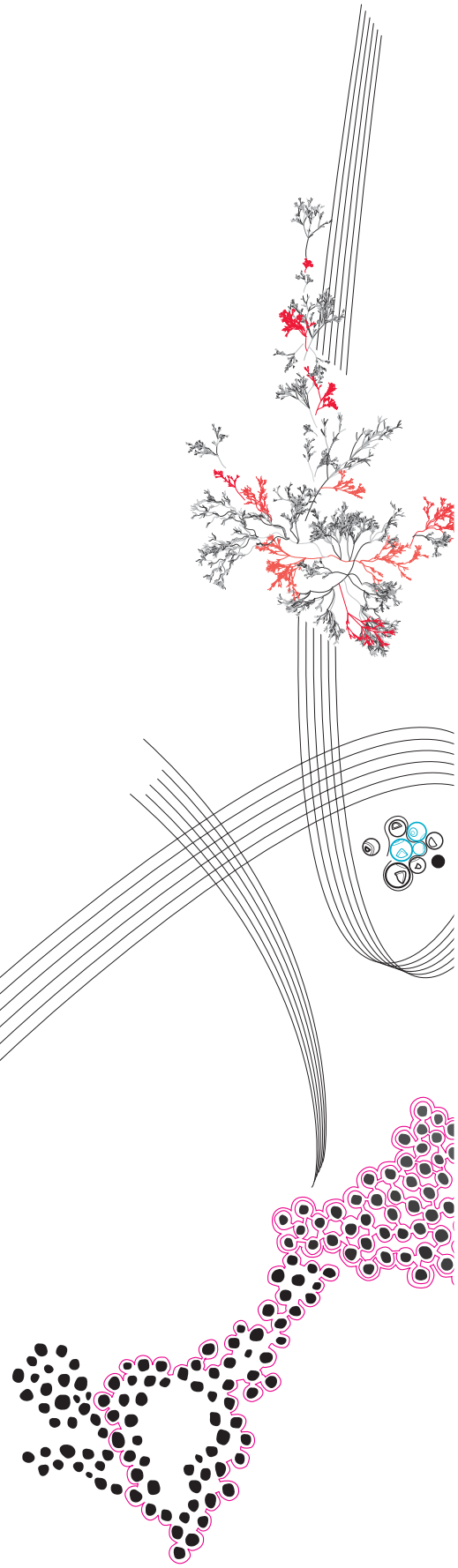


MSc Embedded Systems
Final Thesis

Automated Transformer Deployment on FPGA for Particle Tracking

Arjan Blankestijn

Supervisors:

Prof.dr.ir. A.L. Varbanescu

Dr.ir. U. Odyurt

Dr.ir. A. Yousefzadeh

External examiner: Dr.ir. L.J. Spreeuwers

April, 2026

Faculty of Electrical Engineering,
Mathematics and Computer Science,
University of Twente

UNIVERSITY OF TWENTE.

Contents

Abstract

The High-Luminosity Large Hadron Collider (HL-LHC) will significantly increase data processing requirements for tasks such as particle tracking. Transformer-based models have recently shown promising results for track reconstruction, but running such models in hardware environments relevant to high-energy physics is far from straightforward. Strict latency constraints, limited on-chip memory, and integration into existing systems all have to be considered.

This thesis explores how a Transformer-based tracking model can be deployed on an FPGA platform under realistic resource and accuracy constraints. Instead of aiming for a full end-to-end implementation, the focus is placed on accelerating a representative Transformer encoder layer and studying the practical trade-offs that arise in hardware. An FPGA accelerator prototype is developed using high-level synthesis, with particular attention to on-chip weight storage, memory organization, and various hardware optimizations.

Beyond the hardware design itself, this work also addresses the practical challenge that deploying machine learning models on FPGAs typically requires expertise in both deep learning and hardware design. To lower this barrier, a semi-automated workflow is developed that enables selective offloading Transformer model parts to the FPGA, while the remaining parts of the model execute on a general-purpose processor. The goal is not only to build an accelerator, but also to make experimentation with FPGA-based acceleration more accessible to researchers without extensive hardware experience.

Experimental results show that the design is largely memory-bound, with on-chip memory usage forming the main bottleneck. Multi-layer implementations improve throughput and energy efficiency, but the achieved latency remains higher than that of modern GPU solutions. Overall, the results demonstrate that selective acceleration of Transformer components on FPGA hardware is feasible and provide a practical, reproducible workflow that helps bridge the gap between machine learning models and FPGA deployment in a high-energy physics context.

Chapter 1

Introduction

1.1 Data Processing at the HL-LHC

High-Energy Physics (HEP) experiments, such as those conducted at the Large Hadron Collider (LHC) at CERN, study the products of high-energy particle beam collisions. When particles produced in these collisions pass through the detector systems, they leave signals in the detector elements. These signals are recorded as spatial measurements, commonly referred to as hits. From these hits, the trajectories and properties of the particles are reconstructed through subsequent data processing.

With the upcoming High-Luminosity Large Hadron Collider (HL-LHC) upgrade [8], the collision rate will increase significantly. In collider experiments, the detector data associated with a single bunch crossing is referred to as an event, which may contain collisions due to pile-up. As event complexity grows, the amount of detector data that must be processed increases substantially. This places additional pressure on the entire data-processing chain of experiments such as ATLAS [37], from real-time triggering systems to offline reconstruction and analysis. Consequently, efficiently processing this data with low latency has become a key challenge.

A central task in this process is particle tracking, which involves reconstructing particle trajectories from a large set of detector hits. Traditionally, this tracking has relied on algorithms such as the Kalman filter [19]. While these methods are well understood and have been successfully used in current experiments, their computational cost increases rapidly in high-occupancy environments such as those expected at the HL-LHC. This makes it increasingly challenging to process events fast enough, motivating the exploration of more efficient approaches.

1.2 Machine Learning for Particle Tracking

Machine Learning (ML) models have shown strong performance across various fields and are beginning to influence the HEP community. ML methods are currently used for tasks such as jet tagging and event classification, and are increasingly explored as alternatives or complements to traditional tracking algorithms. Once trained, these models can be applied to detector data to produce predictions, such as assigning hits to particle tracks. This stage, where a trained model processes new input data to generate outputs, is commonly referred to as inference.

Recently, Transformer architectures have gained popularity due to their ability to model complex relationships, handle long-range dependencies, and effectively scale to large inputs

[28]. These features make them particularly well-suited for tracking tasks, where numerous correlated hits must be processed together [23, 40].

1.3 Hardware Acceleration and FPGAs

To run such models efficiently, specialized hardware accelerators are often used. Typically, this involves employing GPUs, which provide high throughput and flexible programming environments. However, in situations where low latency, deterministic behavior, and energy efficiency are essential—such as in hardware trigger systems in HEP—field-programmable gate arrays (FPGAs) are crucial. FPGAs serve as the foundation for many trigger systems [7, 34, 12], which require decisions to be made within a few microseconds.

Integrating ML-based tracking, particularly Transformer-based models, into FPGAs could enhance intelligent processing near the detector and potentially improve trigger performance and expand the physics reach.

1.4 Challenges of Transformer Deployment on FPGAs

Deploying large models such as Transformers on FPGAs is challenging. Transformer layers require significant memory and computing resources, and components such as multi-head attention are difficult to map efficiently within tight resource constraints. These constraints coupled with memory requirements pose significant architectural challenges.

Furthermore, existing FPGA machine-learning tool-flows are often optimized for convolutional neural networks (CNNs) or relatively small fully connected networks, providing limited direct support for complex Transformer architectures. Although research on accelerating Transformers on FPGAs is growing, most work focuses on natural language processing or computer vision tasks [43, 30, 24], with limited guidance for particle-tracking models and high-energy physics applications.

In addition to these technical challenges, there is also a significant expertise barrier. Successfully deploying a Transformer model on an FPGA requires knowledge from multiple domains: an understanding of machine learning and Transformer architectures, familiarity with hardware design, experience with FPGA development and high-level synthesis (HLS). Researchers and engineers working on particle tracking or physics analysis may have strong domain knowledge and/or machine learning expertise, but not necessarily the hardware background required to design and optimize FPGA accelerators.

In practice, taking a trained model and turning it into an efficient FPGA design is still a complicated and largely manual process. It often requires a lot of hardware-specific adjustments and trial-and-error. Making this process simpler through better tooling and more automation would make FPGA acceleration far more approachable for researchers in the community who are not hardware specialists.

1.5 Research Objective

This thesis investigates the deployment of a Transformer-based tracking model on an FPGA platform. Rather than aiming for a complete end-to-end implementation, the focus is on implementing and evaluating a representative Transformer encoder layer, identifying hardware bottlenecks, and studying the impact of design choices such as on-chip parameter storage and reduced numerical precision.

In addition to the hardware and model-level challenges, this work also addresses the question of accessibility. A central motivation is to explore how the deployment process can be structured such that researchers with limited FPGA expertise can still experiment with and evaluate hardware acceleration for Transformer-based models. Reducing the required manual intervention and hardware-specific knowledge is considered an important step toward practical adoption in HEP environments.

Specifically, we consider a Transformer-based model for assigning hits to particle tracks and select a single encoder layer from a representative encoder-only model as a key component. This encoder layer is implemented and deployed on an FPGA. Additionally, we explore automation and toolchain possibilities for deploying and accelerating an existing model to an FPGA kernel.

The main research question of this thesis is:

How can a Transformer-based model for particle trajectory reconstruction be effectively implemented on an FPGA platform while considering realistic resource, accuracy, and usability constraints?

This question is addressed through the following sub-questions:

1. **State of the art:** What is the current state of the art in Transformer inference on FPGAs?

2. **Accuracy vs. hardware constraints:** How do hardware-oriented model adaptations—such as reduced numerical precision and quantization—affect the tracking accuracy of the selected model?

3. **Tooling and automation:** What functionality is required from a toolchain to deploy an existing Transformer model on an FPGA in a largely automated fashion, while minimizing the required hardware expertise?

1.6 Contributions

This thesis makes four main contributions:

- A systematic literature review of recent work on FPGA-based Transformer inference is conducted. The review focuses on model weight storage strategies, synthesis and tooling flows, model compression techniques, and computational optimizations. It provides an overview of current approaches and highlights trends and open challenges that are particularly relevant for particle-tracking applications in HEP.

- An FPGA accelerator prototype for a representative Transformer encoder layer is implemented and evaluated on the Zynq UltraScale+ MPSoC ZCU102 platform using Vitis HLS. The design investigates the feasibility of storing all model parameters in on-chip BRAM and applies optimizations such as loop pipelining, array partitioning, and the removal of large off-chip intermediate buffers. The resulting implementation is analyzed with respect to resource utilization (BRAM, LUTs, DSPs, FFs) and latency.

- The impact of reduced numerical precision is examined. Quantization and half-precision floating-point representations are applied to the encoder layer, and the resulting changes in tracking performance are evaluated using accuracy metrics derived from the full particle-tracking pipeline. This analysis highlights the trade-offs between hardware efficiency and model accuracy.

- A prototype Python-based toolchain, referred to as the FPGA Accelerator for Transformers (FAT), is developed. This toolchain can slice existing models and extract the sub-parts to be accelerated on an FPGA, extract model parameters, automatically run synthesis and implementation, and deploy and run the kernel on an FPGA. It demonstrates how semi-automated workflows can simplify the mapping of complex Transformer models to FPGA accelerators.

1.6.1 Publications related to this thesis

Parts of the research presented in this thesis have also been disseminated separately through academic publications. At the time of writing, one paper has been accepted for publication, while another is currently under review.

Survey on FPGA-based Transformer inference

The systematic literature review presented in section 3.2 has also been prepared as a separate survey paper titled *Recent Developments in Transformer Inference Deployment on FPGA Platforms: A Survey* [11]. This manuscript is currently under review at *SciPost Physics Proceedings*. The paper investigates recent advances in Transformer inference on FPGA platforms through a systematic literature review, with a focus on implementation strategies, optimisation techniques, and broader design trends. Its aim is to provide a structured overview and taxonomy that can help guide future research on efficient Transformer deployment.

TrackCore-F

The practical deployment workflow and FPGA implementation methodology developed in this thesis have also been described in the paper *TrackCore-F: Deploying Transformer-Based Subatomic Particle Tracking on FPGAs* [10]. This paper has been accepted for publication in *SciPost Physics Proceedings*. The paper focuses on methodologies and tooling for Transformer inference deployment on FPGA platforms. It presents the development approach, discusses practical resource constraints, and reports preliminary implementation results.

1.6.2 Source code

All source code for this project, including the FAT toolchain as well as supporting code for the systematic literature review is available at [9][1].

1.7 Thesis Structure

The remainder of this thesis is structured as follows. Chapter 2 provides background information on high-energy physics experiments, particle tracking, the use of machine learning in HEP, relevant datasets, and FPGA-based acceleration. Chapter 3 presents related work on Transformer architectures and their implementation on FPGA platforms.

Chapter 4 describes the research methodology, including the setup of the systematic literature review and the overall workflow used for accelerator design and evaluation. Chapter 5 details the implementation of the encoder-layer accelerator, covering the hardware platform, the high-level synthesis design, and the supporting software tools.

[1] Also available at https://arjanblankestijn.nl/fat

Chapter 6 presents and analyzes the experimental results, focusing on resource utilization, latency, accuracy implications, and tooling evaluation in relation to the research questions. Finally, Chapter 7 discusses the implications and limitations of the work, and Chapter 8 concludes the thesis and outlines possible directions for future research.

Chapter 2

Background

2.1 High-Energy Physics Experiments

High-Energy Physics (HEP) experiments, such as those conducted at the Large Hadron Collider (LHC) at CERN, produce an enormous volume of data. The LHC accelerates proton beams in opposite directions and collides them at specific interaction points, which are equipped with large detectors like ATLAS and CMS[36]. At the design luminosity of the LHC, approximately 25 proton-proton collisions occur in each $25ns$ bunch crossing, where proton bunches from opposite proton beams pass through each other at an interaction point. The upcoming High-Luminosity LHC (HL-LHC) upgrade aims to increase this number to around 200 simultaneous collisions per crossing [36]. The significant increase in event pileup leads to a dramatic rise in the number of detector hits and the overall data output.

Handling the massive influx of data from numerous overlapping collisions necessitates substantial upgrades to the data acquisition and trigger systems. In a typical Large Hadron Collider (LHC) detector, a multi-tiered trigger system filters 40 MHz of collision events down to a manageable recording rate—initially to approximately 100 kHz at the first level and then to approximately 1 kHz for storage—by rapidly discarding uninteresting events.

In the High-Luminosity LHC (HL-LHC) era, maintaining acceptable trigger rates with 5-8 times higher event multiplicity requires more sophisticated real-time data-reduction methods. One crucial improvement identified is the inclusion of tracking information in

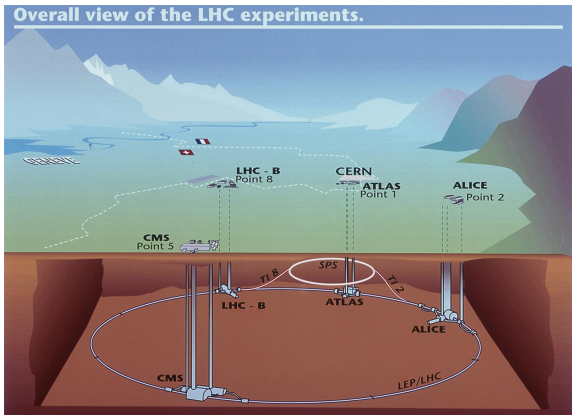


FIGURE 2.1: Diagram showing the locations of the four main experiments (ALICE, ATLAS, CMS and LHCb) [38]

.

hardware triggers. Including particle tracking within the trigger system can significantly enhance the selection of interesting events, thereby reducing the discarded amount of potentially interesting information.

Both the ATLAS and CMS experiments are undergoing major detector upgrades, particularly by introducing new silicon tracking detectors, and will implement dedicated real-time tracking subsystems to manage the extreme conditions of the HL-LHC. These advancements highlight the broader implications of the HL-LHC: not only will it advance the frontiers of physics, but it will also push the boundaries of computing and electronics, driving innovations in real-time data processing [14].

In this context, advances in electrical and computer engineering—ranging from high-throughput hardware triggers to machine-learning accelerators—are crucial. This thesis investigates how these modern hardware and machine-learning approaches can be applied to meet the real-time track reconstruction requirements of the High-Luminosity LHC.

