Performance of ML-Based Bandwidth Compression on FPGAs

ALEKO LILIUS
MASTER´S THESIS
DEPARTMENT OF ELECTRICAL AND INFORMATION TECHNOLOGY
FACULTY OF ENGINEERING | LTH | LUND UNIVERSITY

Performance of ML-Based Bandwidth Compression on FPGAs

Aleko Lilius
al2810li-s@student.lu.se

Department of Electrical and Information Technology
Lund University

Supervisors: Fredrik Edman & Alexander Ekman

Examiner: Erik Larsson

June 24, 2024

Printed in Sweden
Tryckeriet i E-huset, Lund

Abstract

This thesis investigates the integration of machine learning (ML)-based compression on Field-Programmable Gate Arrays (FPGAs) to enhance bandwidth compression of data, a crucial aspect in scientific research where large amounts of data are produced in real-time. The compression tool Baler, utilizing autoencoders for ML-based compression, is designed to handle scientific data efficiently. By combining the adaptability of ML models with the computational efficiency of FPGAs, this thesis aims to evaluate the performance of Baler's bandwidth compression. The thesis work reveals that smaller models can effectively fit onto the FPGA, resulting in a throughput increase of 16.9 times compared to a CPU in a desktop computer. This significant improvement demonstrates the potential of FPGA-accelerated ML solutions. Key factors influencing optimal FPGA performance, including model size, precision levels, and clock period, were identified. This thesis lays a foundation for further developing hardware implementation of the Baler algorithm, suggesting that the convergence of ML and FPGA technology holds significant potential for enabling more efficient hardware-accelerated ML solutions.

Acknowledgements

I would like to begin by expressing my gratitude to my supervisor, Fredrik Edman, at the Department of Electrical and Information Technology, for his invaluable guidance and insightful feedback throughout the entirety of my master's thesis.

I am also profoundly thankful to the Baler team for allowing me to undertake my master's thesis with them. Special thanks are due to Alexander Ekman, my supervisor within the team, whose insightful suggestions and unwavering support significantly contributed to the success of this project. Additionally, I wish to convey my gratitude to Maayan Tamari from the Baler team, whose invaluable assistance enhanced the quality and depth of this thesis.

I would also like to extend my thanks to my examiner Erik Larsson, at the Department of Electrical and Information Technology, for his expertise on this subject and for taking the time to review my work.

Popular Science Summary

Machine learning is a well-known concept that you probably have heard a lot about. Have you ever wondered how scientists are harnessing the power of machine learning to revolutionize data compression? This thesis explores integrating machine learning-based compression with Field-Programmable Gate Arrays (FPGAs) to enhance bandwidth compression, which is crucial for scientific research where vast amounts of data are generated rapidly in real-time.

Imagine your average streaming service buffering your videos 17 times faster because the data transferred is reduced to 1/17 of its original size before being sent. This is precisely the potential that FPGA-accelerated machine learning offers for scientific data processing. In the future, there is a possibility that data collected in real-time will be compressed in real-time as well.

But it's not just about compressing more data in less time. The research has to be thoroughly evaluated. By assessing the performance of advanced compression algorithms like Baler on FPGAs, key factors influencing optimal performance, such as model size, precision levels, and clock period, have been identified. These findings not only advance our understanding of FPGA-accelerated machine learning but also hold significant potential for advancing computational methods and enabling more efficient, adaptable solutions in various fields of science and technology.

So, where are the findings from this thesis most applicable, and when is bandwidth compression most needed? Many research centers worldwide, such as CERN in Switzerland and MAX IV in Sweden, are experiencing an abundance of produced data. The amount of data produced is currently getting out of hand, and to save time, resources, and data storage, they require some sort of data reduction solution. By compressing data in real-time as it is produced, the amount of data that can be saved for storage will increase. This potentially enables further research at CERN, MAX IV, and other research centers that would benefit from this.

This thesis lays a foundation for further developing the hardware implementation of the Baler algorithm and highlights the exciting intersection of machine learning and FPGA technology. Continuing to explore the possibilities of this innovative approach brings us one step closer to unlocking the full potential of data-driven discovery in the digital age.

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction

This chapter serves as an introduction to the subject, laying the groundwork and disposition for the subsequent chapters of the thesis. Comprehensive background material will be covered, as well as prior work within this field. In addition to providing a wide context, readers will be acquainted with the motivation behind the thesis and the problem description of the work, along with the research questions.

1.1 Background

The huge amounts of data produced from modern scientific research, such as the research conducted at the Large Hadron Collider (LHC) at CERN in Switzerland, are quickly becoming a problem. In late September last year, 2023, the research center celebrated its capacity to store one exabyte of data, which translates to one million terabytes. The data center's storage capacity was less than a third of this just five years ago and only about 6% of this ten years ago, which makes this a great achievement [42]. However, the pressing need for more storage only indicates that the need for storage is steadily increasing, exponentially. For instance, upgrades to the LHC are expected to enable the accumulation of an order of magnitude more data than currently possible by 2026 [14]. Beyond this, the ability to transfer all this data is essential and with many scientific collaborations all over the world, the need for efficient compression of the data is just as important.

Researchers at the Division of Particle Physics at Lund University have tackled the issue of imminent data storage needs with the development of a data compression tool named Baler [8]. This tool employs lossy compression algorithms, derived through machine learning methods. The compression models can be customized for the user's dataset, improving decompression and reconstruction performance [10]. A lossy compression is typically constrained by the requirement for specific input data to a designated compression algorithm to achieve high compression ratios with minimal data loss. Nevertheless, a well-trained ML-based lossy compression algorithm can effectively reduce data while maintaining the ability to decompress it with minimal loss of information. In scientific contexts, such as at CERN and MAX IV in Sweden, where massive amounts of data are produced, some data loss is acceptable. With this in mind and the fact that lossy compression often provides a higher compression ratio than lossless, Baler is well-suited for this kind of data.

These research facilities face challenges not only in compressing their data to a manageable size but also in achieving this in real time. For example, MAX IV conducts experiments with CMOS detectors that produce several gigabytes of data every second [35] and would benefit from data size reduction from the moment of collection. This could be accomplished by performing suitable bandwidth compression, which aims to enhance data throughput by compressing the data before sending it to storage. While compression algorithms for bandwidth aim to reduce data size, their main priority is often to balance compression ratio and computational efficiency.

Therefore, the goal of this master thesis is to deploy Baler on Field-Programmable Gate Arrays to potentially accelerate the ML-models responsible for compression and to measure the actual bandwidth compression Baler could achieve. FPGAs offer great potential due to their high parallelism, enabling them to execute multiple operations simultaneously. Compression algorithms often involve repetitive and independent tasks, making them well-suited for parallel processing. FPGAs can exploit this parallelism to compress multiple data elements concurrently, leading to faster compression rates [26]. Given this advantage, the implementation of FPGAs could prove significant for accelerating Baler's most computationally intensive tasks during compression.

1.2 Scope & Limitations

At present, there is no apparent method for generating the high-level synthesis (HLS) required to translate Python-created ML models into FPGA implementations. The FPGA market is dominated by a few major hardware manufacturers, namely Xilinx (AMD), Altera (Intel), and Lattice Semiconductor (Nvidia collaborator) [40]. However, the research on running ML-models on FPGAs is limited, and third-party applications in this area are scarce. Consequently, there is a limitation on developed software applications that provide sufficient support for both high-level programming and the transition to the hardware description language (HDL) used for FPGAs.

Recently, there has been significant scientific research conducted in the field of FPGA-acceleration for ML-models. There is, however, no obvious choice for which approach to take regarding the transition from Python to HLS or HDL. The software chosen to convert the ML-models produced by Baler is the Python package hls4ml [21]. Developed by individuals at CERN, hls4ml is designed to interpret and translate ML algorithms for implementation on FPGAs [19]. However, it is important to note that hls4ml at this stage in time is not widely adopted within this niche research field, partly because the research in itself is new. This limitation might pose a challenge in the practical aspects of implementation. With fewer prior use cases explored and less general experience with the tool, its more detailed utilization may prove to be even more challenging.

