high-level synthesis tools and toolchains. *IEEE Access* 8, 174692. doi: 10.1109/ACCESS.2020.3024098

Nurvitadhi, E., Weisz, G., Wang, Y., Hurkat, S., Nguyen, M., Hoe, J. C., et al. (2014). "GraphGen: an FPGA framework for vertex-centric graph computation," in *2014 IEEE 22nd Annual International Symposium on Field-Programmable Custom Computing Machines* (New York, NY, USA: IEEE), 25. doi: 10.1109/FCCM.2014.15

Ozdal, M. M., Yesil, S., Kim, T., Ayupov, A., Greth, J., Burns, S., et al. (2016). Energy efficient architecture for graph analytics accelerators. *Comput. Arch. News* 44, 166. doi: 10.1145/3007787.3001155

Pappalardo, A. (2020). brevitas.

Pata, J., Duarte, J., Vlimant, J.-R., Pierini, M., and Spiropulu, M. (2021). MLPF: Efficient machine-learned particle-flow reconstruction using graph neural networks. *Eur. Phys. J. C* 81, 381. doi: 10.1140/epjc/s10052-021-09158-w

Qasim, S. R., Kieseler, J., Iiyama, Y., and Pierini, M. (2019). Learning representations of irregular particle-detector geometry with distance-weighted graph networks. *Eur. Phys. J. C* 79, 608. doi: 10.1140/epjc/s10052-019-7113-9

Qu, H., and Gouskos, L. (2020). ParticleNet: jet tagging via particle clouds. *Phys. Rev. D* 101, 056019. doi: 10.1103/PhysRevD.101.056019

Shlomi, J., Battaglia, P., and Vlimant, J.-R. (2020). Graph neural networks in particle physics. *Mach. Learn. Sci. Tech.* 2, 021001. doi: 10.1088/2632-2153/abbf9a

Sirunyan, A. M., Adam, W., Ambrogi, F., Arnold, B., Bergauer, H., Bergayer, T., et al. (2020). Performance of the CMS Level-1 trigger in proton-proton collisions at $\sqrt{s} = 13$ TeV. *J. Instrum.* 15, P10017. doi: 10.1088/1748-0221/15/10/P10017 Available online at: https://arxiv.org/pdf/2006.10165.pdf (accessed October 19, 2020).

Strandlie, A., and Frühwirth, R. (2010). Track and vertex reconstruction: from classical to adaptive methods. *Rev. Mod. Phys.* 82, 1419. doi: 10.1103/RevModPhys.82.1419

Summers, S., et al. (2020). Fast inference of boosted decision trees in FPGAs for particle physics. *J. Instrum.* 15, P05026. doi: 10.1088/1748-0221/15/05/P05026

Trocino, D. (2014). The CMS high level trigger. *J. Phys. Conf. Ser.* 513, 012036. doi: 10.1088/1742-6596/513/1/012036

Xilinx, Inc. (2020). *Vivado Design Suite User Guide: High Level Synthesis.*

Xilinx, Inc. (2021). *UltraScale+ FPGAs Product Tables and Product Selection Guide.*

Yan, M., Deng, L., Hu, X., Liang, L., Feng, Y., Ye, X., et al. (2020). "33HyGCN: a GCN accelerator with hybrid architecture," in *2020 IEEE International Symposium on High Performance Computer Architecture (HPCA)* (New York, NY: IEEE), 15. doi: 10.1109/HPCA47549.2020.00012

Zeng, H., and Prasanna, V. (2020). "GraphACT: accelerating GCN training on CPU-FPGA heterogeneous platforms," in *2020 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* (New York, NY: ACM), 255. doi: 10.1145/3373087.3375312

Conflict of Interest: The authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Publisher's Note: All claims expressed in this article are solely those of the authors and do not necessarily represent those of their affiliated organizations, or those of the publisher, the editors and the reviewers. Any product that may be evaluated in this article, or claim that may be made by its manufacturer, is not guaranteed or endorsed by the publisher.