2.2 Particle Tracking

In detectors, charged particles leave distinct marks, or hits, in successive layers of the detector (such as silicon sensor layers). The particle tracking problem involves reconstructing the trajectory (track) of each particle based on these spatially distributed hits. Essentially, particle tracking is a pattern-recognition problem in which hits originating from the same particle are identified and grouped, after which they are fitted to a trajectory model, typically a helix in a magnetic field.

Figure 2.2 shows how particle hits are detected. Track reconstruction generally occurs in two phases. First, the track-finding stage selects groups of hits that likely form a trajectory and assigns them to track candidates. Second, the track fitting stage determines the track parameters—such as curvature, direction angles, and impact parameters—from which particle properties like momentum and charge can be inferred. This separation of stages is beneficial because identifying the correct combinations of hits can be combinatorially complex, while fitting a known set of hits to a model (for instance, using a least-squares approach) is a more straightforward problem involving linear algebra [14].

Traditionally, the track-finding phase consumes most of the CPU time and often dominates the computational cost of event reconstruction. As a result, in large experiments much of the tracking is performed offline after data acquisition due to these computational constraints [1].

The dominant method for track finding in the ATLAS experiment has been the Combinatorial Kalman Filter (CKF). In this approach, tracks are constructed iteratively, beginning from seed hits in the initial layers of the detector. A Kalman filter is used to propagate a prediction of the track state to the next detector layer and evaluate the compatibility of candidate hits with the evolving track hypothesis. Compatible hits are incorporated into the track candidate, and the track parameters are updated accordingly. The Kalman filter's statistical framework—predicting the next state with associated uncertainties and refining it using measurements—is particularly effective for following a particle's curved path through the detector while accounting for effects such as multiple scattering and measurement noise. This approach forms the basis of the track reconstruction software used in the ATLAS experiment. [36]

However, the CKF is inherently a sequential algorithm that can generate multiple track hypotheses when ambiguities arise, resulting in significant computational costs. Each time several compatible hits are found in a detector layer, the algorithm may branch into multiple candidate tracks. If not controlled carefully, this branching leads to a rapid

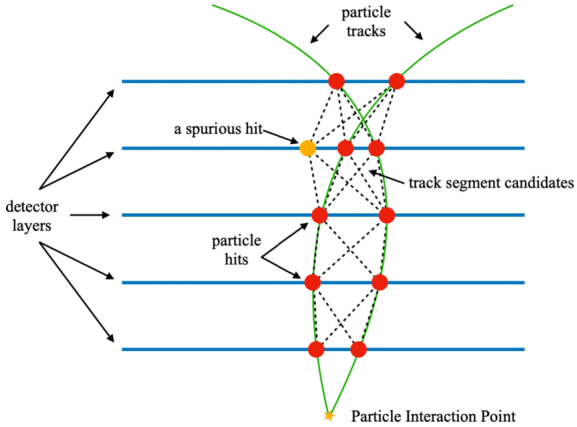


FIGURE 2.2: Illustration [39] of the particle track reconstruction problem showing how particles move through detectors registering 'hits'. The aim of particle tracking is to identify the trajectory of the particle(s).

growth in the number of track candidates that must be considered. Consequently, the computational complexity of Kalman-filter-based tracking can increase non-linearly with detector occupancy and the number of hits. [37]

The HL-LHC will operate at significantly higher instantaneous luminosity than the current LHC. Higher luminosity results in a larger number of simultaneous proton–proton interactions per bunch crossing, commonly referred to as pileup. At the HL-LHC, up to around 200 interactions may occur in a single bunch crossing. [8]

This high level of pileup leads to extremely dense hit environments in the tracking detectors, with tens of thousands of hits recorded in each event. In such conditions, hits originating from different particles can lie close together within the detector layers, making it difficult to determine which hits belong to the same trajectory. The resulting ambiguities significantly increase the number of possible hit combinations that must be evaluated during track finding.

Due to these challenges, track reconstruction is expected to remain one of the most computationally demanding components of event processing at the HL-LHC. Studies have shown that reconstructing a single high-pileup event with conventional tracking algorithms can require several seconds of CPU time, even with significant algorithmic optimizations. [1]

Therefore, there is a strong motivation to explore faster and more parallelizable tracking approaches that can cope with the increased detector occupancy while maintaining the accuracy required for physics analyses. This challenge has motivated research into alternative algorithms, including approaches that incorporate machine learning and advanced pattern-recognition techniques. [6]

2.3 Machine Learning in HEP

Due to the limitations of sequential algorithms in the conditions of the HL-LHC, the High Energy Physics (HEP) community has increasingly turned to machine learning to address the tracking problem. The premise is that machine learning models could identify track patterns more efficiently than traditional, hand-crafted algorithms. A significant milestone in this shift was the TrackML challenge in 2018, a public Kaggle competition where participants were provided with simulated HL-LHC event data containing approximately 100,000

detector hits per event. Their task was to cluster these hits into around 10,000 actual tracks [6]. The challenge showcased various innovative approaches, utilizing both classical algorithms and machine learning techniques. It demonstrated that a combination of thoughtful data representations and learning methods could achieve remarkable accuracy [6].

Recently, attention has turned to the Transformer architecture [41], a well-known ML model design commonly used in applications such as natural language processing or computer vision [28], as a promising approach for particle tracking [14, 15]. Transformers utilize a self-attention mechanism that can capture global relationships, making them well-suited for connecting widely separated detector hits that belong to the same track. The TrackFormers project [14] investigated a family of Transformer-based models aimed at one-shot track finding, requiring only a single forward pass (inference) of the ML model. Two encoder-only Transformer designs from this work are particularly relevant: EncCla and EncReg. Both designs operate on the complete hit data of an event simultaneously (known as "single-shot" tracking), but they utilize different methods to create tracks:

- **Encoder-Classifier (EncCla)**: This model utilizes a Transformer encoder to assign a track ID class to each hit in a single forward pass, effectively classifying hits into specific track labels. The model defines a fixed number of track classes (for example, by discretizing track parameter space into bins), and each hit is assigned a probability of belonging to each class. The class with the highest probability becomes the assigned track label for that hit. Hits that share the same predicted class label are grouped together as one reconstructed track. Since all hits are processed simultaneously and assigned labels in one step, EncCla represents a global approach that avoids iterative predictions. In the TrackFormers implementation, EncCla was designed with a sufficiently large network (with the largest version containing approximately 1.5 million parameters) to ensure that it can learn the complex mappings of hits to track IDs. Despite the output label space being extensive, with many potential track IDs, the model achieved high accuracy in correctly grouping hits.

- **Encoder-Regressor (EncReg)**: This model does not use explicit class labels; instead, it predicts continuous track parameters for each hit, such as the helix parameters of the particle that generated the hit. The Transformer encoder outputs a set of track parameter values for each hit (including curvature, angles, etc.), effectively mapping each hit into a track parameter space. Following this, a separate clustering algorithm (in the TrackFormers study, HDBSCAN) groups hits with similar predicted parameters to form track candidates. In comparison to classical methods, EncReg functions similarly to a learned Hough Transform; it transforms hit coordinates into a parameter space where tracks become clustered points. EncReg's architecture is somewhat simpler, containing tens of thousands of parameters, as it outputs only a few regression values per hit rather than a full classification across numerous categories. This makes it resource-efficient, although the post-processing step introduces additional complexity.

Both EncCla and EncReg demonstrated promising results on simulated data. The EncCla model stood out due to its speed and accuracy; it operates in a single inference step for the entire event and was found to be the fastest among the tested designs. In the TrackFormers benchmarks, EncCla achieved high track-finding accuracy without the need for any external post-processing, successfully clustering hits for tracks across a range of event complexities. Meanwhile, the EncReg model achieved comparable physics performance and even outperformed EncCla in some low-density scenarios. However, it incurred

a slight penalty in execution time and a minor drop in accuracy due to the clustering step. Regardless, both models scaled significantly better than traditional algorithms as the number of tracks increased.

Crucially, the study demonstrated that these transformer models could manage moderately complex events with millisecond inference times on modern GPU hardware. For instance, assigning approximately 10,000 hits to tracks—which is comparable to a full HL-LHC event scaled down to 200-500 tracks—took only about 3-6 milliseconds with the one-shot Transformer. In contrast, a highly optimized combinatorial Kalman filter took about 1.8 seconds per event on CPU, and a graph neural network approach required approximately 2.2 seconds on GPU. This several-hundredfold speedup underscores the potential of Transformer models to meet the demanding throughput requirements of real-time tracking. In principle, an inference latency of just a few milliseconds could align with the stringent Level-1 trigger budgets (which are on the order of tens of microseconds), though this could be achieved with further model compression or hardware acceleration. At minimum, these machine learning models are strong candidates for use in high-level triggers or online reconstruction farms, where slightly longer latencies (from a few milliseconds to tens of milliseconds) are acceptable.

The ultimate goal of integrating machine learning into high-energy physics (HEP) tracking is to deploy these models at the online stage—either in software triggers running on CPU/GPU farms or in dedicated hardware within the detector's trigger pipeline. Achieving this requires not only algorithmic speed but also implementation on low-latency hardware. The TrackFormers results have paved the way by illustrating that one-step Transformer-based tracking is algorithmically viable for HL-LHC data. The next step, and the focus of this thesis, is to implement such models on an FPGA-based platform to achieve the true real-time performance required at the LHC. The Transformer's structure (matrix multiplications and attention operations) must be translated into a firmware design that meets strict latency and throughput constraints. There are challenges to consider—ranging from resource limitations on FPGAs to the latency of multi-stage pipelines—which we will explore in detail. Nevertheless, the EncCla and EncReg models provide a compelling foundation: they are relatively small networks (with a memory footprint of only a few megabytes or less) and produce results in one pass, making them well-suited for hardware parallelization. The following sections will discuss the datasets on which these models are trained and evaluated, followed by an in-depth look at the considerations for deploying such machine learning models on FPGA hardware accelerators.

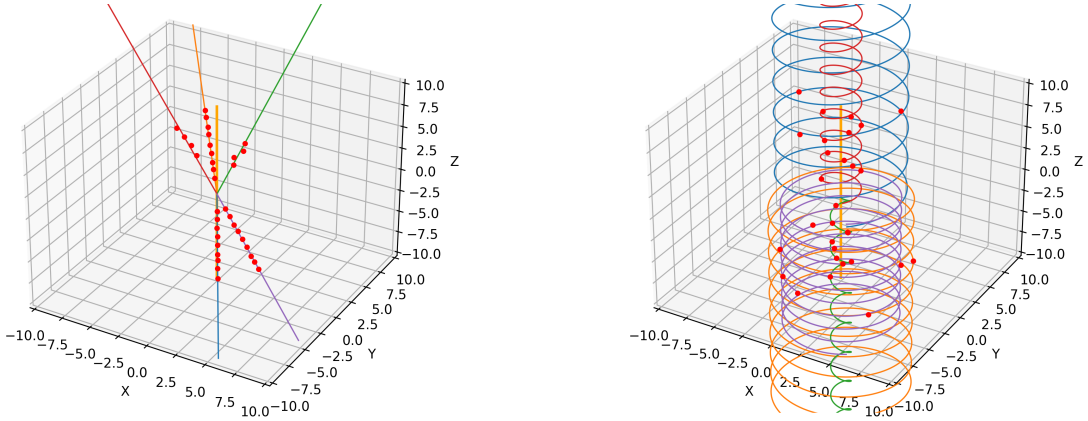
2.4 Datasets

Various versions of the EncReg and EncCla models have been trained on different datasets of increasing complexity. In all cases, the data is organized as collections of detector hits, where each hit corresponds to a single interaction of a charged particle with a detector element. Hits are grouped into events using an 'event_id', and all hits belonging to one event are processed together by the model.

2.4.1 REDVID

The first datasets are generated using the REDVID simulation framework [33], which simulates the propagation of subatomic particles in a virtual detector. Three REDVID datasets are used: one with linear tracks and two with helical tracks, containing between 10–50 or 50–100 tracks per event. Helical tracks emulate the motion of charged particles

in a magnetic field, while linear tracks represent simplified motion. Each dataset includes 100,000 simulated events with added noise to make the data more realistic. The detector geometry is loosely based on the ATLAS detector and includes several sub-detector types such as Pixel, Short-strip, Long-strip, and Barrel. In the REDVID datasets, each row corresponds to a single detector hit and includes both hit-level information and ground-truth track information. Hit positions are provided in cylindrical coordinates (r, θ, z), which align naturally with the detector geometry. In addition, REDVID stores analytical track parameters used by the simulator to generate the trajectories, such as parameters describing linear or helical motion. These parameters serve as truth-level information and are primarily used for supervision, evaluation, and debugging, rather than as direct model inputs.



(A) Linear tracks (B) Helical tracks

FIGURE 2.3: REDVID Sample events showing linear and helical tracks with hit coordinates [32]

2.4.2 TrackML

To test the models on more realistic and large-scale data, two datasets derived from the TrackML challenge [23] are also used. TrackML simulates proton–proton collisions at the High-Luminosity LHC, including pile-up effects with many simultaneous interactions. It provides three-dimensional hit coordinates together with extensive truth-level information about the particles that generated them. For this work, subsets containing 10–50 and 200–500 tracks per event are used.

Each row in the TrackML dataset represents a single detector hit and includes Cartesian hit coordinates (x, y, z), detector geometry identifiers, and truth-level particle labels, such as the particle identifier and kinematic properties. The data includes a large number of noise hits not associated with any true particle track, making the tracking task more challenging than in REDVID. During preprocessing, hit coordinates and track parameters are normalized and transformed into a spherical or cylindrical representation suitable for input to the Transformer models.

To enable controlled experimentation and rapid iteration during development, this thesis focuses primarily on models trained using the REDVID datasets.

2.5 FPGA Acceleration

FPGAs (Field-Programmable Gate Arrays) are reconfigurable hardware devices that enable the deployment of custom digital circuits for specific applications. Unlike CPUs and GPUs, FPGAs allow developers to define the hardware datapath itself using configurable logic, memory resources, and basic arithmetic and storage blocks.

This fine-grained configurability enables FPGAs to deliver unique advantages for machine learning inference when low latency, deterministic timing, and energy efficiency are critical, directly addressing requirements that commonly challenge traditional hardware. Rather than executing a model as a sequence of software instructions, an FPGA can be programmed to implement a custom hardware pipeline that mirrors the model's structure, allowing data to flow through it directly. Operations such as matrix multiplications, reductions, and non-linear activations can be executed concurrently and deeply pipelined, resulting in high throughput with predictable latency.

In high-energy physics experiments, FPGAs are widely used in trigger and data-acquisition systems, where decisions must be made within microseconds. Their deterministic behavior, parallelism, and low power consumption make them well-suited to real-time environments. As experiments grow to higher event rates and more complex algorithms, there is greater interest in running machine-learning models directly on FPGA-based trigger hardware.

Using FPGAs for machine learning inference extends their well-established role in HEP by bringing uniquely capable, low-latency inference to real-time environments. However, deploying larger and more complex models, such as Transformer architectures, remains challenging. These models demand significant computational power and on-chip memory, so mapping their operations efficiently to FPGA resources requires careful design. Additionally, while FPGA development tools for machine learning have improved, support for Transformer-based models is less mature than for traditional architectures, underscoring a key area for innovation.

The specific hardware platform and software tools used in this project are described later in Section5.1.

Chapter 3

Literature Research

3.1 Transformer models

A Transformer model is a type of Artificial Intelligence (AI) model first introduced by Vaswani et al [41] in 2017. A Transformer model is based on the concept of 'attention'. In this concept, each input is compared with three vectors, the Query (Q), Key (K) and Value (V) vectors. The Query and Key vectors are multiplied to compute the attention score. Then, these scores can be used with the Value vector to focus the 'attention' on the most relevant information.

$$\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{3.1}$$

To capture different kinds of relationships, the Transformer uses Multi-Head Attention (MHA). This means that several attention functions, called heads, run in parallel. Each head learns to focus on different parts or aspects of the input, and their results are combined at the end to form a single output.