Another difficulty that may arise during the course of this project is the need for a proper dataset. Generally, the datasets available for testing during the implementation phase are limited. This originates from the necessity of obtaining consent from data providers such as scientists at CERN, MAX IV, or similar institutions. The relevance of the data's structure also plays a significant role in the selection process as the available data might not always fit the model. Additionally, it is worth noting that comparing Baler's performance across different types of data is somewhat impractical. For meaningful comparisons with other similar tools, using the same dataset they employed would be more appropriate.

Finally, within the scope of compression, this thesis will prioritize exploring the implementation and viability of bandwidth compression over decompression performance. This approach allows for a more comprehensive examination of bandwidth compression in conjunction with FPGAs, rather than concentrating solely on optimizing compression models for decompression performance.

1.3 Problem Definition

Based on the background and current limitations, the following research questions have been devised:

1. To what extent can the deployment of Baler's compression algorithms on FPGAs improve bandwidth compression?
2. What are, if any, the limitations of deploying Baler's compression algorithms on FPGAs?

1.4 Previous Work

In the following section, previous research and developments relevant to the field of FPGA acceleration for ML-models will be reviewed. This exploration provides essential context for understanding the current state of the field and informs the approach taken in this study.

1.4.1 SZ: Fast Error-Bounded Lossy Compression

In 2016, Sheng Di and Franck Cappello introduced a fast error-bounded lossy compression algorithm for High-Performance Computing (HPC) data [16]. The development of this algorithm was prompted by the demand for enhanced data compression solutions in the field of extreme-scale HPC applications, where data processing is becoming a bottleneck [16]. Squeeze, or SZ for short, utilizes error-bounded compression with a best-fit curve-fitting model implementation to achieve a significant increase in compression ratio. Error-bounded compression is defined as maintaining the error of compressed data points, D', within a bound of $[D - \Delta, D + \Delta]$, where D represents the original data points and Δ is the error specified by the user. The algorithm examines each data point to determine if it can be predicted by a best-fit curve-fitting model. If the prediction falls within the specified error boundary, the data point is replaced using a two-bit code that indicates the model type. Subsequently, the leftover unpredicted data is compressed based on its binary representation, and the final result of the entire process is losslessly compressed using Gzip compression [23]. In worst-case scenarios, the compression ratio improves by about 2x, while for most cases, it attains an order of magnitude enhancement compared to existing compression techniques [16].

However, the time it took to compress the data was comparable to that of other solutions, indicating a potential need for further development of their compression throughput.

1.4.2 GhostSZ: Transparent FPGA-Accelerated Lossy Compression

In 2019, Xiong et al. developed an FPGA framework designed to run the SZ data compression algorithm at line rate [51]. It was the first implementation that ran SZ on an FPGA and was developed in order to decrease the time it took to compress data. The framework is divided into five steps, which are offline configuration, linearization, curve-fitting, error-bound quantization (improvement made since the original version of SZ), and lossless Gzip compression. This follows the workflow structure of SZ but adds the steps to calculate the curve-fitting models and the Gzip compression on the FPGAs instead, accelerating the process.

The modification resulted in a notable speedup, ranging from 9.5x to 80x with single-core execution compared to a single FPGA pipeline. Additionally, there was a marginal improvement in compression ratio compared to the original SZ. This improvement is attributed, in part, to the high number of repetitions in data that GhostSZ benefits from. The parallelization aspect of the test results demonstrated that the performance of GhostSZ scaled linearly with the number of pipelines used. In contrast, the thread count of the CPU when running SZ in parallel with OpenMP did not show linear scaling, likely due to memory contention. In conclusion, the overall performance significantly increased with the incorporation of FPGAs, and the parallelization aspect exhibited promising potential with multiple pipelines.

1.4.3 WaveSZ: FPGA-Implemented Error-Bounded Lossy Compression

In 2020, Sheng Di and Franck Cappello et al. introduced an improved version of the SZ algorithm called WaveSZ [45], with the overarching aim to implement the compression on FPGAs using HLS. The name is derived from the combination of adapting the wavefront memory layout, which mitigates data dependency during prediction and utilizing the SZ-based algorithm. This version incorporates co-optimization modifications in both hardware and algorithm design. The algorithm unfolds in four primary stages: wavefront preprocessing, Lorenzo prediction (a further improvement from the curve-fitting models previously used) adhering to user-specified error bounds, linear-scaling quantization, and Gzip compression. To ensure that the compression error consistently complies with the error-bound specified by the user, it is imperative that the neighbouring values employed in data prediction are the decompressed values rather than the original data values. This constraint poses a challenge for SZ when attempting to leverage the pipelining features in FPGAs since the prediction of data cannot occur until its preceding points have been compressed.

However, the use of a wavefront memory layout alleviates this dependency and enables a more optimized utilization of FPGAs. With this in mind and the incentive that FPGAs can exploit parallelization with an inherently better pipelining process and loop-unrolling structure, in the form of a higher number of logic units, than that of CPUs, promoted the idea to make an FPGA-based design. This was realized through HLS written in C/C++. The outcome was a significant improvement in compression throughput, with a speedup ranging from 6.9x to 8.7x compared to the original SZ running on a CPU. The major breakthrough, however, was the throughput enhancement of 5.8x compared to existing FPGA-based SZ designs.

1.4.4 Fast Inference of Neural Networks in FPGAs for Particle Physics

Research on implementing neural networks on FPGAs was conducted by Javier Duarte et al. in 2018, complementarily resulting in the companion compiler package hls4ml [17]. This work examines the implementation of neural networks on FPGAs by mapping out resource usage and latency across different networks and their associated hyperparameters. The primary objective of the study was to identify which neural networks most effectively work in conjunction with FPGAs, considering resource and latency constraints within the context of particle physics research. One experiment conducted was the conversion of a neural network designed for jet substructure classification into an FPGA implementation. The hls4ml package is designed to simplify this process, providing a solution to bridge the gap between machine learning models and FPGA hardware. Even though the neural network used in their experiment does not strive for data compression they do mention compression of the network, using L_1 regularization, as part of reducing the FPGAs resource usage. Thereby, the built-in functionality of neural networks and data compression with FPGAs were identified as a side product.

For instance, the Compact Muon Solenoid (CMS) collaboration at CERN successfully implemented a neural network for hardware detection of physics signatures on FPGAs using hls4ml [53]. Although the model used was very small, the results were promising. This thesis aims to explore more thoroughly the built-in functionality of neural networks and data compression with FPGAs, as well as the usage of hls4ml.

1.5 Disposition

This section outlines the thesis structure, giving a brief description of the chapters with an emphasis on their respective contributions.

- **Introduction** The introduction provides a general overview of the thesis, offering background information on the work to be undertaken and summarizing prior research in the field. Additionally, it establishes the scope of limitations, defines the problem, and formulates the research questions that guide this thesis.

- **Theoretical Background** This chapter delves deeper into the background research supporting this thesis, offering a more detailed exploration of ML-based data compression and FPGAs.

- **Methodology** The methodology covers the rudimentary workflow of the thesis, including the tools employed during the research. It features the reasoning behind the use of Baler together with hls4ml in order to explore the potential of incorporating FPGAs into the work process.

- **Implementation** The implementation chapter highlights every major stage of development, offering a comprehensive overview of the project's progression. Every pivotal element that left a mark on the overall solution is thoroughly recorded and reviewed. By delving into each crucial part, this chapter provides valuable insights into the project's evolution, emphasizing the decisions made and their impact on the final outcome.
- **Discussion** This chapter undertakes an in-depth analysis and discussion of the implementation results, while also exploring the underlying reasons for any uncertainties encountered. Within this section, the author's own reflections are presented, alongside considerations for potential future directions and enhancements of the project. Additionally, this chapter also concludes the thesis with the project's key findings and insights, providing a concise summary of the research outcomes and their implications.

Chapter 2

Theoretical Background

This chapter introduces the background and theory relevant to the technologies explored in this thesis, focusing on ML-based data compression and FPGAs. It will also examine certain interchanges between the two being done today.

2.1 Machine Learning-based Data Compression

The fundamental aim of data compression is to minimize the number of bits needed to represent meaningful information, a task achieved through various methods tailored to different types of data.

Data compression can be categorized into two main types: lossless compression and lossy compression. The lossless approach compresses and decompresses data without any loss, while the lossy method involves a certain degree of data loss during compression and/or decompression. Each type has its own set of advantages and disadvantages [2], making them suitable for different tasks. Notably, the primary advantage of lossless compression is its ability to reduce data size without sacrificing fidelity. To achieve this reduction, the lossless approach leverages statistical redundancy [48], such as data duplication and cross-correlation. However, due to the intrinsic entropy of the data [41], this approach often encounters limitations on the achievable compression ratio. This ratio is typically calculated using the formula below, where D represents the size of the original data and D' represents the size of the compressed data.

$$Compression\ Ratio = \frac{D}{D'}$$

While lossless compression algorithms typically achieve a compression ratio of 2:1, meaning the compressed size is half of the original, some lossy compression algorithms surpass this considerably, reaching ratios of 100:1 or even 200:1 for specific types of data [33]. This incentive has driven considerable research on lossy data compression algorithms, resulting in compression algorithms such as JPEG [22][3] which has been a dominating lossy image compression for decades. It has been continuously developed since 1992 [31] and has undergone a series of improvements since then. It remains one of the most widely used image compression algorithms today, setting a standard for lossy image compressions to follow.

However, JPEG might be suitable for everyday compression needs, but it is not as applicable to scientific data or high-performance computing (HPC) data due to its impracticality for handling extremely large datasets. This algorithm lacks the compressing ratio required for such vast amounts of data. As discussed in Section 1.1, the storage capacity at the CERN research center has reached one exabyte of data. In such scenarios, a general compression ratio of 20:1 [18] as in JPEGs case is insufficient, making it more advantageous to employ specialized compression algorithms capable of achieving compression ratios of up to 200:1 in the best-case scenario [33].

Instead, Autoencoders (AEs) have been considered a more suitable approach for this task, primarily because they are neural networks purposely trained to replicate their input as closely as possible in their output [24].

2.1.1 Autoencoders in Data Compression

The composition of AEs usually consists of two main components, namely, the encoder function $h = f(x)$ and the decoder function $r = g(h)$. The encoder is responsible for mapping the input data x to a hidden layer or latent space h, which serves as a code representing the input. The decoder then reconstructs the original input data r as closely as possible from this code, which is illustrated in Figure 2.1. The encoder-decoder architecture of AEs facilitates the transformation of input data into a compressed representation, enabling efficient storage and retrieval of essential features, a process essential for tasks such as data compression [9].

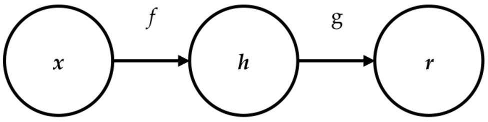

Figure 2.1: General process of an AE. Involves the encoder function *f*, responsible for mapping the input data x to a latent space h, and the decoder function g, which maps h to a reconstructed output r.

The primary goal of an AE is not just to perfectly learn and reproduce the input x as $g(f(x)) = x$ everywhere, that would not make them particularly useful. On the contrary, they are intentionally designed to be incapable of learning to simply copy the input, as they might not capture meaningful information if doing so. AEs are instead typically constrained or restricted in a manner that allows them to copy input data only approximately. This restriction may involve copying only input data that closely resembles the training data. Because of this, they are forced to prioritize which aspects of the input to copy and often end up learning useful properties of the data. In other words, the model is compelled to focus on the most relevant features or characteristics of the input data during the copying process. The efficiency of data compression with AEs arises from this specific ability to capture and represent essential features or patterns in the input data in a compact form [9].

AEs are trained to encode the input into a compressed or hidden representation [11] that retains the essential features of the data, through the encoder function $h = f(x)$. The intentional imperfection in copying during training ensures that the AE does not merely memorize the input data but learns to represent it in a more general and compact way. This compressed representation usually requires less space than the original data while retaining the critical information.

In order to obtain this information from an AE, the dimensionality of the latent space h can be constrained to be smaller than the dimensionality of the input data x. With this restriction, the encoder learns to prioritize the most important features of the data and encapsulate the essence of the data as h. The learning process can be briefly described as minimizing a loss function $L(x, g(f(x)))$, wherein the model adjusts its parameters to reduce the discrepancy between x and actual values from *g(f(x))*, ultimately enhancing its predictive capabilities. The loss function may vary between different AE-implementations, but the most commonly used is the mean squared error (MSE) or a cross-entropy solution.

2.1.2 Baler

The Baler algorithm, introduced in Section 1.1, employs the aforementioned concept of AEs for compression. In a proof-of-concept study dedicated to compressing high-energy physics (HEP) data, Baler demonstrated a lossy compression ratio of 6:1 [10], surpassing the 4:1 ratio achieved by the lossless compression format Gzip [23]. Baler also achieved a noteworthy ratio of 88:1 for computational fluid dynamics (CFD) data, showcasing the diverse compression potentials across different types of data.

This subsection aims to provide insight into Baler's compression structure, the current limitations of the Baler algorithm as well as the future prospects of its ML-based data compression capabilities. First and foremost, the use of AEs for Baler's compression algorithm is based on the notion that they can be tailored to the user's dataset, making it significantly more adaptable compared to alternative approaches. The ability to train the model on specific or similar data to achieve positive results makes the ML aspect of Baler highly desirable. Figure 2.2 shows how Baler conveys the functionality of AEs in more detail. The input is first encoded into a latent space with reduced dimensionality, representing the compressed data. Subsequently, the compressed representation is decoded to generate an output with the same dimensionality as the input data.

Example Methodology

In the study of Baler's compression performance of HEP data, the Baler team employed a neural network comprising three encoding layers with nodes organized as 200, 100, and 50, along with three decoding layers featuring nodes in the reverse order – 50, 100, and 200. The input and output had a dimensionality of 24. This was done using the Python framework *PyTorch* [38] and the model was trained using a loss function defined as,

$$L_{total} = (1 - \beta)L_{reco} + \beta L_{sparse}$$

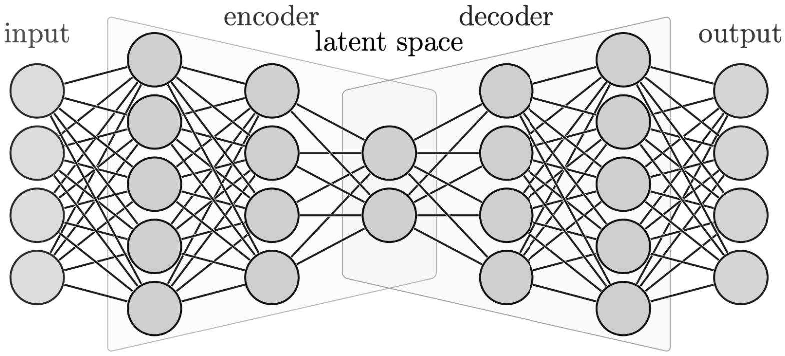

Figure 2.2: Encoding and decoding process of Baler. Axel Gallén, original by Izaak Neutelings. (License: CC BY-SA 4.0).

The parameter β serves as a loose hyperparameter, dictating the influence of each term on the overall net loss. The error term L_{reco} represents the reconstruction loss, quantified as the MSE summed over every mini-batch m, going through the variables n for each data entry. This is illustrated by the formula below, where X denotes the model input vector and $\hat{X}$ represents the corresponding model reconstruction.

$$MSE = \frac{1}{n} \sum_{j=1}^{m} \sum_{i=1}^{n} (X_i - \hat{X}_i)_j^2$$

Moreover, the L_{sparse} variable implements L_1-type regularization on the model, aiming to induce sparsity in the AE weights as such: $L_{sparse} = \sum_i |w_i|$. This regularization helps mitigate overhead generated by auxiliary files in the AE process, where weights and biases are saved separately along with additional metadata for running the *PyTorch*-generated model. These have been proven to be negligible in the HEP example, as they did not exceed sizes above 1 MB while the total file size is about 117 MB, but for future considerations, they should be mentioned.

The Baler team has chosen *NumPy* [37] arrays as the file format for both input and output. This choice is driven by factors such as compatibility with *PyTorch* and the otherwise widespread use of these arrays in numerous scientific domains. This decision is further dedication to the adaptable nature that Baler aims to cultivate.