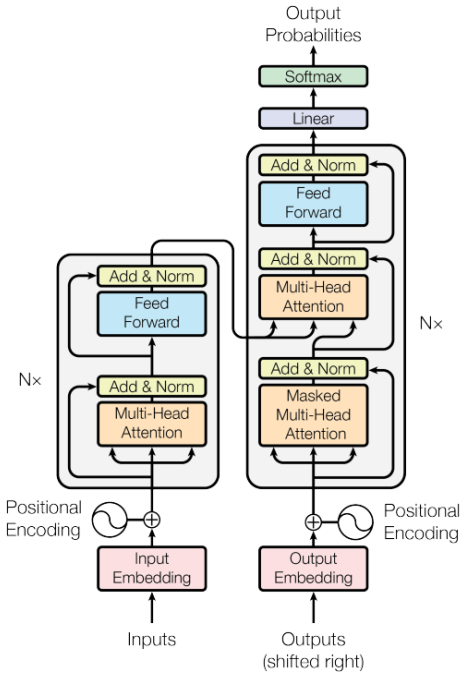


FIGURE 3.1: The complete Transformer model architecture proposed by [41]

The Transformer model also includes Layer Normalization (LN) and Feed-Forward (FF) layers. Layer Normalization normalizes the input values to improve stability during training. The Feed-Forward layer applies two linear transformations with a non-linear activation function in between. This allows the model to further transform and refine the information from the attention layers before passing it on to the next layer. The full Transformer architecture as seen in figure 3.1 consists of an encoder and a decoder. The encoder processes the input, the decoder processes the previous output and combines it with the input from the encoder. This way, an output is generated continuously step by step.

3.2 Systematic Literature Review

In recent years, research on Transformer models and their deployment on FPGAs has grown rapidly. This is reflected not only in the increasing number of applications using Transformers, but also in the growing body of work that explores how these models can be mapped efficiently onto reconfigurable hardware. Especially over the last two years, there has been a clear increase in publications focusing on FPGA-based Transformer inference as can be seen in Figure 3.2

To better understand the current state of the art and to avoid relying on an ad hoc selection of related work, a systematic literature review (SLR) was conducted. The goal of this review was to identify common design choices, optimisation strategies, and tooling approaches used in existing FPGA-based Transformer accelerators, as well as to highlight open challenges and gaps in the literature.

The review was carried out using a predefined and repeatable protocol, allowing the selection and analysis process to be reproduced or extended in the future. Rather than aiming to compare individual implementations in detail, the review focuses on identifying

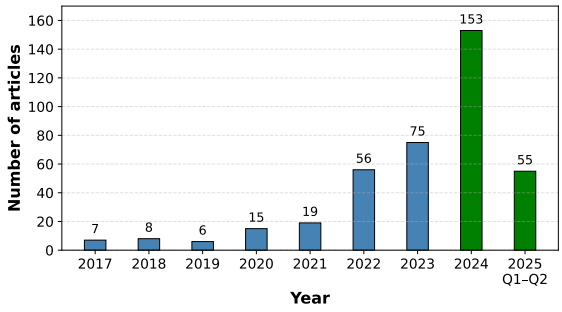


FIGURE 3.2: Distribution of article counts per year (till the end of 2025 Q2), focusing on the topic of Transformer inference on FPGAs and considering the three repositories: IEEE Xplore, Scopus, and arXiv as sources. The scope for this survey is the last two bars (in green).

broader trends across recent work, such as how model weights are stored, how numerical precision is handled, and which development tools are commonly used.

The full literature review has been written as a separate survey article and published independently [11]. In this thesis, the review is treated as preliminary research. Only a high-level summary of the findings that are most relevant to the design and evaluation choices made in this work is included here.

3.2.1 Literature Research Questions

The systematic literature review was guided by the following research questions:

- **RQ1:** How are Transformer models mapped onto FPGA platforms for inference, and which parts of the model are typically accelerated?
- **RQ2:** How do existing implementations manage model parameters and intermediate data, particularly with respect to on-chip and off-chip memory?
- **RQ3:** Which numerical precision and model compression techniques are most commonly used, and how do they affect performance and accuracy?
- **RQ4:** What tooling and development flows are used to implement Transformer accelerators on FPGAs, and what limitations do these tools impose?

The answers to these questions provide the basis for the related work overview and help motivate the design choices made in the remainder of this thesis.

3.2.2 Methodology

The literature review was conducted using a structured multi-step process. Relevant publications were collected from IEEE Xplore, Scopus, and arXiv using a fixed search query targeting Transformer models, FPGA platforms, and inference or acceleration. The search was executed in April 2025 and initially resulted in several hundred candidate articles.

A series of pruning steps was then applied to narrow this set down to the most relevant work. First, duplicate entries across repositories were removed. Next, articles published before 2024 were filtered out to focus on recent developments. Abstract-level screening was used to exclude papers that did not primarily focus on Transformer models, followed

by the removal of surveys and review papers. Finally, articles with a primary focus on training rather than inference were excluded.

After pruning, the remaining articles were ranked based on the occurrence of keywords related to synthesis, memory usage, compression, acceleration techniques, and inference. Only the highest-ranking papers were selected for detailed analysis.

Each selected article was analysed using a fixed set of attributes, including the scope of the accelerator (full model or partial), the strategy for storing model weights, numerical precision and compression techniques, and the type of tooling used for FPGA development. Based on these attributes, a taxonomy of approaches was constructed, which is used to organise the related work discussion in this thesis.

3.2.3 Key Findings and Implications for This Thesis

The systematic literature review reveals a number of consistent trends in recent research on FPGA-based Transformer inference. Rather than providing an exhaustive overview of individual implementations, this subsection highlights the dominant approaches identified in the literature and discusses how they inform the design choices made in this thesis.

Scope of acceleration. A first observation is that FPGA-based Transformer accelerators differ significantly in the scope of computation they target. Some works aim to implement full end-to-end inference entirely on the FPGA, as demonstrated by architectures such as FTrans [25]. While this approach offers tight integration and potentially low end-to-end latency, it often requires substantial on-chip resources and limits flexibility.

More commonly, recent work focuses on accelerating only the most computationally intensive parts of the Transformer model. ME-ViT [31], for example, proposes an accelerator that operates in multiple modes, each targeting a specific computation such as linear projection, self-attention, or feed-forward layers. Similarly, Li et al. [26] focus exclusively on tiled matrix multiplication, a fundamental kernel underlying several Transformer operations. These component-level approaches suggest that selectively accelerating critical kernels can provide a favourable balance between performance and resource usage, particularly for resource-constrained FPGA platforms. This observation motivates the focus on the encoder layer design strategy adopted in this thesis.

Parameter storage and memory management. Memory management emerges as a central concern across nearly all surveyed implementations. Designs that store all model parameters on-chip typically achieve lower inference latency by avoiding repeated off-chip memory transfers, but are limited by the available on-chip memory capacity. An example of this approach is BETA [21], which stores all model parameters in on-chip memory and loads the full input before inference, thereby minimising data movement during execution.

In contrast, accelerators that rely more heavily on off-chip memory can support larger models but have to deal with additional latency due to memory access overhead. To bridge this gap, several recent works adopt hybrid strategies that prioritise on-chip storage while falling back to off-chip memory when required. COBRA [35] exemplifies this approach by selectively utilising on-chip memory for frequently reused data while accommodating larger models through off-chip accesses. These trends highlight the importance of carefully balancing memory hierarchy design.

Numerical precision and model compression. Reduced-precision arithmetic and model compression techniques are pervasive in FPGA-based Transformer inference. Quan-

tization is widely used to reduce both memory footprint and computational complexity. HG-Pipe [20], for instance, employs 3-bit quantization, enabling the use of LUT-based multiply–accumulate operations and significantly reducing DSP usage.

More aggressive approaches include binary quantization, where model parameters are restricted to binary values. Du et al. [17] demonstrate that binary Transformer models can achieve exceptionally high energy efficiency, albeit at the cost of reduced generality and more restrictive training requirements. Between these extremes, mixed-precision quantization has emerged as a practical compromise. Byun et al. [13] propose a Hessian-driven approach that assigns higher precision to more sensitive parameters and lower precision elsewhere, achieving substantial compression with minimal accuracy loss. However, such schemes increase architectural complexity, as the accelerator must support multiple numerical formats.

In addition to quantization, pruning techniques are commonly used to introduce structured sparsity into Transformer models. Wang et al. [42] propose a block-wise balanced pruning method that achieves high sparsity while maintaining accuracy through uniform pruning across matrix blocks. Importantly, several works emphasize that sparsity must be explicitly exploited by the hardware to yield performance gains. Li et al. [27], for example, demonstrate that the inner product of the Q and K matrices often contains a large fraction of near-zero values and propose a low-precision precomputation step to identify and skip insignificant operations.

Tooling and development flows. The surveyed literature reveals a clear divide between Register-Transfer Level (RTL) designs using HDLs such as VHDL or Verilog, and High-Level Synthesis (HLS)-based approaches that generate hardware from C/C++ descriptions. RTL designs generally offer greater control over resource usage and timing, but require substantial expertise and development effort. HLS-based approaches, while offering faster development and improved productivity, often trade off fine-grained optimisation capabilities.

Beyond traditional HLS workflows, higher-level frameworks aim to further reduce the barrier to FPGA-based machine learning deployment. hls4ml [18] enables translation of trained neural networks into HLS code and has gained popularity in the high-energy physics community. Recent work has extended hls4ml towards supporting Transformer models [22], although full support is still under development. Alternative approaches operate at the RTL level while retaining automation. Ling et al. [29] propose Python-driven VHDL template generation for integer-only quantized models, illustrating a middle ground between full manual RTL design and HLS abstraction.

Implications for this work. Overall, the literature indicates that efficient FPGA-based Transformer inference relies on selective acceleration of key computational kernels, careful management of on-chip and off-chip memory, and judicious use of reduced-precision arithmetic. Rather than pursuing fully general or end-to-end solutions, many successful designs focus on well-defined components and tailor architectural optimisations to those targets. Informed by these findings, this thesis focuses on selectively accelerating a representative Transformer encoder layer, with an emphasis on keeping parameters on-chip, using HLS for implementation, and supporting a semi-automated CPU–FPGA deployment workflow.

Chapter 4

Methodology

The research questions of this thesis are addressed through the implementation and evaluation of a prototype FPGA accelerator for a Transformer-based particle model. Rather than implementing a complete end-to-end model, the focus is placed on the deployment of a single Transformer encoder layer. This allows the study to concentrate on the core computational components of the architecture while avoiding model-specific operations that are not inherently part of the Transformer design.

Additionally, the tracking models considered in this work (EncReg and EncCla, discussed in Section 2.3) are encoder-only architectures. As a result, implementing the decoder portion of the Transformer is not necessary for evaluating the feasibility of FPGA-based inference.

Alongside the accelerator implementation, a semi-automated deployment toolchain is developed to support the process of adapting pre-trained models to the FPGA implementation. This toolchain assists with tasks such as exporting model parameters, preparing data structures for the hardware implementation, and integrating the synthesized accelerator kernel with the remaining parts of the model pipeline.

The overall methodology follows a workflow in which first, the encoder layer is reproduced in software to obtain a verifiable reference implementation. Next, the encoder logic is translated into a C implementation compatible with High-Level Synthesis (HLS) and validated through simulation. Hardware-oriented optimizations are then applied in order to evaluate achievable performance and resource utilization on FPGA hardware. Finally, a deployment toolchain is developed to simplify the integration of trained models with the accelerator framework.

4.1 Model Preparation

The first step is to develop a reference implementation of the Transformer encoder layer in a high-level programming language. This implementation mirrors the behavior of the original PyTorch model and produces identical outputs for the same inputs.

While this step is technically optional, it provides several practical benefits. Reimplementing the model helps clarify the exact computations performed within the encoder layer and exposes the structure of the data flow between operations. Additionally, the reference implementation makes it straightforward to extract intermediate results, which can later be used to verify individual components of the FPGA implementation.

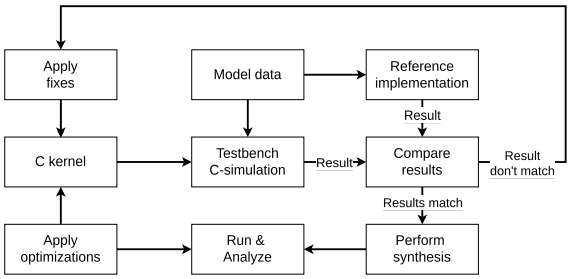


FIGURE 4.1: Diagram showing the general iterative workflow of developing an HLS kernel for a Transformer accelerator.

4.2 HLS Kernel Development

4.2.1 C Implementation and Simulation

After validating the reference implementation, the encoder layer logic is translated into C code suitable for High-Level Synthesis. At this stage, the focus is on functional correctness rather than performance optimization. The objective is to create a baseline implementation that reproduces the behavior of the reference model.

To verify correctness, the C implementation is evaluated using C-level simulation. In this process, identical input data is provided to both the reference implementation and the HLS-compatible C code. The resulting outputs are then compared element-by-element within a testbench environment. If discrepancies are detected, the C implementation is corrected and the simulation is repeated until the outputs match.

Once functional equivalence is established, the design can proceed to the hardware-oriented optimization stage.

4.2.2 Hardware-Oriented Optimization

With a correct C implementation available, the next step is to prepare the design for efficient execution on the FPGA and synthesize it into a hardware kernel. This stage focuses on restructuring the implementation to better utilize the available hardware resources.

Several optimization strategies are applied during this process. These include increasing computational parallelism, organizing operations so they can execute in a pipeline, and minimizing unnecessary memory transfers. Particular attention is given to reducing external memory accesses and maximizing on-chip data reuse.

The optimized implementation is synthesized using the HLS toolchain and deployed as an FPGA kernel. The resulting design is analyzed in terms of latency, throughput, and hardware resource utilization. Based on these results, additional iterations of optimization may be performed to further improve performance or reduce resource consumption.

4.2.3 Testing and Hardware Validation

After synthesizing the optimized kernel, the final step of the kernel development process is to validate its behavior on the target FPGA platform. The kernel is executed on the board and its output is compared against the results produced by the reference implementation.

Once functional correctness is confirmed, the encoder kernel can be integrated with the remaining parts of the model pipeline. This allows the impact of the accelerator on overall performance and numerical accuracy to be evaluated.

4.3 Deployment and Automation Toolchain

Finally, a semi-automated toolchain is developed that combines and connects all the steps. These steps are model preparation, synthesis and deployment.

For model preparation, the toolchain should be able to analyze and partition the model such that the required encoder layers are accelerated on the FPGA. Furthermore, the toolchain has to be able to extract the parameters required by the accelerator and format them accordingly for the kernel.

Then, for the synthesis step, the toolchain should be able to run the entire process of synthesis, implementation and place and route, with a generated bitstream as the end result. To accomplish this, the toolchain will have to interact with external synthesis tools.

Finally, during deployment, the toolchain should be able to load the generated bitstream onto the FPGA and initialize the accelerator. It should also manage the data exchange between the host system and the accelerator, such that the encoder layers executed on the FPGA can be integrated with the remaining parts of the model that run on the CPU. In this way, the accelerator can operate as part of a hybrid inference pipeline, where different components of the model are executed on different hardware platforms.

4.4 Evaluation Metrics

To evaluate the proposed system, several metrics are defined that capture both the performance of the FPGA accelerator and the usability of the deployment toolchain. The accelerator is evaluated in terms of computational performance, hardware resource usage, and numerical accuracy. The toolchain is evaluated separately based on its ability to support deployment of multiple models within the same workflow.

Table 4.1: Summary of evaluation metrics used for the FPGA accelerator

Metric	Description	Unit
Latency	Time required to process one encoder layer	ms
Throughput	Encoder layers processed per second	layers/s
Power Efficiency	Performance relative to power consumption	layers/s/W
Resource Utilization	FPGA hardware resources used	% or absolute
RMSE	Numerical deviation from reference output	-

4.4.1 Encoder Accelerator Kernel

The FPGA accelerator is evaluated using several metrics that characterize its performance, hardware efficiency, and numerical accuracy. A summary of these metrics is provided in Table 4.1.

1. **Latency**

   Latency is defined as the time required to process a single encoder layer. It is measured as the time between providing the input to the FPGA kernel and receiving the complete output. Latency measurements are obtained during runtime execution of the accelerator on the FPGA.

2. **Throughput**

   Throughput represents the maximum number of encoder layers that can be processed per second when the design operates in steady state. In simple implementations

throughput may approximate the inverse of latency. However, when pipelining or parallelism is introduced, throughput can exceed this relationship.

3. **Power Efficiency**

   Power consumption during inference is measured to evaluate the energy efficiency of the accelerator. Rather than comparing raw performance alone, FPGA results are considered in terms of performance per watt. This allows for a more meaningful comparison with CPU and GPU implementations that operate under significantly different power budgets.