Limitations

The current limitations of Baler, as explored in this thesis, primarily revolve around bandwidth compression rather than its existing compression capabilities, mentioned in Section 2.1.2. Bandwidth compression, measured in compression throughput (bytes/s), is a crucial factor and often as important as the compression ratio when compression is performed on data in real-time as it is produced. This is called online compression and requires the model to train on a different dataset with similar characteristics and still produce a good result. Currently, Baler only employs offline compression, which is the compression of data after it has been collected, where the model has trained on that specific dataset and learned its characteristics.

Understanding the advantages and limitations of both online and offline compression techniques is crucial for optimizing compression performance and addressing the specific requirements of different applications. The adaptability of a compression algorithm significantly improves with enhanced computational efficiency, which might make it applicable in a broader range of real-world scenarios. For instance, Baler could greatly benefit from applying its compression on FPGAs, since it would pave the way for possible online compression as well. This will be further investigated in the thesis, in alignment with Research Question nr 1.

Future Prospects

As discussed in Section 1.4, previous work has demonstrated that ML-based data compression algorithms can achieve accelerated performance through FPGA implementation, leading to enhanced bandwidth compression. Therefore, integrating FPGAs into Baler's current workflow is believed to result in enabling its bandwidth compression.

2.2 Field-Programmable Gate Arrays

FPGAs, as the name implies, are programmable devices and were initially developed to reduce manufacturing costs and production time of other integrated circuits [12]. For Application-Specific Integrated Circuits (ASICs), the conventional development process can take months and involves substantial overhead costs ranging from about \$20,000 to \$200,000, making it hard to produce only a few cards at once [12]. In contrast, FPGA prototypes can be produced at approximately \$100 each in just a few minutes [12]. It is their innate ability to be programmable to fit a specific purpose that makes them not only useful for replacing ASICs but also for prototyping different hardware solutions and realizing them in a short time. The ability to discard an initial version of a design and replace it with a new one can prove useful in a lot of areas where changes occur frequently.

The general architecture of FPGAs consists of an array of configurable logic blocks (CLBs), I/O pads, and routing channels [20], as shown in Figure 2.3. The CLBs serve as a foundational element within an FPGA, offering essential logic and storage capabilities to accommodate the requirements of a specific application design. They are mainly composed of lookup tables (LUTs) and flip flops (FFs) to save the state at each clock cycle, which is illustrated in Figure 2.4. LUTs, which can be described as programmable truth tables, significantly reduce computational time compared to traditional logic circuits by allowing the implementation of various boolean functions in a flexible and configurable manner. This adaptability is crucial in optimizing the performance for specific tasks, as designers can tailor the LUT configurations to match the unique computational needs of their applications. Furthermore, the FFs provide vital memory elements, enabling the storage and synchronization of intermediate data during the execution of complex

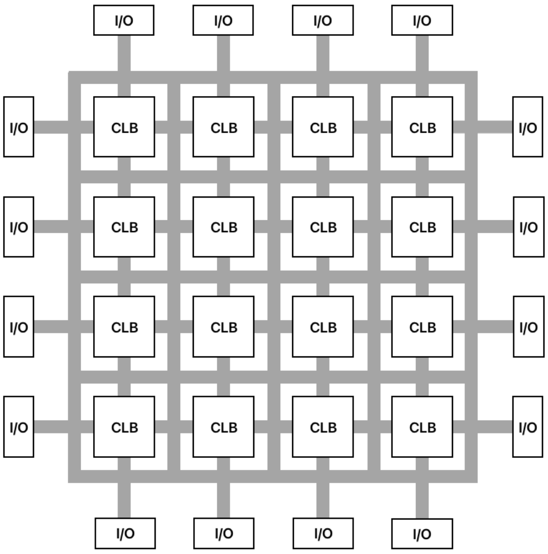

Figure 2.3: General overview of FPGA architecture.

algorithms. This combination of LUTs and FFs within CLBs empowers FPGAs to efficiently process diverse tasks ranging from signal processing to accelerating the computational speed of ML-based algorithms [13].

Current-generation FPGAs include more complex CLBs capable of multiple operations with a single block; CLBs can combine for more complex operations such as multipliers, registers, counters, and even digital signal processing (DSP) functions.

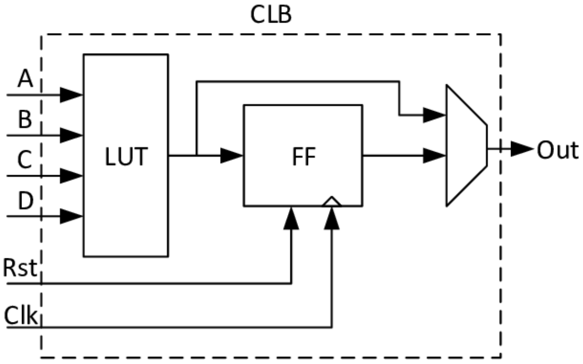

Figure 2.4: Common structure of a CLB.

The structure of FPGAs can be viewed as a network of logic blocks, which can be used either independently or combined into larger operational modules. They exhibit true parallelism, allowing them to perform different tasks simultaneously and autonomously. This inherent parallel nature makes FPGAs highly flexible and exceptionally useful for handling exceedingly difficult and complicated computations. For instance, FPGA parallelism can be leveraged when certain expensive multiplications need to be made. An example of this process is illustrated in Figure 2.5, where each part of a larger operation is handled in a pipelining fashion by different logic blocks, forwarding their output as input for the subsequent logic block. This results in a fourfold increase in throughput, from 0.25 instructions per clock cycle to 1 instruction per clock cycle, showcasing the advantages that FPGAs' parallel nature can achieve.

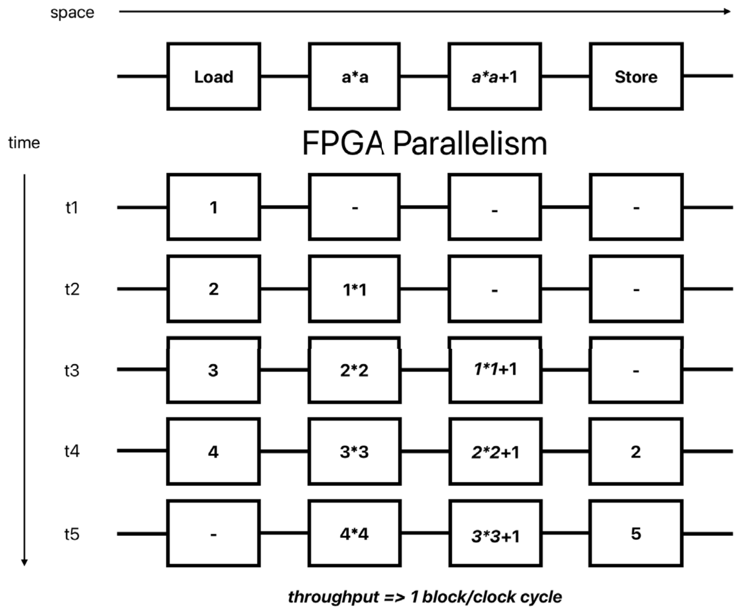

Figure 2.5: Example of FPGA parallelism.

The adaptability of FPGA boards not only ensures futureproofing [49] but also contrasts with the limitations posed by specific structures, as in the case of ASICs. The mitigation of the possibility for future changes in a predefined structure, while suitable for certain applications, can prove challenging for various dynamic and evolving scenarios. In industries where rapid technological advancements and adaptability are essential, such as telecommunications and artificial intelligence, the flexibility inherent in FPGAs becomes a key advantage. This capability allows systems to evolve and incorporate changes seamlessly, ensuring their relevance and efficiency over time.

Transitioning from the hardware-centric advantages of FPGAs, the programming aspect also plays a pivotal role in harnessing their capabilities. Programming FPGAs involves the use of HDLs such as Verilog or VHDL, allowing designers to specify the functionality and interconnections of the logic blocks within the FPGA. These languages provide a level of abstraction that facilitates the design and simulation of complex circuits. Moreover, advancements in HLS tools have streamlined the programming process by allowing developers to express their algorithms in higher-level languages like C or C++. HLS tools automatically translate these high-level descriptions into hardware implementations, providing a more accessible entry point for software engineers and accelerating the development cycle of FPGA-based applications.

Hardware Description Language - HDL

HDL is a specialized programming language used in digital design to model and describe the behaviour and structure of electronic systems, particularly digital circuits, and systems [36]. HDLs, such as Verilog [44] and VHDL [34], provide a means for engineers to specify the functionality, interconnections, and timing requirements of electronic components. These languages allow designers to create detailed and accurate representations of hardware designs, facilitating simulation, verification, and synthesis processes. HDLs are crucial in the development of complex digital systems, such as FPGAs, providing a systematic and precise way to capture the logic and functionality of hardware components at various abstraction levels, from higher-level behavioural descriptions to lower-level structural details.

However, the work conducted during this thesis will not attempt to bring forth a solution in pure HDL code but rather in HLS.

High-Level Synthesis - HLS

In contrast to HDL, HLS provides the possibility to implement solutions on FPGAs without extensive knowledge of the FPGA architecture and technology [15]. It is a design methodology in digital electronics that enables the transformation of high-level programming languages, such as C or C++, into hardware descriptions suitable for implementation on FPGAs. HLS tools, such as Vivado or Intel Quartus, automate the process of translating algorithmic descriptions into hardware, providing a bridge between software development and hardware design. This approach simplifies and accelerates the design process, allowing software engineers to focus on algorithmic aspects while automatically generating efficient hardware implementations. HLS enhances productivity, facilitates design exploration, and contributes to the broader goal of making FPGA design more accessible to a wider range of developers.

The hls4ml tool will aid the generation of HLS projects from the ML-models conceived with Baler. This approach aims to accelerate the development process and make the project less dependent on detailed knowledge about HLS and HDLs.

Chapter 3
Methodology

This chapter presents an overview of the work process employed in this thesis, as well as the essential tools utilized for this project. It will specifically delve into additional information about the Baler algorithm, the hls4ml package, and the Vivado program, which plays a crucial role in the FPGA implementation and testing process.

3.1 Baler Compression

Baler, as a compression tool, provides users with the ability to assess the feasibility of compressing various types of scientific data using AEs. The inherent constraining nature of AEs makes them useful for compressing data and efficiently learning its features. When provided with the correct instructions, they have been designed to yield excellent results, achieving a higher compression ratio than other conventional compression methods.

For first-time users, Baler provides straightforward instructions on how to operate the training, compression, and decompression of the tool. Users have the option to choose from predefined models based on the dataset in need of compression, which can be useful for those with lesser knowledge about ML-based data compression. However, in order to utilize Baler to the fullest and to reach greater results, custom-fitting a model to the specific characteristics of the provided dataset is recommended. This proves especially beneficial for offline data, where the model has been trained on a specific dataset [52] and optimized for compressing that data. For online data compression, a future goal of Baler, the model must be trained on data similar to the target data to discern the most important characteristics when compressing and decompressing the data.

The Baler project is written in Python and leverages the innate functionality of the numerous ML-packages developed for this programming language [32]. Additionally, the Python ecosystem allows for effortless data handling and manipulation to achieve a desirable model. The primary Python packages used for the Baler algorithm include PyTorch, TensorFlow, NumPy and Matplotlib.

3.2 Hls4ml

The hls4ml tool is a companion compiler package specifically developed for the purpose of converting ML-models to HLS [17], aligning well with the objectives of this thesis. Currently in development, the tested version available is 0.8.1. Originally conceived as a complementary tool during research on the inference of deep neural networks in FPGAs within particle physics, it was crafted by individuals at CERN. The tool originates from the ambition to simplify the rapid prototyping of ML algorithms, particularly for physicists, allowing them to assess both firmware feasibility and physics performance without requiring extensive Verilog/VHDL experience. The developers at CERN recognize the critical role of FPGAs in this context, emphasizing hls4ml's urgency for these purposes.

It can be a useful tool for bridging the gap between Python programming and FPGA programming, for those who are not familiar with the latter.

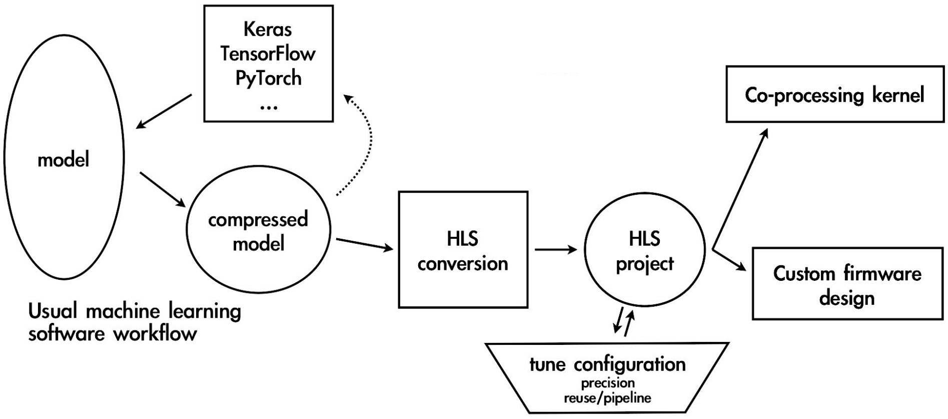

Figure 3.1: Standard hls4ml workflow, fetched from [27].

The workflow of hls4ml is illustrated in Figure 3.1. The standard procedure involves taking a model created in Python, using frameworks like Keras, TensorFlow, or PyTorch, and converting it into an HLS project. This project can then be further tuned and configured within an FPGA integrated development environment (IDE), such as Vivado [6] or Intel Quartus [29], depending on the manufacturer of the board.

The tool is developed in Python and serves as a supporting package within any project requiring the functionality of converting ML-models into HLS projects. This approach has the distinct advantage of a closely tied structure to the current Python environment that most ML applications already possess, resulting in less overlay than if it were its own standalone application. However, a considerable disadvantage is that the tool requires a computer with an FPGA-integrated development environment (IDE) installed, along with the correct licenses, to function properly. This means that the entire work chain needs to be on the same machine. Additionally, the tool only supports the Linux operating system, potentially requiring users to allocate additional resources to acquire new machines for this purpose.

3.3 Vivado

Vivado is an IDE for programming and implementing solutions on physical FPGA boards. It offers a wide range of functionalities, many of which are beyond the scope of this thesis. Vivado was chosen over Intel Quartus because the FPGA provided by the institution, the ZCU104 Evaluation Board [7], is manufactured by Xilinx. This FPGA is technically designated as XCZU7EV-2FFVC1156 MPSoC. This part number provides specific information about the FPGA's features. For instance, the "-2" denotes a speed grade of 2 on a scale from 1 to 3, indicating a mid-range speed grade. A mid-range speed grade typically balances performance and power consumption, with this FPGA achieving a maximum frequency of up to 600 MHz [50], depending on the actual implementation.

The main functionalities to be explored include the HLS compiler, the Vivado simulator, and the Vivado Intellectual Property (IP) integrator. The HLS compiler translates C and C++ code into register transfer-level (RTL) designs. The Vivado simulator is a simulation tool supporting both functional and timing simulations for HDL languages. Lastly, the Vivado IP integrator enables custom system designs by instantiating and interconnecting various IPs from a catalogue of existing IPs.

3.4 Work Process

For this thesis, with the correct environment secured, the hls4ml tool served as a cornerstone on which the development of ML inference on FPGAs could be built. For obvious reasons the selected programming language for high-level implementation and further development of the Baler algorithm was Python. This language was also selected for various data processing steps required when fetching and handling the dataset later used in the implementation.

The data was provided by the MAX IV laboratory and contained results from one of their experiments, investigating the rheology of liquid foams by fast synchrotron X-ray tomographic microscopy [39]. The total size of the dataset amounted to around 290 gigabytes, resulting in the choice of using only parts of the dataset during implementation. The data was fetched from TomoBank [46], a repository for storing tomographic datasets and phantoms.

Acquiring appropriate data was the first step in the work process, followed by applying the Baler algorithm to the data to produce a model for the hls4ml conversion. The model itself was trained on parts of the dataset, consisting of entire frames of data. The data was processed in a blocking fashion, dividing each block into integer vectors of length 20. This approach aimed to reduce the model size and accommodate the limitations of the FPGA board, which could not handle overly large input sizes. The extensive dataset posed computational challenges for processing through hls4ml and simulating in Vivado due to its size. As a result, the data was divided into smaller chunks to enhance resource utilization and manageability. The compression and decompression performed by Baler in this fashion resulted in Figure 3.2, which depicts the X-ray projections of the liquid foam flowing through a constriction and being rotated around the tomographic axis. This was performed on a CPU, allowing for optimal performance in compression accuracy. However, it is important to note that the reconstructed frame is not a perfect copy of the original and has lost some of its characteristics that cannot be seen clearly in the figure.