4. **Resource Utilization**

   Resource utilization describes the amount of FPGA hardware resources required by the accelerator. This includes resources such as BRAM, DSP slices, LUTs, and flip-flops. These values are obtained from the synthesis and implementation reports generated by the FPGA toolchain and provide insight into the scalability and hardware bottlenecks of the design.

5. **Root Mean Square Error (RMSE)**

   Numerical precision is evaluated using the Root Mean Square Error (RMSE). The RMSE is calculated by comparing the output produced by the FPGA kernel with the reference output obtained from the CPU implementation using full 32-bit floating-point precision. Specifically, the RMSE is calculated by squaring the difference between the FPGA kernel output and reference output, and subsequently taking the square root of the mean of all these differences. This metric indicates how much the FPGA result deviates from the baseline model. Normally, overall model accuracy would be used as the primary metric for correctness. However, this is not possible in this work because the FPGA implementation only covers the encoder layer rather than the complete tracking model. Additionally, some models used in this thesis, such as EncReg, include additional processing steps after the machine learning inference stage. RMSE therefore provides a suitable layer-level metric for evaluating the impact of reduced numerical precision or quantization.

4.4.2 Toolchain

Evaluating the usability of the developed toolchain is less straightforward than measuring hardware performance. Unlike latency or resource utilization, it is difficult to define clear quantitative metrics that capture the practical usefulness of the deployment workflow.

For this reason, the toolchain is evaluated through an application-based approach. The accelerator and associated tooling are first developed and tested using a single model (EncReg). Once a stable prototype with the required functionality is established, the same toolchain is used to attempt deployment of a different model (EncCla).

This second deployment serves as a validation step for the generality of the toolchain. Any limitations, missing features, or usability challenges encountered during this process are used to identify shortcomings of the workflow and provide a qualitative assessment of its practicality.

Chapter 5

Implementation

5.1 Hardware and Software tools

The AMD Zynq UltraScale+ MPSoC ZCU102 [3] will be used as the development platform during this project. This board is an evaluation kit which includes an Arm Cortex-A53 processor, also called a Processing System (PS), and programmable logic (PL i.e. FPGA). The Programmable Logic is the Zynq UltraScale+ XCZU9EG. Table 5.1 lists the hardware specifications. The 32.1 Mb BRAM On-Chip memory is divided into 912 36Kb BRAM blocks. For implementing and developing the kernel, Vitis HLS [4] and Vivado [5] 2022.2 are used. Vitis HLS is used to write C/C++ HLS code and run C-simulation to verify functionality. Vitis HLS does have the ability to synthesize this to register transfer level (RTL) code but cannot perform the place & route steps. Instead, Vitis HLS can export the RTL code as a Vivado IP. This IP can then be used within Vivado to integrate it into a block design with the Xilinx MPSoC. Finally, the final synthesis, place & route and bitstream generation using Vivado can take place.

Table 5.1: ZCU102 Evaluation Kit hardware specifications.

DDR4 Memory (PS)	4 GB
DDR4 Memory (PL)	512 MB
System Logic Cells	600K
LUTs	274080
Flip-flops	548160
DSP Slices	2,520
On-Chip Memory / BRAM	32.1 Mb / 912 blocks

5.2 Requirements

To run the current version of the FAT toolchain, a few software and hardware requirements must be met. First, the synthesis flow depends on the Xilinx toolchain, in particular Vitis HLS 2022.2 and Vivado 2022.2. Furthermore, at the moment the integration between the Python application and Xilinx toolchain requires a Linux environment, since it relies on bash and TCL scripts.

The runtime part of the toolchain requires a Python environment with the required packages installed. In addition to Python itself, this includes at least `onnx`, `onnxruntime`,

`numpy`, `pyyaml`, and `pynq`. These packages are used for model inspection, CPU-side execution, configuration handling, and communication with the FPGA kernel.

In addition, the toolchain requires access to an FPGA platform. In its current form, only the Xilinx ZCU102 board is supported.

As model input, the toolchain expects a Transformer model in ONNX format. The minimum supported ONNX opset is version 21. If an older opset is detected, FAT will attempt to upgrade it automatically, although this is not guaranteed to work in all cases.

In practice, not all steps have to run on the same machine. Synthesis requires a system with Vivado and Vitis HLS installed, while deployment and inference require a system with access to the target FPGA board and a working PYNQ environment.

5.3 High-Level architecture

This section describes the high-level architecture of the final system implementation. The goal of this system is to design a process where an existing Transformer model can be deployed and accelerated on an FPGA. The EncCla and EncReg models used in this work are both encoder-only models. Therefore, accelerating and deploying the encoder layers is the main focus of this work. However, the system is designed in such a way that it should be relatively easy to extend it beyond just the encoder layers. Figure 5.1 shows all steps that must be taken in the deployment process before the model can be run. This includes steps such as model conversions, template generation, synthesis and more. Some of these steps are done using Vitis and Vivado. For other parts a dedicated Python application called FAT (FPGA Accelerator for Transformers) is developed. The FAT application consists of different parts which implement different steps of the deployment and running process. These steps are mainly model splicing, kernel synthesis, and finally deployment and running of the kernel. The FAT application has a few different dependencies. In particular, it uses the Vitis and Vivado for kernel synthesis, and it uses PYNQ [2] for kernel deployment. Due to these dependencies, it may not be possible for the user to run all steps on the same machine. The synthesis steps must be executed on a machine with Vitis and Vivado installed. While the last deployment and running of the kernel must be executed on a machine with the FPGA. The remaining sections of this chapter will explain each of these steps in detail.

5.4 Preparation

Before generating the kernel that can be deployed on the FPGA, some preparation steps must be taken.

5.4.1 Convert model to ONNX

The TrackFormers models used in this work are originally developed in PyTorch. To enable flexible deployment and integration with both software and hardware components, these models are converted to the Open Neural Network Exchange [16] (ONNX) format. ONNX is an open, framework-agnostic standard for representing machine learning models, designed to improve interoperability across tools and platforms.

PyTorch provides built-in functionality to export trained models to ONNX. The resulting ONNX model represents the network as a directed graph, in which nodes correspond to individual operations and edges represent tensor dependencies. A key advantage of

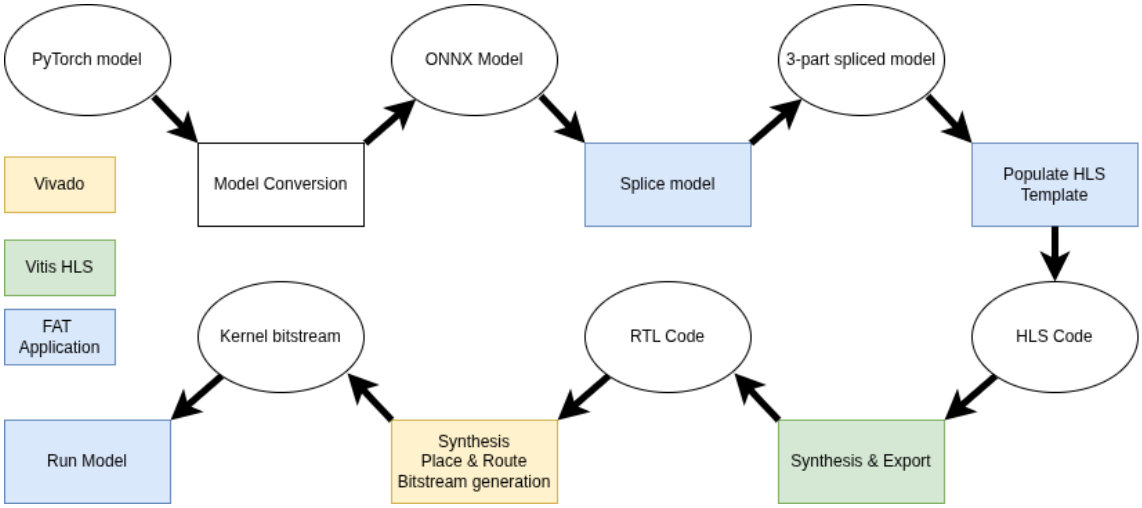


FIGURE 5.1: Diagram showing all steps in deploying and running an existing Transformer model on an FPGA.

using ONNX is that the model graph can be partitioned into smaller subgraphs. This enables incremental development and verification, as individual components, such as a single Transformer encoder layer, can be isolated and executed independently.

ONNX employs *opset versions* to specify the set of supported operations and their semantics. At the time of this writing, the most recent ONNX opset is version 21. Depending on the PyTorch version used during export, the generated ONNX model may target an older opset. To ensure compatibility with the execution environment, FAT automatically checks the opset version when loading a model and attempts to upgrade it to opset 21 if required.

Model splicing

The accelerator implementation in this work is designed specifically for Transformer encoder layers. Therefore, the ONNX model is partitioned such that only the encoder computation is offloaded to hardware. All operations preceding and following the encoder layers are executed on the CPU using ONNX Runtime.

Model slicing is performed by exploiting the hierarchical naming convention used by ONNX for nodes and tensors. For example, encoder-related nodes are typically named using paths such as:

`/encoder/layers.0/self_attn/Transpose`
or
`/encoder/layers.1/self_attn/MatMul`

By identifying the first and last nodes associated with a specific encoder layer, the corresponding subgraph can be extracted into a separate ONNX model.

Using this approach, the original model is partitioned into three logical components: a pre-encoder subgraph, an encoder-layer subgraph, and a post-encoder subgraph. The pre- and post-encoder subgraphs are executed on the CPU, while the encoder-layer subgraph is mapped to the FPGA accelerator. Figure 5.2 shows this process. Functional correctness of the hardware kernel can then be verified by comparing its output against the output produced by executing the corresponding encoder subgraph using ONNX Runtime.

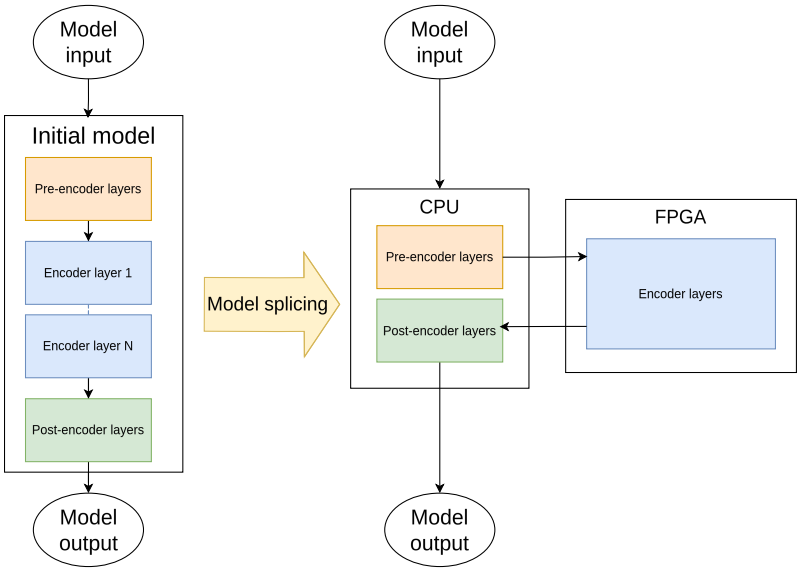


FIGURE 5.2: Diagram showing how the model is spliced and mapped between the CPU and FPGA.

5.4.2 Reference implementation

As discussed in Section 4.1, the first step is to develop a reference implementation. This is not necessarily part of the deployment process. However, having a reference implementation is an important step during the development of the kernel. The reference implementation allows the possibility of comparing the final output of the kernel, as well as intermediate results. It was decided to implement this reference implementation in Python using standard NumPy methods. This implementation is developed by implementing functions for computing the various parts of an encoder layer, i.e. the multi-head attention, layer normalization and feed forward. The input to these functions are NumPy arrays and the standard NumPy functions for matrices are employed. Very little effort is put into optimizing this implementation. That is okay since the goal is not a performant inference computation but rather to get a better understanding of the operations at place. The final implementation was tested using the smallest EncReg model and compared to running it using ONNX Runtime in order to verify that the final outputs are correct.

5.5 HLS Kernel development

The first step in developing a kernel is to develop a kernel that can run a single encoder layer. However, the entire synthesis process, including deployment on an FPGA takes quite a long time. In order to speed-up the development workflow, it is essential to verify the kernel's correct functioning before going through this synthesis process. To achieve this, test-driven development using C-simulation is employed. A single encoder layer consists of three main components: the Multi-Head Attention, Layer Normalization, and Feed-Forward network. Separate testbenches are created for each of these components, as well as for the complete encoder layer, to ensure correct functionality before synthesis. Each testbench supplies the input data, weights, and biases, and verifies the output against the known-good results from the reference NumPy implementation. Using C-simulation enables early functional verification and significantly reduces overall development time. Once the kernel is verified to produce a correct output, it can be synthesized using Vitis

HLS. Vitis HLS can report initial information on performance and resource utilization. This report won't be identical to the final output. Vivado synthesis might produce slightly different results, but it should be quite close. Therefore, the Vitis HLS synthesis report is used for intermediate evaluation during development. And only after certain milestones, in particular for final evaluation, will the full synthesis process using Vivado be carried out.

The development of the encoder kernel was done in an iterative manner, resulting in roughly 3 iterations. These iterations are described below.

5.5.1 1st iteration: basic implementation

The first goal in the development phase is to simply get a working prototype which can compute inference of a single encoder layer without focusing on resource consumption requirements or performance. This first implementation is largely developed by directly translating the reference NumPy implementation into C code. Additionally, very little thought has been put into optimizing for memory or performance. Therefore, some of the intermediate results are really big and require the use of off-chip buffers.

Resource	**Value**
Execution time	1.5s
BRAM	347
FF	74484
DSP	67
LUT	65893

TABLE 5.2: Resource consumption for the first basic encoder kernel implementation.

5.5.2 2nd iteration: Removal of intermediate off-chip results

Although the first prototype was functionally correct, its execution time was very high. A likely cause was the use of large off-chip buffers for intermediate results. In the initial implementation, the attention computation was divided into separate loops for each processing step. While this made the design straightforward, it also meant that several large intermediate tensors had to be written to and read from memory. Since these tensors scale with the square of the sequence length, they introduced a substantial memory overhead.

In the second iteration, the attention computation was restructured to avoid storing these large intermediate results explicitly. Rather than materializing the full intermediate tensors in off-chip memory, the computation was performed in a more direct sequence. Masking was applied during score computation, normalization followed immediately, and the resulting values were consumed directly in the weighted sum over the value vectors. In this way, intermediate data was processed on the fly instead of being stored as full tensors. This removed the need for large off-chip intermediate buffers and significantly reduced the associated memory traffic.

5.5.3 3rd and final iteration

For the final iteration of the encoder HLS code, all model parameters have been moved to on-chip buffers. Additionally, HLS array partitioning together with loop unrolling is employed to further optimize computation. Table 5.4 shows the resource utilization for

Resource	1st iteration	2nd iteration
Execution time	1.5s	0.7s
BRAM	347	248
FF	74484	77123
DSP	67	230
LUT	65893	62657

TABLE 5.3: Resource utilization after need for removing large intermediate results

this implementation compared to the previous optimized version. As can be seen, the execution time (latency) of a single pass has decreased significantly. Additionally, the BRAM usage has also gone down even more meaning that the total BRAM usage is now 24% of the total available BRAM. This is significant because this means that it should be possible to fit all 4 encoder layers of this EncReg model on the FPGA in its entirety.

Resource	2nd iteration	Final iteration
Execution time	0.7s	0.105s
BRAM	248	222.5
FF	77123	30632
DSP	230	241
LUT	62657	29199

TABLE 5.4: Resource utilization of a single encoder layer after moving all parameters to on-chip buffers and implementation array partitioning together with loop unrolling.

5.5.4 Multilayer kernel

In the final design, multiple encoder layers are implemented within a single HLS kernel. This approach has two main advantages. First, it avoids repeated transfers of intermediate activations between the processing system and programmable logic. When layers are executed as separate kernels, each layer output must be written to external memory and read back as the input to the next layer, introducing unnecessary latency and memory bandwidth overhead. By keeping intermediate activations on-chip, the full encoder stack can be executed with significantly reduced off-chip communication.