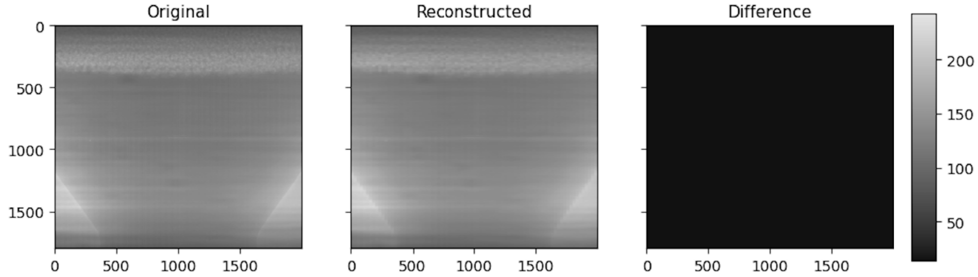

Figure 3.2: Tomography data before and after reconstruction using Baler.

After ensuring the model performed adequately, it was converted into an HLS project using hls4ml. The next step was to assess the feasibility of the HLS project generated by hls4ml. Additionally, Vivado would be utilized to properly program the FPGA and thoroughly test the board.

Chapter 4

Implementation

The implementation chapter describes the entire development chain, from start to finish, including every main part of the thesis that had a considerable impact on the project. Each step involved various subtasks that were executed to achieve the project's objectives. By meticulously detailing the process, this chapter provides a clear roadmap of the project's progression and sets the stage for the subsequent analysis and evaluation in the ensuing chapter.

4.1 Preprocessing & Environment Setup

The first step involved setting up the appropriate work environment, which included configuring Baler with hls4ml and ensuring the correct version of Vivado along with the necessary licenses. The final setup comprised Baler with hls4ml version 0.7.1, integrated with Vivado 2020.1 using licenses provided by the department. Additionally, some configurations were performed in Vivado on clusters run at MAX IV, which helped alleviate the general workload.

Secondly, the task of preprocessing the original data was essential due to its immense size and the format of the original files, which were generated in Hierarchical Data Format version 5 (HDF5) [25]. This format is specifically designed for storing and organizing large amounts of data. However, since Baler takes Numpy arrays as input, the data had to be converted into the correct format before running through training. This was done by fetching the hierarchical data tables of the HDF5 data that represented the actual images. Additionally, the original dataset was approximately 290 gigabytes in size and required pruning. Consequently, the decision was made to divide this data into evenly split fractions, retaining every Nth frame to maintain data integrity. While the optimal solution would have been to use all 290 gigabytes of data, the goal was to train the model on enough data to adequately reconstruct a few frames from the entire dataset, rather than creating the perfect model for reconstruction. Therefore, some data was discarded to make the training process more manageable.

4.2 Model Creation & HLS Conversion

4.2.1 DNN Model

The chosen model, deemed suitable in size for the available FPGA and expected to perform adequately in terms of compression, was a deep neural network (DNN) architecture. The decision was made to opt for a DNN over a convolutional neural network (CNN) architecture due to the latter's increasing demands on memory. Additionally, the CNN was not as compatible with the block training of the tomography data provided by MAX IV, producing reconstructions of poor accuracy. Since a CNN network learns and recognizes patterns, features and spatial hierarchies within images, the block training is believed to have disrupted this part of the CNNs training and made the more general DNN network more appropriate. The DNN architecture comprised an encoder and a decoder, producing an intermediate latent space in accordance with AE design principles.

The encoder, as shown in Figure 4.1, consisted of four fully connected dense layers with ReLU activation functions between them, gradually reducing the input dimensionality from the number of features to the dimension of the latent space. The first layer contained 200 units and processed an input vector of 20 values, producing 200 output values for the subsequent layer. The second layer consisted of 100 units, reducing the dimensionality from 200 to 100 output values for the third layer. The third layer further reduced the dimensionality to 50 units, preparing the data for the fourth and final layer. The fourth layer comprised only 5 units, generating the 5 outputs that constituted the latent space or compressed data.

In contrast, the decoder mirrored the structure of the encoder function in reverse order, consisting of four fully connected dense layers with ReLU activation functions between them. However, unlike the encoder, the decoder's purpose was to reconstruct the input data from the latent space representation. Therefore, instead of decreasing the input dimensionality into the latent space, the decoder increased the latent space dimensions to match the original data dimensions. This architecture aimed to learn a compact representation of the input data in the latent space and reconstruct the input data from this representation.

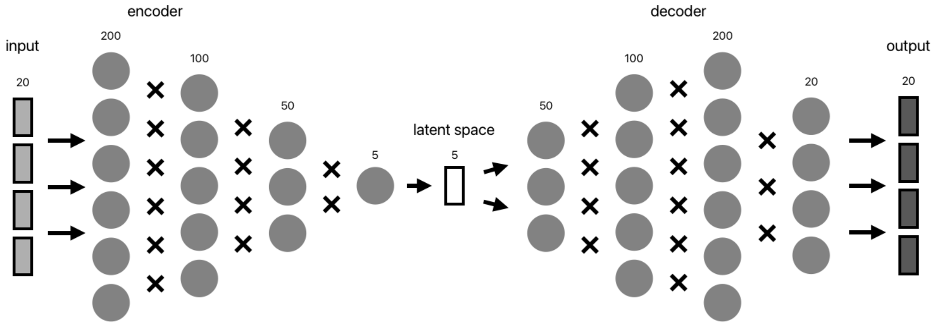

Figure 4.1: DNN Model used for FPGA implementation.

4.2.2 Model Limitations

The suitability of the model relied on both the compression results and the available resources. An issue that surfaced during the implementation phase was the necessity to align the computational complexity of the model with the limitations of the FPGA card. The model went through tens of thousands of computations, specifically multiplication operations, in order to perform the compression. Since the FPGA card has a constraint on the number of multiplications it can perform simultaneously, careful consideration was required to ensure optimal performance. This constraint depends on the number of DSP slices in the FPGA, which for the XCZU7EV-2FFVC1156 is 1728. Apart from this, other limitations on the memory, LUTs, and FFs also became apparent when the first conversion to HLS from the model was done.

The approximate number of multiplications performed by the model can be deduced by calculating the multiplications done by the units in each layer depending on the input to each layer, which can be seen in Table 4.1.

Layer	Input	Units	No. of Multiplications
Encoder1	20	200	4000
Encoder2	200	100	20000
Encoder3	100	50	5000
Encoder4	50	5	250
Decoder1	5	50	250
Decoder2	50	100	5000
Decoder3	100	200	20000
Decoder4	200	20	4000

Table 4.1: Total number of multiplications performed by the DNN model by layer.

By conducting these straightforward calculations regarding the model and combining them with the aforementioned number of DSP slices available, several key conclusions could be drawn. First and foremost, the total number of multiplications for the entire model would be close to 60000, which with 1728 DSP slices poses a complication. With the rough estimation that each DSP slice can perform one multiplication apiece, the reuse factor for every slice would be close to 34 if the utilization of them was at 100%. However, if the utilization were at a more reasonable level, such as around 80%, the reuse factor would be around 42. This means that every DSP would have to be reused 42 times during each run of the model, leading to a significant increase in latency and timing for the board.

A high reuse factor means that each DSP slice must be utilized multiple times during each run of the model. This increases latency, as each DSP slice has to wait for its turn, potentially creating bottlenecks in the pipelining process. Moreover, the frequent reuse of DSP slices can lead to timing constraints being violated, as the FPGA may struggle to meet the required clock frequencies due to the increased workload.

This leads us to the second conclusion: in order to alleviate the strain on the FPGA's resources, either the model would have to be made smaller or the model would have to be split up. The choice was made to divide the encoder part of the model and the decoder part of the model into two separate model parts that could each run independently, effectively halving the FPGA resource requirements without affecting the performance of the model. Additionally, in a real-world application, there would typically be an encoder transmitting the data and a decoder receiving it, making this approach even more relevant.

Having done this, the total number of multiplication operations that need to be computed was only 29250, meaning that each DSP would have to be reused approximately 21 times or 22 times for margin. This approach was more reasonable to meet timing and latency requirements. Additionally, by dividing the model into two parts, testing became easier and could be conducted more efficiently as well.

4.2.3 Hls4ml Conversion

The next step of the implementation was the conversion into an HLS project, which was accomplished using the Python complementary tool hls4ml. The model was split into two separate projects and examined individually, although, for performance purposes, this separation did not significantly impact the process. During the conversion process, however, an interesting discovery was made.

Fixed Point Numbers

The hls4ml tool only supports fixed-point numbers [43] when performing the conversion. Fixed point numbers are a suitable representation of fractional numbers for computers or hardware when floating point numbers are too computationally intensive. They are therefore used when performance takes precedence over precision, which is often the case for FPGAs. Fixed point numbers are typically written as a bit width of the total number within angle brackets, along with the bits for the fractional part of the entire number. For example, <10,4> represents a fixed point number with 10 bits, with 6 bits representing the integer part and 4 bits representing the fractional part. However, the hls4ml tool uses a reversed representation, including the bit width of the full number with the integer part of the number instead of the fractional. This means that the fixed point number <10,4> in hls4ml represents a number with a 4-bit integer part and a 6-bit fractional part instead.

Choice of Precision

The initial precision choice was <16,8> to minimize resource allocation while maintaining decent model accuracy, given the computational demands of the model. The DSP48E slices on the board used in this project can compute 18-bit by 25-bit numbers. The goal was to keep the precision below this 18-bit threshold to conserve computational power, avoiding the need to use two DSP slices for one number. However, experiments with different precision configurations revealed that performance improvements were possible without significantly increasing resource utilization beyond that of <16,8>. This suggests that fine-tuning precision could yield better results while maintaining efficient use of the FPGA's resources.

The initial precision choice was <16,8> to minimize resource allocation while maintaining decent model accuracy, given the computational load of the model. The DSP48E slices that are part of the board used in this project have the capability to compute 18-bit by 25-bit numbers. The intention was to keep the precision below this 18-bit threshold to conserve computational power, avoiding the need to use two DSP slices for one number. However, experiments with different precision configurations revealed that performance improvements were possible without significantly increasing resource utilization beyond that of <16,8>.

The first valuable discovery that was made can be depicted in Figure 4.2. Namely, when reaching certain levels of bit width for precision, the DSP usage doubles. This graph is the result of experimenting with different levels of precision during conversion for otherwise identical configurations and models. It is important to note that this is the result of Vivado HLS estimation and not actual implementation on any board, as some utilization values exceed 100%.

The exploration aimed to understand how different precision settings affect the FPGA's resources and how the hls4ml precision configuration impacts the overall outcome. The graph consistently illustrates that DSP usage doubles at a specific precision level, as anticipated. Surprisingly, DSP usage did not increase at a bit width of 19 as initially expected but instead rose at a bit width of 20. This discrepancy is unexpected considering that one DSP slice should only handle a maximum of an 18-bit number at once. The cause for this might be the overall route optimization provided by the Vivado tool, but not something that will be further discussed in this thesis.

Another notable discrepancy apparent in Figure 4.2 was the DSP usage for 26 and 27 bits precision, which should logically be similar or equal to that of 28 bits and above. Nonetheless, the observed DSP usage deviates from this expectation as well.

The advantage that this brought to the project can be understood by observing Figure 4.3. The heatmap illustrates the mean difference (MD) of the model's results when executed on a CPU using the original dataset, compared to the results obtained from simulation in Vivado with the same dataset. Similar to Figure 4.2, the MD is projected by different precision configurations and varies significantly depending on the integer and fractional parts, as well as the total bit width. The heatmap can be interpreted by noting that the integer part is represented on the y-axis, while the fractional part is depicted on the x-axis. Thus, the total bit width can be calculated by summing the integer and fractional parts, as follows: $bitwidth = x + y$.

The MD illustrated in Figure 4.3 displays a discernible pattern, highlighting the significance of the total number of bits used to represent a number in minimizing the MD. This outcome was anticipated, as a number represented by a greater number of bits can inherently facilitate more accurate computations. However, there are still a few aspects to consider when examining these results.

Firstly, despite the precision of <16,8> and <19,10> requiring the same amount of DSP resources according to Figure 4.2, they yield significantly different MDs. This disparity suggests that factors other than DSP usage, such as the distribution of bits between the integer and fractional parts, also influence the model's performance. If the integer part was too small, each number would be

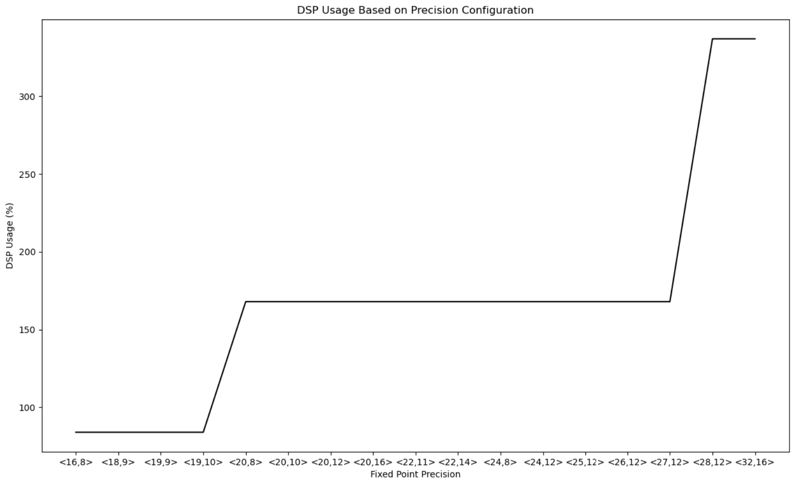

Figure 4.2: DSP usage based on precision configuration in hls4ml conversion.

calculated with less precision than necessary to achieve a fair representation. On the other hand, if the fractional part was neglected the computational detail would be lost.

Secondly, it's noteworthy that the MD of the model output becomes larger when either the integer part or the fractional part is too small, irrespective of the total bit width used for precision. This indicates that finding the right balance between the integer and fractional parts is crucial for minimizing MD.

Lastly, all of the lower values in the heatmap, including the MD at and after a total bit width of 24, are consistently similar in size. This suggests that once a certain level of precision is reached, further increasing precision may not lead to significant improvements in MD.

The most promising precision levels: <19,10>, <20,10>, <22,11>, <24-28,12>, <26,13>, <28,14>, <30,15>, and <32,16>, all yielded a MD lower than 15. This was a significantly lower MD than other lower bit widths and required some careful consideration and evaluation. While the higher precision levels produced better results, they came at a cost. Depending on the precision level, DSP usage varied from 84% for <19,10> up to 337% at most for <28,12> and <32,16>. It is important to note that the acceptable DSP usage percentage was logically capped at 84% to avoid affecting latency and timing constraints. Considering this, along with the fact that the MD differs with a ratio of 8:1 between the originally intended precision of <16,8> and <19,10>, these results decisively informed the choice of precision for the model in the FPGA implementation.

The acceptable precision level for this specific card and this specific model was <19,10>, as it allowed a mean loss of 14.5 to be achieved for the model without breaking any previous constraints. This precision level strikes a balance between

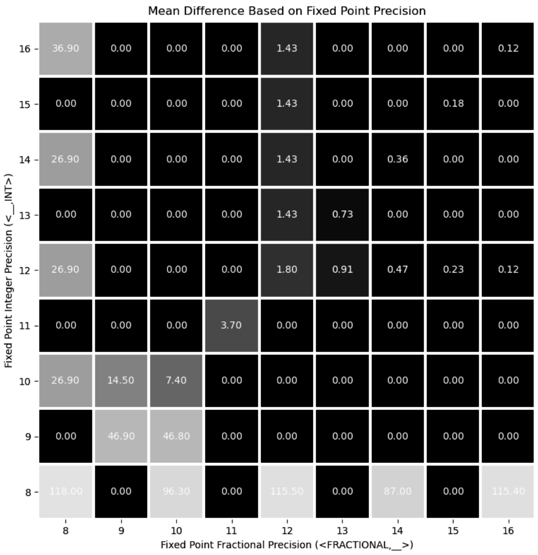

Figure 4.3: Mean difference between model output based on precision configuration in hls4ml conversion.

computational efficiency and accuracy, ensuring optimal performance while adhering to the resource limitations of the available FPGA card.