Second, integrating multiple layers into one kernel enables task-level pipelining across layers using the HLS `DATAFLOW` pragma. While loop pipelining and unrolling improve performance within a single layer, a multi-layer design allows different layers to operate concurrently on different input sequences. The behavior of this pipeline is illustrated in Figure 5.3. During the initial invocations, the pipeline is gradually filled (warmup phase), and no complete encoder output is yet available. Once all layer stages are occupied, the design reaches steady state, where each kernel invocation advances all active inputs by one layer and produces one full encoder output. This overlapping execution increases throughput without increasing the per-invocation latency.

To enable dataflow, the kernel is structured as a set of coarse-grained tasks: input acquisition, per-layer computation, and output streaming. The `DATAFLOW` pragma allows these tasks to execute concurrently, provided that their data dependencies are explicit and non-aliased. A major constraint of Vitis HLS is that intermediate memories within a dataflow region must have a clear producer–consumer relationship and must not be accessed

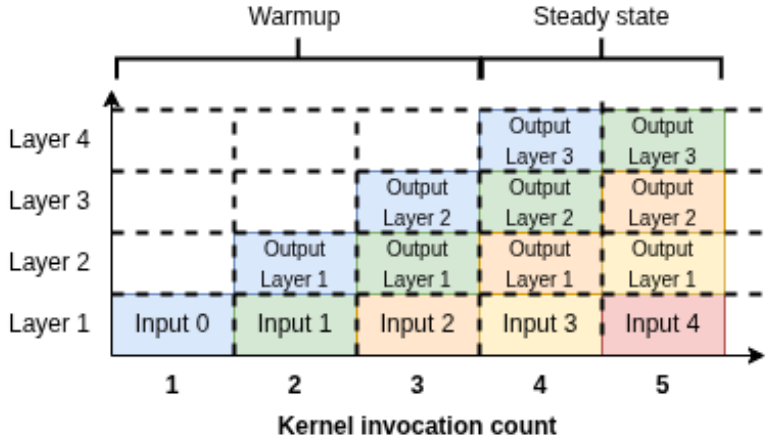


FIGURE 5.3: Illustration of the pipelined execution of a kernel implementing four encoder layers. Each invocation advances inputs by one layer. After the initial warmup, the design reaches steady state, producing one full encoder output per invocation. In the diagram, each color represents the path taken by a single input.

through runtime-selected pointers. To satisfy this requirement, each pipeline stage uses private on-chip buffers that are statically bound to a specific layer at compile time.

To overlap computation across kernel invocations, the design maintains persistent intermediate activation buffers. A ping-pong buffering scheme is used to store intermediate results between layers, where one buffer bank holds the activations of previous sequences while the other bank is written with the current sequence's results. The buffer roles alternate between invocations, ensuring that read and write accesses do not conflict. Metadata such as the attention mask length is stored alongside each buffer to guarantee correct processing as sequences progress through deeper layers.

All model parameters are loaded once into on-chip memory and stored per layer. This eliminates repeated DDR accesses during inference and avoids contention between dataflow stages, as each layer reads a dedicated set of parameters. This organization not only improves performance but also simplifies dataflow analysis by the HLS tool.

5.6 HLS Code Template

At this point, the HLS implementation is largely complete. However, the goal of this work is not to deploy one single fixed model. Instead, the same kernel structure should be reusable for multiple Transformer models with different dimensions and different numbers of encoder layers. To support this, the HLS code is organized as a template. The top part of the template defines model-specific constants, such as sequence length, model dimension, and number of layers. By changing these constants, the same kernel structure can be adapted to different model configurations without modifying the full implementation. Figure 5.4 shows a code snippet containing some of these constants.

5.7 Vivado kernel integration

After the kernel is synthesized and exported, it can be imported and integrated with the Zynq MPSoC with Vivado. A block design is created which contains a block for the Zynq Ultrascale+ MPSoC, and a block representing the IP Kernel imported from Vitis HLS. The

```
constexpr int SEQ_LEN  = 500;
constexpr int D_MODEL  = 32;
constexpr int QKV_DIM  = 3 * D_MODEL;
constexpr int H        = 4;
constexpr int d_k      = D_MODEL / H;
constexpr acc_t LN_EPSILON = 1e-5f;

#ifndef NUM_LAYERS
#define NUM_LAYERS 1
```

FIGURE 5.4: HLS Code snippet of the constants that define the dimensions of the encoder layer to be run by the kernel.

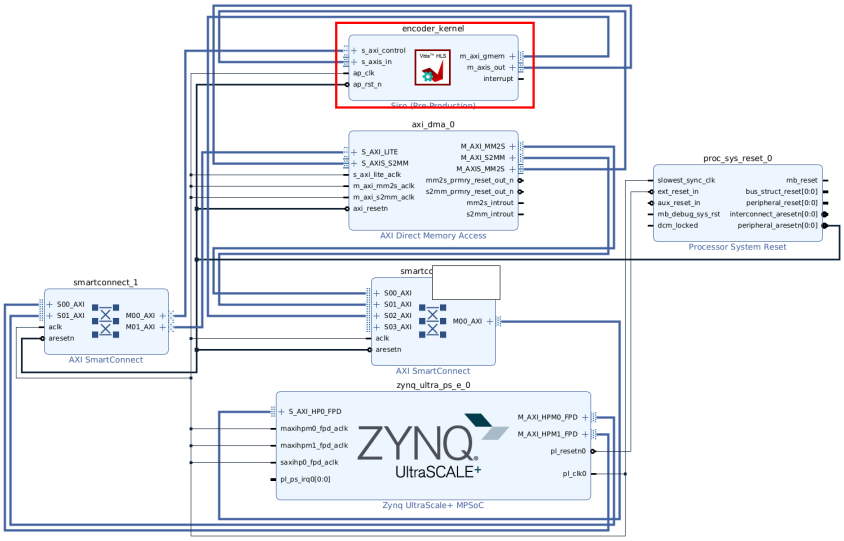


FIGURE 5.5: Vivado Block Design showing how the IP kernel is connected to the Zynq MPSoC. The kernel is circled in red.

IP kernel has a port called *s_axi_control* which is an AXI slave connection used for control registers for functions such as starting, stopping and resetting the kernel. Furthermore, the IP block has a port called *m_axi_gmem* which is the AXI master connection for all data transfers. Both AXI ports are connected to the Zynq block via a set of AXI interconnect blocks. Once the block design is completed and generated, the final synthesis, place and route can be run. Then, the bitstream can be exported for running on the ZCU102.

Vivado can automatically detect any changes to the IP kernel exported from Vitis HLS as long as Vitis HLS exports any new version to the same location. If the new version contains changes to the block design, such as new or different AXI ports. They must first be connected. After which the synthesis, place and route can be run again.

5.8 FAT Python application

So far, we have 3 distinct steps implemented. First, the model is prepared by converting it to ONNX and subsequently sliced into 3 parts. Then, we have developed HLS code for the kernel and export it to RTL code using Vitis. Finally, we take this RTL code, and import it into Vivado to integrate it with the Zynq through AXI streams. In order to tie all these steps together, as well as actually run the model, a Python application is

developed called FPGA Accelerator for Transformers (FAT). The FAT application consists of multiple distinct parts. Some of which must be executed on a third machine running Vitis/Vivado, some parts can be run on the host CPU.

5.8.1 ONNX model graph inspection

The ONNX model is represented as a graph of nodes. These nodes have names such as `/encoder/layers.0/norm2/LayerNormalization`. Based on these names, FAT can detect the existence of encoder layers. Depending on the settings determined by the user, FAT will split up the model. Exactly at which parts the model will be sliced is determined through configuration by the user. Before being able to run the model, the model parameters (weights and biases) must be loaded into the kernel. FAT can automatically detect and extract these parameters. Then, depending on how many layers are run on the kernel, FAT will reformat the parameters in the correct format and load them to the FPGA.

5.8.2 TCL Scripts

The kernel is synthesized using Vivado and Vitis. However, the need to learn and use this software is quite a blockade in deploying existing models. Therefore, it is important to be able to automate this part. FAT can populate the HLS templates discussed in Section 5.6. Then, for the actual synthesis steps, scripts have been developed. There are 2 main scripts. The first is the 'vitis' script which performs C-simulation, synthesis and exporting of the RTL. The second script is the 'Vivado' script, which imports the RTL into the block design. Regenerates the block design, runs synthesis, place & route, as well as bitstream generation. The final outputs are the `.hwh` and `.bit` files needed to run the kernel on the FPGA using Pynq. The scripts are based on TCL, a command line tool for Xilinx IDEs and bash scripts. This does mean that for now this tool is limited to Linux machines supporting bash. However, it should be trivial to update it for Windows machines. Both scripts are called from FAT. Therefore, the end user does not have to learn or have knowledge of Vitis and/or Vivado. The only requirement is that Vitis HLS and Vivado are installed on the machine.

5.8.3 PYNQ

The execution of the IP kernel is done with PYNQ [2]. PYNQ is a library for interfacing with Xilinx Programmable Logic using Python. PYNQ can load the kernel using the bitstream file exported from Vivado. In order to transfer data back and forth, memory must be allocated. For all weights and bias matrices, a buffer is created to allocate memory such that the IP kernel can access the data that is written to this buffer. Then, the output of the first model slice ('model A') is piped into the input to the kernel and the kernel is started, once the IP kernel is finished the output is read from the buffer. The output is then piped into the final model part, 'model C'.

5.8.4 Usage

As mentioned earlier, FAT consists of multiple distinct components that together form a single Python application. The application has one entry point, `main.py`, and different parts of the toolchain are executed by providing specific command-line arguments. Specifically, the application can be run by executing the following command:

```
python main.py <data-directory> <command>
```

When running FAT, the first required argument is the `data-directory`. This directory acts as a central workspace in which all required inputs and generated artifacts are stored, including models, configuration files, and FPGA bitstreams. During the deployment process, additional files may be automatically generated by the tooling and placed in this directory.

The second required argument is the `command` argument, which specifies the step to be executed.

Commands

FAT currently supports the following commands:

- **split**
  The `split` command searches for a `model.onnx` file in the data directory and partitions it into three sub-models. This is done by detecting encoder layers in the ONNX graph. The number of encoder layers to offload to the FPGA is determined by the configuration stored in `model.yaml`. Based on this configuration, the corresponding encoder layers are extracted into a separate model slice.

- **synthesis**
  The `synthesis` command generates and synthesizes the FPGA kernel. Running this command will first fill in the HLS template source code based on the settings in `model.yaml`, then executes the required TCL scripts for Vitis HLS and Vivado, and copies the resulting bitstream files `.bit` and `.hwh` into the data directory. This command can be split into two separate stages using `-synthesis vitis` or `-synthesis vivado`. If no stage is specified, both stages are executed sequentially.

- **inference**
  The `inference` command initializes the FPGA kernel using PYNQ, loads the `.bit` and `.hwh` files from the data directory, and uses the inputs stored in `input.pkl`. The pre-encoder and post-encoder model parts are executed on the CPU using ONNX Runtime, while the extracted encoder slice is executed on the FPGA.

- **evaluation**
  The `evaluation` command executes the same heterogeneous inference flow as the inference command, but additionally compares the FPGA output against the ONNX reference output. This makes it possible to validate the numerical correctness of the offloaded encoder implementation and report an RMSE value. Additionally, this command will measure the performance of inference and report latency and throughput figures.

- **auxiliary commands**
  FAT also provides additional commands for inspecting and preparing models. These include viewing the ONNX model structure and running the original ONNX model without FPGA offloading. A dedicated help command provides a concise overview of the available functionality. Together, these commands support debugging and validation of the deployment flow.

Data files

All files used by FAT are located inside the specified data directory. The most important files are listed below:

- `model.onnx`
  The complete Transformer model in ONNX format.
- `model.yaml`
  Unified configuration file containing options such as the model dimensions, kernel precision settings and the number of encoder layers to offload.
- `*.bit`, `*.hwh`
  The FPGA bitstream and hardware description files generated by Vivado, used by PYNQ to configure and control the FPGA kernel.
- `input.pkl`
  A pickled Python object containing the model input data used for inference.
- `model_a.onnx`, `model_b.onnx`, `model_c.onnx`
  The sliced sub-models used during inference. Model A corresponds to the pre-encoder part, Model B contains the encoder layers offloaded to the FPGA, and Model C represents the post-encoder part.

5.9 Quantization effects

The current implementation makes use of floating-point weights and activations. A common way to increase performance and reduce resource consumption is to employ quantization, often at `INT8`, for both weights and activations. However, this can greatly affect the overall accuracy of the model. Therefore, an investigation was done on the accuracy impact of quantization on the EncReg model. It was first investigated if the model can be quantized using PyTorch since the original model is implemented in PyTorch. However, it turns out that PyTorch does not support exporting quantized model to the ONNX format. Therefore, utilizing PyTorch for quantization is not favorable. Fortunately, ONNX Static quantization is available in the ONNX Python package. ONNX provides configuration options for choosing the quantization width for both the weights and activations. Table 5.5 shows the resulting model accuracies with different quantization configurations. We can conclude that quantizing the activations has a bigger impact on the model accuracy, while quantizing just the weights has a lesser impact.

TABLE 5.5: Model prediction accuracies with different quantization levels applied. The base model has an overall prediction accuracy of 0.97.

Model activations	Model weights	Prediction accuracy
`INT16`	`INT16`	0.90
`INT16`	`INT8`	0.90
`INT8`	`INT16`	0.71
`INT8`	`INT8`	0.70

This experiment shows a very significant impact on final accuracy compared to existing literature. This could be due to the fact that the final accuracy is not calculated based on the output of this model. Rather, the model output is fed into the HDBSCAN clustering algorithm. It could be that this algorithm is very sensitive to small changes in the model output, resulting in significant impact on the final model accuracy.

There are more combinations to try for a comprehensive benchmarking, e.g., it would be interesting to see the effect of not quantizing the activations and only using quantized `INT8`

weights. Unfortunately, ONNX Static quantization does not support this configuration and implementation is more complicated. Since the aim for this work is to develop a prototype tool for easier deployment on FPGA, it was decided to not pursue this any further.

5.9.1 Half Precision Kernel

Instead of using integer quantization, the numerical precision can also be reduced by switching from 32-bit floating point (float32) to 16-bit floating point (float16). In this work, two configurations were implemented: using half precision for all values, or using half precision only for stored data (weights and activations) while keeping full precision for accumulation operations.

To make this configurable, two custom data types were introduced. The type `data_t` is used for weights, activations, and intermediate buffers, while `acc_t` is used for accumulations such as dot products, softmax computations, and layer normalization. By changing these typedefs between `float` and `half`, different precision settings can be selected without modifying the rest of the kernel.

Half precision is implemented using the `half` type from the `<hls_math.h>` library. Since all relevant arrays and computations are defined using `data_t` and `acc_t`, no structural changes to the architecture were required.

The hypothesis is that using half precision for `data_t` will significantly reduce BRAM usage, since all stored weights and intermediate buffers require fewer bits. At the same time, this should have only a minor impact on performance. Using half precision for `acc_t` is expected to further reduce resource utilization and latency, as the arithmetic units become smaller and less complex. The results and a comparison with the full-precision implementation are discussed in Section 6.1.2.

5.10 Measurement methods

Section 4.4 discusses the metrics that are needed in order to evaluate the final implementation. This section will briefly discuss the matter in which these metrics have been gathered.

5.10.1 FPGA Resource Consumption

One of the most important factors is the FPGA resource utilization. These include BRAMs, Flip Flops (FF), LUTs and DSPs. All of these are reported by Vitis HLS after synthesis of the kernel. However, the final exported kernel synthesized by Vivado might report slightly different resource utilization. Therefore, all the metrics are taken from the Utilization Report from Vivado. However, these BRAM blocks are also configurable as dual 18KB blocks. While Vitis HLS reports BRAM utilization as number of 18Kb blocks, Vivado reports it as 36Kb blocks. All BRAM usage metrics in this report are as number of 36Kb blocks.

5.10.2 Power usage

Power usage of the FPGA kernel is taken from the implementation results from Vivado. Specifically, the Total On-Chip Power metric is used.

5.10.3 Latency

Latency is measured as the time it takes for the system to produce the correct output for a given input. This includes the time required to transfer the input data to the FPGA, execute the kernel, and transfer the result back to the host.

For kernel configurations that implement multiple encoder layers, this latency corresponds to the time until the output associated with a specific input becomes available. As a result, the FPGA kernel is invoked multiple times before the output for a given input is produced, due to the internal pipelining of the accelerator.

The measurement is performed on the host using Python's high-resolution timer. Specifically, the `perf_counter()` function provided by the `time` package. The timer is started immediately before invoking the FPGA kernel and stopped after the output data has been received. Consequently, the reported latency represents the end-to-end execution time observed by the software, including all communication overheads.

5.10.4 Throughput

Throughput describes how many encoder layers can be processed per second when the accelerator is running continuously. In other words, it measures the number of encoder-layer evaluations completed per unit of time.

When multiple encoder layers are implemented inside a single FPGA kernel, the design is internally pipelined. The first few kernel invocations are therefore used to fill the pipeline (warmup), and additional invocations are required at the end to drain it (cooldown).

To obtain a representative throughput measurement, the execution times corresponding to the warmup and cooldown phases are excluded. Only steady-state kernel executions, during which one encoder layer is completed per invocation, are considered when computing the average execution time. The throughput is then calculated as the inverse of this steady-state execution time.

For kernel configurations that implement only a single encoder layer, no internal pipelining is present. In this case, there is no warmup or cooldown phase, and the throughput (in encoder layers per second) is simply the inverse of the measured latency.

5.10.5 Root Mean Square Error

The Root Mean Square Error (RMSE) is used to measure how close the FPGA output is to the reference result. The reference output is obtained by running the same encoder layer(s) with ONNX Runtime on the CPU using full 32-bit floating-point precision.

After the FPGA kernel finishes, its output is transferred back to the host and compared element-by-element with the ONNX output. If y_i^{FPGA} represents the FPGA result and y_i^{CPU} the reference result, the RMSE is calculated as

$$\text{RMSE} = \sqrt{\frac{\sum_{i=1}^{N}(y_i^{FPGA} - y_i^{CPU})^2}{N}}$$

where N is the total number of elements in the output tensor.

Both outputs are flattened into one-dimensional arrays before computing the RMSE. The resulting value gives an indication of the average numerical difference per output element.

Chapter 6

Results

This chapter presents the results of this work. It first evaluates the FPGA-based encoder accelerator in terms of resource utilization, performance, power consumption, and numerical precision. Finally, the chapter discusses the developed toolchain, including its current limitations and how well it generalizes to the deployment of another model.

6.1 Encoder acceleration evaluation

6.1.1 Experimental setup

This section describes the experimental setup used to evaluate the FPGA-based encoder acceleration. The hardware and software platforms employed in these experiments are described in Section 5.1. For all experiments, the smallest EncReg model configuration is used. The model dimensions and parameters are summarized in Table 6.1.

TABLE 6.1: The model dimensions for the EncReg model used for evaluation

Parameter	**Value**
Number of encoder layers	6
Sequence length (L)	500
Model dimension (d_{model})	32
Feed-forward dimension (d_{ff})	96
Number of attention heads	4

During inference on the ZCU102 platform, the EncReg model is partitioned as described in Section 5.4.1. Depending on the scenario, a specified number of encoder layers are offloaded to the FPGA, while any remaining non-encoder layers are executed on the host CPU using ONNX Runtime.

All measurements were obtained using a fixed set of 100 input samples. These input samples come from the REDVID dataset for which the EncReg model is trained. Each experiment begins when the first input is provided to the system and ends after all outputs have been produced. When a kernel implements multiple encoder layers, additional invocations are required to fill and drain the internal pipeline. Consequently, the number of kernel invocations may exceed the number of input samples. Latency and throughput are computed according to the methodology described in Section 5.10.

The experimental setup is evaluated under five different scenarios. These scenarios are chosen to enable a direct comparison of different configuration aspects, such as the number of implemented layers and the use of half precision.

- **Scenario A 1-layer FP32/32:** A kernel implementing a single encoder layer using full 32-bit floating-point precision for both data and accumulation.
- **Scenario B 4-layer FP32/32:** A kernel implementing four encoder layers using full 32-bit floating-point precision for both data and accumulation.
- **Scenario C 4-layer FP16/32:** A kernel implementing four encoder layers where model parameters, inputs, and outputs use 16-bit floating-point precision (FP16), while accumulation remains in FP32.
- **Scenario D 6-layer FP16/32:** A kernel implementing all six encoder layers of the EncReg model. Parameters, inputs, and outputs use FP16 precision, while accumulation remains in FP32. In this configuration, the complete encoder stack fits on the FPGA.
- **Scenario E 6-layer FP16/16:** A kernel implementing all six encoder layers using FP16 precision for parameters, inputs, outputs, and accumulation.

The scenario naming scheme follows the format *N-layer FPx/y*, where N denotes the number of encoder layers implemented within a single FPGA kernel. The notation *FPx/y* specifies the numerical precision used in the design: *FPx* indicates the floating-point precision of stored data (model parameters, inputs, and outputs), while *FPy* denotes the precision used for accumulation during arithmetic operations. For example, FP16/32 corresponds to 16-bit floating-point storage with 32-bit floating-point accumulation.

All reported metrics are collected according to the measurement methodology described in Section 5.10.

6.1.2 Resource utilization

Table 6.2 shows the FPGA resource usage for all evaluated scenarios. The results make it clear how both the number of layers and the numerical precision affect the hardware footprint.

TABLE 6.2: FPGA resource utilization for each evaluation scenario, reported as absolute resource usage as well as a percentage of total available device resources on the ZCU102 shown in parentheses.

Scenario	BRAM	LUT	FF	DSP
A 1-layer FP32/32	226.5 (25%)	30126 (11%)	33106 (6%)	241 (10%)
B 4-layer FP32/32	855 (94%)	96846 (35%)	95453 (17%)	964 (38%)
C 4-layer FP16/32	544 (60%)	98403 (36%)	98833 (18%)	964 (38%)
D 6-layer FP16/32	811 (89%)	143833 (52%)	141758 (25%)	1446 (57%)
E 6-layer FP16/16	811 (89%)	78494 (29%)	53510 (10%)	1074 (43%)

Layer count scaling

By comparing Scenario A (1-layer FP32/32) and Scenario B (4-layer FP32/32), we can observe how the resource utilization scales with the number of implemented layers. BRAM usage increases almost linearly from 226.5 to 855 blocks (3.77×). The scaling factor is similar for LUTs and FFs (3.2× and 2.88× respectively). DSP slice usage scales exactly linearly with the number of layers (4×). This suggests that the DSPs are used almost

exclusively for the encoder layers, while some FFs, LUTs, and BRAM are also consumed by miscellaneous components such as AXI interfaces or input/output streaming logic.

Overall, the resource utilization scales very predictably with increasing layer count. On the ZCU102, BRAM is the limiting factor, reaching 93% utilization. Therefore, it is not possible to fit all six layers of the EncReg model using full precision.

Reduced precision and six-layer implementations

Scenario C uses FP16 precision for parameters and activations, while keeping FP32 for accumulation. The most noticeable effect is the drop in BRAM usage, from 855 to 544 blocks (around 36% less) compared to the 4-layer FP32/32 design. A 50% reduction in BRAM usage would be expected when moving from FP32 to FP16 storage. The fact that only a 36% reduction is observed is probably because of BRAM granularity and synthesis constraints that prevent an 'ideal' reduction in BRAM usage.

In contrast, the compute-related resources barely change. The DSP count remains at 964, and LUT and FF usage even increase slightly. This is expected, since all arithmetic is still performed in FP32. The small increase in logic usage is likely caused by additional casting and mixed-precision handling between FP16 storage and FP32 computation.

Using FP16 storage also makes it possible to fit all six encoder layers on the FPGA. Scenario D (6-layer FP16/32) uses 811 BRAM blocks, staying below the device limit. Without reduced precision, this configuration would exceed the available BRAM.

When accumulation is also switched to FP16 (Scenario E), BRAM usage remains unchanged, but LUT, FF, and DSP usage drop significantly. The DSP count decreases from 1446 to 1074, and logic usage is almost halved. This shows that using FP16 for accumulation simplifies the arithmetic operations, resulting in reduced hardware complexity and lower resource utilization.

6.1.3 Performance

Table 6.3 summarizes the measured latency and throughput for all scenarios. In this context, latency is defined as the time required for a single kernel invocation, i.e., the time it takes to instantiate and execute the kernel once.

Table 6.3: Performance metrics per evaluation scenario.

Scenario	Latency (ms)	Throughput (layers/s)
A 1-layer FP32/32	105.4	9.48
B 4-layer FP32/32	105.8	37.66
C 4-layer FP16/32	106.6	37.38
D 6-layer FP16/32	105.8	56.59
E 6-layer FP16/16	50.8	117.6

For Scenarios A–D, the latency remains almost constant at around 106 ms, independent of the number of implemented layers. This means that adding more layers to the kernel does not increase the invocation time. Instead, more computation is performed within the same kernel execution window. This is expected, since the kernel operates in a pipelined manner.

However, the number of layers computed per kernel invocation increases, and thus the throughput, expressed in processed layers per second, scales with the number of instantiated layers.

For example, moving from Scenario A (1-layer FP32/32) to Scenario B (4-layer FP32/32) increases throughput from 9.48 to 37.66 layers/s, which is almost exactly a factor of four. The same behavior can be observed for Scenario D (6-layer FP16/32), where throughput increases further to 56.59 layers/s while latency remains unchanged. This confirms that the multi-layer kernel benefits from internal pipelining and concurrent execution of different layers once steady state is reached.

Switching only the storage precision to FP16 (Scenario C) has no noticeable impact on performance. Latency and throughput remain nearly identical to the 4-layer FP32/32 design. This is expected, since accumulation and arithmetic are still performed in FP32.

A significant change is observed in Scenario E (6-layer FP16/16). Here, latency drops to 50.8 ms, roughly half of the other configurations. As a result, throughput increases to 117.6 layers/s. This shows that using FP16 for accumulation not only reduces resource usage but also shortens the computation time. The simpler arithmetic reduces the critical path and allows faster execution of the kernel.

Overall, performance scales mainly with the number of layers per kernel, while latency per invocation remains constant as long as FP32 accumulation is used. Only when switching to full FP16 computation does the latency itself decrease significantly.

6.1.4 Power

Table 6.4 summarizes the estimated static and dynamic power consumption, as well as the resulting efficiency in layers per second per watt. The static power remains almost constant across all scenarios at around 0.73W, which is expected since it mainly depends on the device itself rather than the specific kernel configuration. The differences between scenarios are therefore primarily caused by changes in dynamic power.

TABLE 6.4: Estimated power consumption and efficiency per evaluation scenario.

Scenario	**Static (W)**	**Dynamic (W)**	**Total (W)**	**Efficiency (layers/s/W)**
A 1-layer FP32/32	0.727	3.383	4.110	2.31
B 4-layer FP32/32	0.745	5.143	5.888	6.40
C 4-layer FP16/32	0.735	4.217	4.952	7.55
D 6-layer FP16/32	0.742	4.997	5.739	9.86
E 6-layer FP16/16	0.738	4.497	5.235	22.47

Dynamic power increases with the number of instantiated layers in full precision. For example, moving from Scenario A (1-layer FP32/32) to Scenario B (4-layer FP32/32) increases dynamic power from 3.383W to 5.143W. This reflects the additional switching activity caused by the replicated compute units.

When switching to FP16 data storage (Scenario C), dynamic power decreases compared to the 4-layer FP32/32 configuration, even though throughput remains almost identical. A similar effect can be observed when comparing Scenario D and Scenario E: using FP16 for accumulation reduces dynamic power but this time significantly increasing throughput.

As a result, efficiency improves noticeably with deeper and lower-precision configurations. Scenario A achieves only 2.31 layers/s/W, while Scenario D reaches 9.86 layers/s/W. The highest efficiency is obtained in Scenario E (6-layer FP16/16), with 22.47 layers/s/W.

Overall, combining multiple layers in a single kernel already improves energy efficiency due to better hardware utilization. Reducing precision further increases efficiency by slightly lowering dynamic power but significantly increasing throughput. The results show

that full FP16 computation provides the best trade-off between performance and power consumption in this setup.

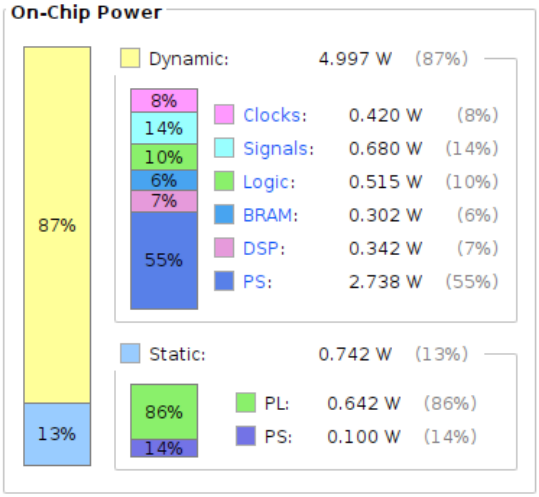


FIGURE 6.1: Detailed breakdown of power usage as reported by Vivado for Scenario D

Dynamic power consumption strongly depends on the amount of utilized hardware resources (BRAM, logic units, etc.). As shown in Table 6.2, Scenario D has generally the highest overall resource utilization among the 4-layer configurations.

In addition to reporting static and dynamic power, Vivado also provides a more detailed breakdown of the power distribution, as shown in Figure 6.1. As can be seen, BRAM contributes by far the largest share, accounting for approximately 55% of the dynamic power consumption. This suggests that if power reduction is the main objective, decreasing BRAM usage would likely be the most effective approach.

6.1.5 Numerical precision

Table 6.5 shows the numerical deviation of each scenario, measured as RMSE with respect to the FP32 reference implementation. The first row in Table 6.5 reports the RMSE between the FP32 reference implementation and its output converted to FP16 format. This serves as a useful lower-bound reference for the expected deviation when reducing precision.

TABLE 6.5: Numerical accuracy per evaluation scenario measured as RMSE as well as final model output accuracy. The original model accuracy is 97.94%.

Scenario	RMSE	Model Accuracy
Original converted to FP16	0.000278	97.78%
A 1-layer FP32/32	0.000242	97.88%
B 4-layer FP32/32	0.000316	97.81%
C 4-layer FP16/32	0.004706	97.81%
D 6-layer FP16/32	0.002749	97.97%
E 6-layer FP16/16	0.009717	96.53%

As expected, the full-precision configurations (Scenarios A and B) show only very small deviations. The observed differences are on the order of 10^{-4} and can be attributed

to minor floating-point implementation differences between the FPGA and the software reference, rather than precision loss.

When switching to FP16 storage (Scenarios C and D), the RMSE increases noticeably, reflecting the additional rounding introduced when weights and activations are stored in reduced precision. However, the magnitude of the error remains small relative to the overall output range, indicating that the model is relatively robust to reduced storage precision.

Interestingly, Scenario D (6-layer FP16/32) shows a slightly lower RMSE than Scenario C (4-layer FP16/32), despite involving more layers with reduced precision. In principle, one would expect the numerical error to increase with the number of layers due to accumulated rounding effects. Given the limited number of evaluated input samples (100), this discrepancy is most likely due to statistical variation.

When accumulation is also performed in FP16 (Scenario E), the RMSE increases further. This is expected, as rounding errors are now introduced not only during storage but also during intermediate arithmetic operations. Nevertheless, the overall deviation remains within a moderate range, suggesting that FP16 computation is still numerically stable for this model configuration.

In addition to RMSE, Table 6.5 reports the impact on the final model output accuracy after the clustering step of the EncReg model. Despite the increase in RMSE for reduced-precision configurations, the corresponding drop in accuracy remains limited. This indicates that the model's final predictions are relatively insensitive to small numerical perturbations in intermediate representations. Only in Scenario E, where both storage and accumulation use FP16, a more noticeable degradation in accuracy is observed.

6.1.6 Summary

Overall, the results show clear trade-offs between resource utilization, power consumption, performance, and numerical precision. Increasing the number of layers per kernel improves throughput without increasing latency per invocation, thanks to internal pipelining. However, this comes at the cost of higher resource usage, with BRAM being the main limiting factor. Reducing storage precision to FP16 significantly lowers BRAM usage and enables the full six-layer model to fit on the FPGA, while having only a minor impact on performance. Switching accumulation to FP16 further reduces LUT, FF, and DSP utilization and nearly halves the latency, resulting in a substantial improvement in throughput and energy efficiency. This performance gain comes with an increase in numerical error, although the observed RMSE values remain relatively small. Overall, full FP16 computation provides the highest efficiency and best hardware utilization, while FP32 accumulation offers slightly better numerical stability at the expense of performance and resource usage.

6.2 Toolchain

One of the main goals of this work is to develop a prototype toolchain that can be used to deploy pre-existing Transformer models on FPGAs, with a particular focus on particle tracking applications. Evaluating whether this goal has been fully achieved is not straightforward. Section 2.3 describes two existing Transformer-based models for particle tracking: EncReg and EncCla. However, throughout much of the development of this work, and particularly throughout development of the toolchain, only the EncReg models have been used.

At its current stage, the tooling supports the (semi-)automatic acceleration of EncReg models. In principle, it should be possible to apply the same steps discussed in the previous

chapter to an EncCla model as well. Ideally, only minimal manual intervention should be required to accelerate (parts of) an EncCla model on an FPGA.

This section first discusses several known limitations and missing features of the current toolchain implementation. These are aspects that were identified during development but were not prioritized for implementation within the scope of this work. The section then describes an attempt to deploy and accelerate an EncCla model using the developed workflow, in order to assess how well the toolchain generalizes beyond the EncReg models used during development.

6.2.1 Missing features

Although the current toolchain already supports the main deployment flow for the evaluated EncReg models, a number of useful features were intentionally left out of the present implementation. These were not considered essential for demonstrating the feasibility of the overall approach, but they would improve the flexibility and usability of future versions of the toolchain.

Off-chip parameter storage

At present, the toolchain assumes that all model parameters are stored in on-chip buffers. This is a sensible design choice, since it generally provides the best performance in terms of latency and throughput. However, it also means that the size of deployable models is strongly limited by the available BRAM resources on the FPGA.

For prototyping and exploratory evaluation, it would be useful to support alternative memory strategies in which all or part of the model parameters are stored off-chip. This would reduce on-chip memory pressure and make it possible to deploy larger models than are currently feasible with the existing implementation. The trade-off is that accesses to off-chip memory would likely increase latency and reduce throughput. Even so, such a feature would make the toolchain more flexible and more suitable for early-stage design exploration.

Automatic extraction of model dimensions

The current toolchain is configured through a `model.yaml` file, which allows the user to define model-specific information required by the workflow. Among other things, this includes the dimensions of the model parameters. While this configuration mechanism is already practical and relatively easy to use, some of this information should in principle be derived automatically from the ONNX model graph itself.

Adding this kind of automatic extraction would not fundamentally change the workflow, since the user would still need to provide some model-specific configuration in more complex cases. Nevertheless, it would improve usability by reducing the amount of manual setup required and by removing configuration steps that are repetitive and prone to small errors. As such, it would be a useful improvement for future versions of the toolchain.

Automatic reporting of implementation metrics

Another practical limitation of the current workflow is that implementation metrics such as resource utilization and power consumption are not reported automatically by the toolchain itself. Instead, the user must open Vivado and extract these results manually after synthesis and implementation have completed.

This interrupts the otherwise largely automated workflow and makes evaluation less convenient. Since these metrics are central to the assessment of FPGA deployments, it would be beneficial for future versions of the toolchain to collect and report them directly, for example by extending the TCL scripts used during implementation.

6.2.2 EncCla deployment

To assess how well the developed workflow transfers beyond the EncReg model family, the toolchain was also applied to the EncCla model from the TrackFormers work [14]. This serves as a practical test of the generality of the current implementation. The aim is not only to determine whether deployment is possible, but also to identify which parts of the workflow already generalize well and which parts still rely on assumptions specific to EncReg.

Model conversion

The first step in the deployment process was to convert the EncCla model from PyTorch to ONNX. Strictly speaking, this conversion is not part of the core toolchain, since the workflow takes an ONNX model as input. However, because the EncCla model was originally provided in PyTorch format, this step was necessary before the rest of the workflow could be applied.

In practice, the conversion was straightforward and closely resembled the earlier conversion of the EncReg model. This is a positive result, as it suggests that the scope of the toolchain could be extended in the future to accept both PyTorch and ONNX models, with PyTorch models being converted to ONNX internally as a preprocessing step.

Graph structure and parameter extraction

The first major difficulty appeared during manual inspection of the resulting ONNX graph. Although the overall model structure remained similar, there were clear differences in graph structure and naming. These differences may be caused by small changes in the original PyTorch implementation, but regardless of their cause, they have an important consequence: the node names and parameter names no longer match the assumptions built into the toolchain.

For example, the corresponding layer normalization nodes are named differently in the two ONNX graphs:

```
EncReg:  /encoder/layers.0/norm2/LayerNormalization
EncCla:  node_layer_norm
```

As a result, the original tooling could no longer automatically locate and extract the required parameters.

To continue the evaluation, the codebase was extended so that the user can configure the relevant node and matrix names for a given ONNX graph. This requires a manual inspection step, which can be performed using external tools such as Netron. The `model.yaml` file was therefore extended with additional configuration options that allow the user to define how the required matrices should be located and extracted from the model graph. This improves the generality of the workflow, but it also shows that the current implementation is not yet fully model-agnostic.

Model splicing

A similar issue arose during model splicing. Since the splicing logic also depends on node names, the original implementation could not be reused directly for EncCla. To address this, the toolchain was modified so that the user can provide a custom implementation of the `splice()` function. This function takes the number of encoder layers to extract as input and returns the names of the corresponding input and output nodes for the generated submodels.

It would likely be possible to move this functionality into the `model.yaml` configuration as well, which could make the workflow more consistent. At the same time, keeping it as a function gives the user more flexibility in defining how the model should be partitioned.

Potential for assisted configuration

An additional observation is that these configuration tasks are difficult to automate in a conventional way, since there is no single fixed standard that defines how model graphs should be structured or named. At the same time, once the graph has been inspected, the required mapping and configuration work is often repetitive and structured. This suggests that large language models could be useful for generating the required configurations based on a manually inspected model graph. Although this would not remove the need for user inspection entirely, it could make the process simpler and reduce the amount of manual effort required.

Implementation results

Once the model had been successfully converted, configured, and spliced, the remaining steps proceeded without issues. The HLS template was populated, and synthesis, place-and-route, and bitstream generation were run using Vitis and Vivado. This part of the process completed successfully on the first attempt, which indicates that these later implementation stages are relatively robust once the model-specific configuration has been resolved.

Chapter 7

Discussion & Future Work

7.1 HLS Templates

An important design decision in this work was to implement the accelerator using High-Level Synthesis (HLS) rather than a Register-Transfer Level (RTL) design in a language such as Verilog or VHDL. HLS has a lower barrier to entry, since it is based on C/C++ and is generally faster to develop with. In contrast, RTL offers much finer control over the generated hardware and therefore provides more opportunities for highly optimized designs.

Although HLS is easier to work with during development, the relevance of this distinction is less clear from the perspective of the end user. The goal of this project is to provide a toolchain that automates FPGA deployment for users with limited hardware expertise. Since the implementation of the accelerator is already abstracted behind the toolchain, the user does not directly interact with either HLS or RTL. In principle, this means that an RTL-based implementation could be used internally to improve performance or efficiency without increasing the complexity of the user experience.

At the same time, HLS proved highly valuable during the development process. The kernel architecture went through several iterations, and the higher abstraction level made it much easier to modify, test, and extend the design. This was especially beneficial in an exploratory project such as this one, where rapid iteration was important.

Because the current deployment flow already produces RTL as an intermediate output from Vitis HLS, the HLS synthesis stage could in principle be replaced by a hand-written RTL implementation. A promising direction for future work would therefore be to extend the toolchain with support for RTL templates alongside the existing HLS templates. This would allow developers to choose between a more accessible and flexible HLS-based approach and a potentially more efficient RTL-based implementation.

7.2 Viability of Pre-Trained Models

This work focused on deploying pre-trained Transformer models to FPGA hardware without modifying the original training process. The models were not retrained, and no hardware-aware optimization techniques such as quantization-aware training or pruning were applied. The aim was to evaluate how far an existing model can be pushed on FPGA hardware without requiring changes to the machine learning workflow.

From a performance perspective, this approach has clear limitations. The implemented kernels mainly rely on floating-point representations, using FP32 and optionally FP16. As shown in Chapter 6, memory usage quickly becomes the dominant bottleneck, and the

achieved latency remains far above that of a high-end GPU. This is not unexpected, since many FPGA accelerators reported in the literature rely on lower-precision arithmetic such as INT8 in order to reduce memory footprint and increase computational parallelism.

Using full 32-bit floating point is expensive in terms of both storage and compute resources. More BRAM is required to store weights and intermediate activations, while the available hardware budget leaves less room for parallel processing. If raw performance is the primary objective, it is therefore more sensible to design and train the model with the constraints of the target FPGA platform in mind from the beginning.

A more practical route toward high-performance deployment would be to train models that are already adapted to reduced precision or reduced complexity. Techniques such as quantization-aware training, structured pruning, or architectural simplification can significantly improve efficiency, especially for memory-bound designs such as the one presented in this thesis.

Nevertheless, deploying pre-trained models still offers clear benefits. It lowers the barrier for experimentation and makes it possible to evaluate model feasibility on FPGA hardware without redesigning the training pipeline. For toolchain development, rapid prototyping, and early-stage research, this flexibility is valuable. It also enables existing research models to be tested on FPGA platforms with relatively little additional effort.

Overall, the results show that deploying pre-trained FP32-based Transformer models to an FPGA is feasible and useful for exploration and development. However, when maximum performance or energy efficiency is the main objective, the model itself should ideally be designed and trained with FPGA constraints in mind.

7.3 Support for Additional Model Components

The current toolchain is mainly focused on Transformer encoder layers. This was a deliberate choice, since it kept the scope of the project manageable while still covering a meaningful real-world use case. At the same time, it means that the current implementation only supports a limited subset of machine learning models and layer types.

One obvious direction for future work is to extend the toolchain beyond encoder-style computation. In particular, support for decoder layers would be a valuable next step. Many modern Transformer-based models rely on both encoder and decoder blocks, or use decoder-only architectures. Adding support for these structures would therefore make the framework applicable to a much broader range of models.

Beyond decoder layers, the toolchain could also be extended to support more generic neural network building blocks. Examples include fully connected layers, alternative attention variants, or other commonly used operations that do not fit neatly into the current encoder template. This would make the framework less dependent on one specific model structure and improve its usefulness in more varied deployment scenarios.

At the moment, the generated accelerator works best when the model follows a fairly standard and predictable structure. Models with unusual layer arrangements, custom operations, or irregular graph structures are much harder to map automatically to the existing templates. This is one of the trade-offs of the current template-based approach: it works well for models that match the expected pattern, but becomes less effective as the model architecture becomes more diverse.

Thus, the current toolchain shows that automated FPGA deployment is feasible for Transformer encoder models. A natural next step would be to broaden this support to include other model components, such as decoder layers and more general neural network layers, such that the framework becomes more applicable to a wider class of machine

learning models.

7.4 Integration into Larger Systems

At the moment, the FAT toolchain is mainly used through a command-line interface (CLI) and configuration files. For the current scope of this work, this is sufficient, since the workflow is largely offline and the input data for the models that was used to test and verify the implementation is already known before execution. This makes it easy to run the different stages of the toolchain in a controlled and reproducible way.

In a real deployment, however, the situation will be different. Input data may be received live from another system rather than loaded from a fixed file, and the generated output may need to be passed on directly to other software components for further processing. In that setting, a purely CLI-based workflow is less practical, especially for the actual execution of the accelerator.

A useful direction for future work would therefore be to make the runtime part of the toolchain accessible as a software library in addition to the existing command-line interface. This would allow the accelerator to be integrated more easily into a larger application or processing pipeline, where data can be provided programmatically and results can be consumed immediately by downstream components.

Not every part of the toolchain needs this kind of integration. Steps such as synthesis, bitstream generation, and model splicing are naturally batch-oriented tasks and are well suited to a CLI-based workflow. In contrast, the runtime stage has a stronger need for programmatic access, since this is the part most likely to be embedded into a complete system. Supporting both usage modes would therefore make the toolchain more practical without changing its overall design.

7.5 FPGA performance in comparison to GPU and CPU

The original intent was to carry out a more thorough performance comparison between the FPGA, the ARM CPU on the ZCU102, and GPU execution. This is an important step in determining whether this type of automatic deployment is worth pursuing in the first place. In practice, however, this turned out to be difficult to do properly within the available time. In particular, it was not possible to obtain ARM power measurements that were accurate and consistent enough to report with confidence. Depending on the measurement method and conditions, the observed ARM power draw varied between about 4 W and 22 W, which is too large a spread to support a fair efficiency comparison. Because of that, no strong conclusions can be drawn about whether the FPGA is actually better than the ARM CPU in terms of overall power efficiency.

That said, the performance results are still useful for providing context. Running the full six-layer encoder on the ARM Cortex-A53 resulted in an average end-to-end latency of 130.7 ms, corresponding to 45.88 layers/s. For the FPGA, the most representative scenario, Scenario D, reached 56.59 layers/s with an estimated total on-chip power of 5.74 W. So although the FPGA does not achieve lower end-to-end latency for a single sample, it does show slightly higher steady-state throughput once the pipeline is filled. This makes the FPGA result promising, especially since the kernel was not optimized as aggressively as it could have been, as that was not necessarily the main goal of this project.

For the GPU, the result reported in the original TrackFormers work is much faster, with a latency of 2.4 ms on an NVIDIA A100. The A100 also has a maximum TDP of 300 W, which already shows that it operates in a very different hardware class. So while the

GPU is clearly much quicker, it is also designed for a completely different performance and power envelope. For that reason, the GPU result is mainly useful as a point of reference rather than as a fair head-to-head comparison.

Overall, this comparison should be interpreted with care. The GPU is clearly much faster, but it also belongs to a very different hardware category, especially in terms of power usage. The FPGA result is closer to the ARM CPU result and looks encouraging, but the available measurements are not strong enough to claim that the FPGA is definitively better in either performance or efficiency. A more careful measurement setup, especially for CPU power, would be needed before making stronger claims. Still, it is encouraging that the FPGA achieves higher throughput than the ARM CPU.

Chapter 8

Conclusion

The High-Luminosity Large Hadron Collider (HL-LHC) will significantly increase event complexity and data rates, putting strong pressure on real-time particle tracking systems. Transformer-based tracking models such as EncReg and EncCla have shown promising algorithmic performance on simulated tracking data. However, bringing such models into hardware environments relevant to high-energy physics requires careful attention to latency, memory constraints, numerical precision, and deployment workflow. This thesis investigated how Transformer encoder layers for particle tracking can be mapped to FPGA hardware in a structured and practical way.

A first contribution of this work is the systematic literature review on FPGA-based Transformer inference. The survey shows several recurring themes: most implementations accelerate selected components rather than full models, memory management is often the main design constraint, and high efficiency is usually achieved through reduced precision and hardware-aware optimization. These observations are especially relevant for HL-LHC tracking, where low latency and tight resource limits are essential. The findings of the survey directly informed the architectural decisions and evaluation strategy used in this thesis.

Building on these insights, a semi-automated deployment workflow was developed in the form of the FPGA Accelerator for Transformers (FAT). The goal of this toolchain is not to compete with handcrafted RTL designs, but to make FPGA-based experimentation with existing Transformer models more accessible to researchers with limited FPGA experience. FAT supports model splicing to isolate encoder layers, parameter extraction, automated HLS template generation, synthesis using Vitis and Vivado, and runtime integration through PYNQ. By combining these steps into a single workflow, FAT lowers the barrier to exploring FPGA acceleration of Transformers.

To validate the approach, an encoder-layer accelerator for the EncReg tracking model was implemented and deployed on the AMD Zynq UltraScale+ MPSoC ZCU102 platform. The evaluation confirms several trends identified in the literature. Most importantly, the design is strongly memory-bound: BRAM utilization scales almost linearly with the number of encoder layers and becomes the main limiting factor well before compute resources are exhausted.

The performance evaluation shows that multi-layer kernels improve throughput and power efficiency through pipelining, but the achieved latency remains much higher than that of high-end GPU execution. A broader comparison with ARM CPU and GPU execution was intended, since this is important for determining whether this type of automatic deployment is worthwhile in practice. However, within the available time it was not possible to carry out this comparison as thoroughly as intended. In particular, ARM power

measurements were too inconsistent to support a reliable and accurate efficiency comparison. The available results still provide useful context: the GPU result is clearly much faster, but also belongs to a very different hardware and power class, while the FPGA result is closer to the ARM CPU result and shows promising throughput even without extensive kernel optimization. Still, these measurements are not strong enough to support firm conclusions about whether the FPGA is definitively better than ARM or GPU execution in performance or efficiency.

The investigation of reduced numerical precision further highlights the difficulty of deploying particle tracking models under hardware constraints. While half-precision storage reduces resource usage and power consumption, more aggressive quantization introduces substantial deviations in model output. These deviations propagate through the downstream clustering stage and reduce final tracking accuracy. For HL-LHC applications, where physics performance is critical, such losses may not be acceptable without retraining or model adaptation.

The research questions introduced in Chapter 1 can now be answered.

1. **What is the current state of the art in Transformer inference on FPGAs?**

   The literature shows that FPGA-based Transformer inference is mainly limited by memory bandwidth and on-chip storage rather than raw compute. As a result, many successful designs focus on reducing data movement and keeping weights on-chip as much as possible.

   Quantization, pruning, and structured sparsity are central techniques in this design space. Reduced precision lowers BRAM usage and bandwidth pressure, while sparsity is only useful when the hardware is explicitly designed to exploit it.

   This also means that deploying existing full-precision Transformer models without compression is difficult on FPGAs. Such models are usually designed for GPU execution, where large off-chip memory and high bandwidth are available. On FPGA platforms, limited on-chip storage and memory bandwidth quickly become the dominant constraints. Overall, the state of the art, as well as the experience from this project, suggest that models trained or adapted with FPGA constraints in mind are generally more effective and more scalable than models that are simply ported from GPU-oriented implementations.

2. **How do hardware-oriented model adaptations—such as reduced numerical precision and quantization—affect the tracking accuracy of the selected model?**

   There is a clear trade-off between hardware savings and tracking performance. When applying ONNX static quantization after training, the effect is severe. The baseline model achieves a prediction accuracy of 0.97, but this drops to around 0.90 with INT16 quantization and further to about 0.70–0.71 when using INT8 activations. This is a substantial loss in tracking quality.

   Switching from full precision (FP32) to half precision (FP16) storage within the kernel is much less disruptive. The RMSE increases compared to FP32, but remains relatively small, on the order of 10^{-3} to 10^{-2}. At the same time, FP16 significantly reduces memory usage and improves performance, making it possible to fit larger models on the FPGA.

   In short, post-training quantization using ONNX static quantization causes too much accuracy loss in this setup, while half-precision floating-point storage provides a much better balance between efficiency and acceptable numerical stability.

3. **What functionality is required from a toolchain to deploy an existing Transformer model on an FPGA in a largely automated fashion?**

   To make FPGA deployment accessible to users without FPGA design experience, the toolchain must hide most of the low-level complexity. In practice, this means it should support model slicing, parameter extraction, kernel configuration, synthesis, and runtime integration in as automated a way as possible.

   In this work, the FAT toolchain demonstrates several of these core capabilities. The model must first be available in a known and supported format, currently ONNX. The toolchain must then be able to split the model into parts so that selected components, such as encoder layers, can be offloaded to the FPGA. Parameters must be extracted and reformatted for the FPGA kernel, while the synthesis flow using Vitis and Vivado must be scriptable so that the user does not need to manually work through FPGA design tools. Finally, the runtime interface should support smooth data transfer between CPU and FPGA, together with clear feedback and error handling when something goes wrong.

   At the moment, the FAT toolchain still has to be run across separate platforms. Synthesis must be done on a machine with the AMD tools installed, while kernel execution happens on the FPGA platform itself. Moving files and artifacts between these systems is still fairly manual, and a more streamlined workflow would make the tool significantly easier to use.

   Overall, the experiments show that semi-automation is possible, but not yet fully effortless. Some manual configuration and basic understanding of the model structure are still needed. A practical FPGA toolchain must therefore balance automation, flexibility, maximum performance, and clear user feedback.

Overall, this thesis presents a structured approach to mapping Transformer encoder layers for particle tracking onto FPGA hardware. The main contribution is not the development of a highly optimized accelerator, but the creation of a practical bridge between machine learning model development and FPGA deployment. The FAT workflow makes it possible to partially offload existing Transformer models to hardware and evaluate them on a real FPGA platform without requiring extensive low-level hardware expertise.

In its current form, the toolchain does not achieve the maximum performance that is possible on FPGA. State-of-the-art FPGA accelerators typically rely on hardware-aware model design, quantization, sparsity exploitation, and careful architectural tuning. In contrast, FAT focuses on deploying pre-existing models that were originally developed for GPU execution. Such models are generally not structured with FPGA constraints in mind, which naturally limits the achievable performance and efficiency.

That does not reduce the practical relevance of the workflow. A toolchain like FAT can still be valuable during early design and prototyping phases, where feasibility, resource usage, latency, and numerical effects need to be evaluated before committing to a more specialized implementation. In the context of HL-LHC tracking, where algorithmic performance and hardware constraints must be considered together, this kind of structured exploration is useful.

In summary, this thesis shows that deploying Transformer-based tracking models on FPGA hardware is feasible, but strongly constrained by on-chip memory capacity and numerical precision. The experimental results make these trade-offs visible on a real platform, while the FAT toolchain provides a practical workflow for exploring them without requiring deep low-level hardware expertise. Although the resulting accelerator is not intended as a

fully optimized final solution, it forms a concrete step toward making FPGA-based acceleration of Transformer tracking models more accessible, reproducible, and better understood within the high-energy physics community.

Bibliography

[1] G Aad and B Abbott. Software Performance of the ATLAS Track Reconstruction for LHC Run 3. *Computing and Software for Big Science*, 8(1), 3 2024. doi:10.1007/s41781-023-00111-y.

[2] Inc. Advanced Micro Devices. PYNQ, 2025. URL: https://github.com/Xilinx/PYNQ.

[3] AMD. Zynq UltraScale+™ MPSoC ZCU102. URL: https://www.amd.com/en/products/adaptive-socs-and-fpgas/evaluation-boards/ek-u1-zcu102-g.html.

[4] AMD. Vitis HLS, 2022. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vitis/vitis-hls.html.

[5] AMD. Vivado, 2022. URL: https://www.amd.com/en/products/software/adaptive-socs-and-fpgas/vivado.html.

[6] Sabrina Amrouche, Laurent Basara, Paolo Calafiura, Victor Estrade, Steven Farrell, Diogo R Ferreira, Liam Finnie, Nicole Finnie, Cécile Germain, Vladimir Vava Gligorov, Tobias Golling, Sergey Gorbunov, Heather Gray, Isabelle Guyon, Mikhail Hushchyn, Vincenzo Innocente, Moritz Kiehn, Edward Moyse, Jean-François Puget, Yuval Reina, David Rousseau, Andreas Salzburger, Andrey Ustyuzhanin, Jean-Roch Vlimant, Johan Sokrates Wind, Trian Xylouris, and Yetkin Yilmaz. The Tracking Machine Learning Challenge: Accuracy Phase. pages 231–264, 2020. doi:10.1007/978-3-030-29135-8{_}9.

[7] B. Angelucci, R. Fantechi, G. Lamanna, E. Pedreschi, R. Piandani, J. Pinzino, M. Sozzi, F. Spinella, and S. Venditti. The FPGA based trigger and data acquisition system for the CERN NA62 experiment. *Journal of Instrumentation*, 9(1), 2014. doi:10.1088/1748-0221/9/01/C01055.

[8] G Apollinari, O Brüning, T Nakamoto, and L Rossi. Chapter 1 High Luminosity Large Hadron Collider HL-LHC. Technical report.

[9] Arjan Blankestijn. Thesis Source Code, 2026. doi:10.5281/zenodo.19543647.

[10] Arjan Blankestijn, Uraz Odyurt, and Amirreza Yousefzadeh. TrackCore-F: Deploying Transformer-Based Subatomic Particle Tracking on FPGAs, 2025. doi:10.48550/arXiv.2509.26335.

[11] Arjan Blankestijn, Uraz Odyurt, and Amirreza Yousefzadeh. Recent Developments in Transformer Inference Deployment on FPGA Platforms: A Survey. 2026.

[12] James Brooke, Emyr Clement, Maciej Glowacki, Sudarshan Paramesvaran, and Jeronimo Segal. LHC Triggers using FPGA Image Recognition. 3 2025. URL: http://arxiv.org/abs/2503.09428.

[13] Woohong Byun, Jongseok Woo, and Saibal Mukhopadhyay. Hardware-friendly Hessian-driven Row-wise Quantization and FPGA Acceleration for Transformer-based Models. pages 1–6, 2024. doi:10.1145/3665314.3670806.

[14] Sascha Caron, Nadezhda Dobreva, Antonio Ferrer Sánchez, José D Martín-Guerrero, Uraz Odyurt, Roberto Ruiz de Austri Bazan, Zef Wolffs, and Yue Zhao. TrackFormers: In Search of Transformer-Based Particle Tracking for the High-Luminosity LHC Era. 2024. URL: http://arxiv.org/abs/2407.07179.

[15] Sascha Caron, Nadezhda Dobreva, Antonio Ferrer Sánchez, José D Martín-Guerrero, Uraz Odyurt, Roberto Ruiz de Austri Bazan, Zef Wolffs, and Yue Zhao. Efficient ML-Assisted Particle Track Reconstruction Designs. *EPJ Web of Conferences*, 337, 2025. doi:10.1051/epjconf/202533701299.

[16] ONNX Runtime developers. ONNX Runtime. https://onnxruntime.ai/, 2021.

[17] Congpeng Du, Seok Bum Ko, and Hao Zhang. Energy Efficient FPGA-Based Binary Transformer Accelerator for Edge Devices. *Proceedings - IEEE International Symposium on Circuits and Systems*, pages 1–5, 2024. doi:10.1109/ISCAS58744.2024.10558631.

[18] FastML Team. fastmachinelearning/hls4ml, 2025. URL: https://github.com/fastmachinelearning/hls4ml, doi:10.5281/zenodo.1201549.

[19] R Frohwirth. APPLICATION OF KALMAN FILTERING TO TRACK AND VERTEX FITTING. Technical report, 1987.

[20] Qingyu Guo, Jiayong Wan, Songqiang Xu, Meng Li, and Yuan Wang. HG-PIPE: Vision Transformer Acceleration with Hybrid-Grained Pipeline. 2024. URL: http://arxiv.org/abs/2407.17879.

[21] Yuhao Ji, Chao Fang, and Zhongfeng Wang. BETA: Binarized Energy-Efficient Transformer Accelerator at the Edge. *Proceedings - IEEE International Symposium on Circuits and Systems*, 2:1–5, 2024. doi:10.1109/ISCAS58744.2024.10558636.

[22] Zhixing Jiang, Dennis Yin, Yihui Chen, Elham E Khoda, Scott Hauck, Shih-Chieh Hsu, Ekaterina Govorkova, Philip Harris, Vladimir Loncar, and Eric A Moreno. Low Latency Transformer Inference on FPGAs for Physics Applications with hls4ml. 2024. URL: http://arxiv.org/abs/2409.05207.

[23] Moritz Kiehn, Sabrina Amrouche, Paolo Calafiura, Victor Estrade, Steven Farrell, Cécile Germain, Vava Gligorov, Tobias Golling, Heather Gray, Isabelle Guyon, Mikhail Hushchyn, Vincenzo Innocente, Edward Moyse, David Rousseau, Andreas Salzburger, Andrey Ustyuzhanin, Jean-Roch Vlimant, and Yetkin Yilnaz. The TrackML high-energy physics tracking challenge on Kaggle. *EPJ Web of Conferences*, 214:06037, 2019. doi:10.1051/epjconf/201921406037.

[24] Nikoletta Koilia and Christoforos Kachris. Hardware Acceleration of LLMs: A comprehensive survey and comparison. 2024. URL: http://arxiv.org/abs/2409.03384.

[25] Bingbing Li, Santosh Pandey, Haowen Fang, Yanjun Lyv, Ji Li, Jieyang Chen, Mimi Xie, Lipeng Wan, Hang Liu, and Caiwen Ding. FTRANS: Energy-efficient acceleration of transformers using FPGA. *ACM International Conference Proceeding Series*, 2020. doi:10.1145/3370748.3406567.

[26] Richie Li and Sicheng Chen. Design and Implementation of an FPGA-Based Hardware Accelerator for Transformer. 2025. URL: http://arxiv.org/abs/2503.16731.

[27] Zuohao Li. Energy Efficient FPGA-Based Accelerator for Dynamic Sparse Transformer. *2024 13th International Conference on Communications, Circuits and Systems (ICCCAS)*, pages 7–12, 2024. doi:10.1109/ICCCAS62034.2024.10652850.

[28] Tianyang Lin, Yuxin Wang, Xiangyang Liu, and Xipeng Qiu. A Survey of Transformers. 6 2021. URL: http://arxiv.org/abs/2106.04554.

[29] Tianheng Ling, Chao Qian, and Gregor Schiele. Integer-only Quantized Transformers for Embedded FPGA-based Time-series Forecasting in AIoT. *Proceedings - 2024 IEEE Annual Congress on Artificial Intelligence of Things, AIoT 2024*, pages 38–44, 2024. doi:10.1109/AIoT63253.2024.00017.

[30] Ze Liu, Yutong Lin, Yue Cao, Han Hu, Yixuan Wei, Zheng Zhang, Stephen Lin, and Baining Guo. Swin Transformer: Hierarchical Vision Transformer using Shifted Windows. *Proceedings of the IEEE International Conference on Computer Vision*, pages 9992–10002, 2021. doi:10.1109/ICCV48922.2021.00986.

[31] Kyle Marino. ME-ViT : A Single-Load Memory-Efficient FPGA Accelerator for Vision Transformers.

[32] Uraz Odyurt. REDVID Sample events. URL: https://virtualdetector.com/redvid/.

[33] Uraz Odyurt, Stephen Nicholas Swatman, Ana Lucia Varbanescu, and Sascha Caron. Reduced Simulations for High-Energy Physics, a Middle Ground for Data-Driven Physics Research. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 14833 LNCS:84–99, 2024. doi:10.1007/978-3-031-63751-3{_}6.

[34] Krzysztof T. Pozniak. FPGA-based, specialized trigger and data acquisition systems for high-energy physics experiments, 2010. doi:10.1088/0957-0233/21/6/062002.

[35] Ye Qiao, Zhiheng Chen, Yian Wang, Yifan Zhang, Yunzhe Deng, and Sitao Huang. COBRA: Algorithm-Architecture Co-optimized Binary Transformer Accelerator for Edge Inference. 2025. URL: http://arxiv.org/abs/2504.16269.

[36] Anders Ryd and Louise Skinnari. Tracking Triggers for the. pages 1–26, 2020.

[37] T. A. Collaboration. The ATLAS Experiment at the CERN Large Hadron Collider, Journal of Instrumentation. *Journal of Instrumentation*, 2008.

[38] A C Team. The four main LHC experiments. URL: https://cds.cern.ch/record/40525.

[39] Cenk Tüysüz, Carla Rieger, Kristiane Novotny, Bilge Demirköz, Daniel Dobos, Karolos Potamianos, Sofia Vallecorsa, Jean Roch Vlimant, and Richard Forster. Hybrid quantum classical graph neural networks for particle track reconstruction. *Quantum Machine Intelligence*, 3(2), 2021. doi:10.1007/s42484-021-00055-9.

[40] Samuel Van Stroud, Philippa Duckett, Max Hart, Nikita Pond, Sébastien Rettie, Gabriel Facini, and Tim Scanlon. Transformers for Charged Particle Track Reconstruction in High Energy Physics. 12 2025. URL: http://arxiv.org/abs/2411.07149http://dx.doi.org/10.1103/md46-yqgd, doi:10.1103/md46-yqgd.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in Neural Information Processing Systems*, 2017-Decem(Nips):5999–6009, 2017.

[42] Saiqun Wang. Efficient FPGA-Based Transformer Accelerator Using In-Block Balanced Pruning. *2024 13th International Conference on Communications, Circuits and Systems (ICCCAS)*, pages 18–23, 2024. doi:10.1109/ICCCAS62034.2024.10651591.

[43] Bingyi Zhang, Rajgopal Kannan, Carl Busart, and Viktor K Prasanna. VisionAGILE: A Versatile Domain-Specific Accelerator for Computer Vision Tasks. *IEEE Transactions on Parallel and Distributed Systems*, 35(12):2405–2422, 2024. doi:10.1109/TPDS.2024.3466891.