4.3 FPGA Implementation

The next stage of the implementation phase focuses on finalizing the Vivado configurations, preparing the FPGA board for testing, and conducting the tests. At this point, the model, now an HLS project, must go through the essential Vivado stages: synthesis, implementation, and bit file generation for the FPGA deployment.

To understand the project workflow and the role of hls4ml's VivadoAccelerator backend and the PYNQ [4] open-source software, the following sections provide further explanations of these components.

4.3.1 VivadoAccelerator

The hls4ml suite provides several backend options for conversion, including Quartus for Intel Quartus, Vivado for Vivado, and VivadoAccelerator, designed to accelerate operations using FPGAs. Initially, knowledge of the hls4ml backend options was limited, with the assumption that only Quartus and Vivado were available, representing different manufacturers. The plan was to use the Vivado backend due to the availability of a Xilinx board at the institution. However, further investigation revealed the VivadoAccelerator backend, which allows leveraging

PYNQ open-source software for easy model deployment on supported devices. The ZCU102 Evaluation Board, similar to the available ZCU104, was supported. With a minor tweak after conversion, the ZCU102 settings could be adjusted in Vivado to fit the ZCU104 board. Additionally, the ZCU104 board's support by PYNQ facilitated a smooth transition between the boards.

In conclusion, the choice to use the VivadoAccelerator backend aimed to streamline the deployment process. By using hls4ml's accelerator option, tested with PYNQ for deployment, the project could be efficiently prepared without unnecessary time expenditure. Given the primary objective of accelerating the ML model for data compression, VivadoAccelerator was the most suitable option for this project.

4.3.2 PYNQ

PYNQ, developed by AMD, is an open-source project that provides a Python API-based framework within the Jupyter environment. It is designed specifically for Xilinx Adaptive Computing platforms, supporting models like Zynq, Zynq Ultrascale+, and others. PYNQ offers architects, engineers, and programmers a convenient means to leverage Adaptive Computing platforms. With its Python APIs and integration with Jupyter, PYNQ simplifies the utilization of programmable logic circuits, removing the necessity for specialized ASIC-style design tools.

The main idea for PYNQ is that programmable logic circuits can be described as hardware libraries called overlays, which resemble software libraries in functionality. Users can select the most suitable overlay for their application, accessing it through a Python API. While developing a new overlay necessitates expertise in designing programmable logic circuits, the primary advantage lies in the ability to create once and reuse multiple times. Additionally, overlays, like software libraries, are designed to be configurable and reusable across a variety of applications.

The workflow of PYNQ becomes clearer when examining Figure 4.4, which illustrates the architecture of PYNQ. The figure provides a comprehensive view of the software and hardware components involved in the PYNQ framework and their interactions, highlighting the ease of development and productivity enabled by using Python and Jupyter for FPGA-based applications.

Hlsm4l with PYNQ

The hls4ml VivadoAccelerator backend enables the user to create an HLS project and generate the model as an IP in Vivado. The IP generated for this thesis closely resembled the DNN model described in previous sections. The model was represented as far as the precision allowed, as discussed in Section 4.2.3. Subsequently, the VivadoAccelerator backend facilitated the creation of a Block Design in Vivado IP Integrator containing the model IP, along with other necessary IPs to form a complete system. To enable data transfer between the programmable logic containing the model and the processing system, an AXI Direct Memory Access IP was added.

With the complete system finalized, it underwent synthesis and implementation in Vivado to evaluate the feasibility of the structure and generate a bit

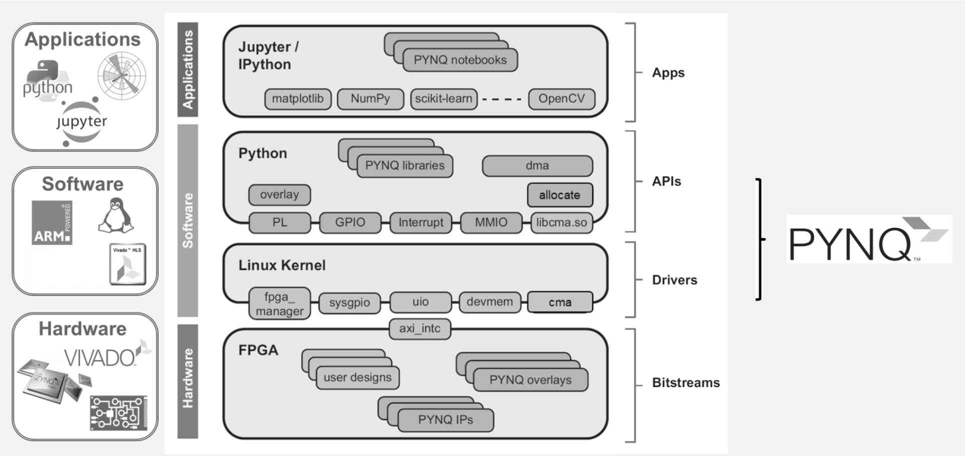

Figure 4.4: Overview of the PYNQ framework, picture from [28].

file. Following this, PYNQ was leveraged with a custom overlay for the model IP, following the workflow outlined in Figure 4.4.

4.3.3 Development Chain

Despite the fact that the implementation was a major part of the overall workflow, the development chain encompassed everything from coding in high-level Python to generating and running the bit file on the FPGA. Figure 4.5 provides an overview of the development chain, illustrating the entire work process and highlighting the critical elements necessary to conduct the actual tests.

This workflow can be compared to the hls4ml workflow, illustrated in Figure 3.1, where the model and HLS conversion were done in a high-level environment, and the synthesis of the model and implementation on the FPGA were done with Vivado. The utilization of the PYNQ overlay, with the help of hls4ml, saved a lot of time during implementation, allowing more time for conducting the actual tests.

The different steps in Figure 4.5 are, of course, simplified, but they describe the general approach taken during the work. Some steps were repeated multiple times during the implementation phase to achieve the desired results. However, this figure provides a clear representation of the final development chain of the thesis work and implementation.

4.3.4 FPGA Test Runs

Once the entire finalized system was in place, testing on the FPGA card commenced. Throughout the implementation phase, various theories and considerations arose, prompting additional tests to be conducted to address any questions that arose. Consequently, a few new models were trained and tested, along with other configurations. For clarity, the original DNN model discussed thus far will be

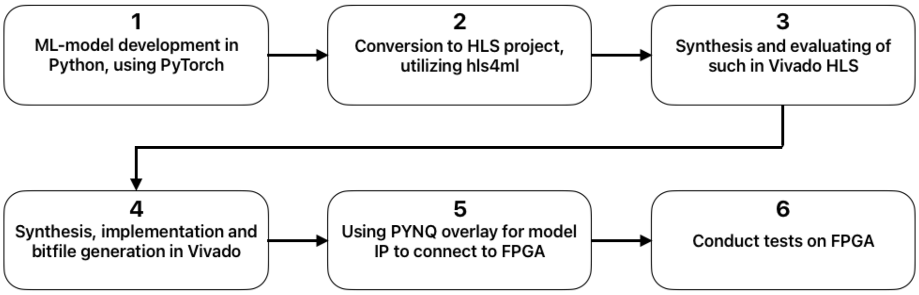
1
ML-model development in Python, using PyTorch
2
Conversion to HLS project, utilizing hls4ml
3
Synthesis and evaluating of such in Vivado HLS
4
Synthesis, implementation and bitfile generation in Vivado
5
Using PYNQ overlay for model IP to connect to FPGA
6
Conduct tests on FPGA

Figure 4.5: Overview of the workflow during the thesis work.

referred to as *DNN Large* in subsequent sections, while any new models introduced will be given appropriate names.

Data Input Size

The testing on the actual FPGA card was conducted once the system was in place, along with an adequate amount of data to fit into the AXI Direct Memory Access buffer. This approach aimed to assess the speed and throughput of the model programmed on the board. The amount of data used for testing was deemed irrelevant, as the card's performance was expected to scale linearly with the amount of data processed. This theory was also put to the test by running the same model, both encoder and decoder, with varying amounts of data, as shown in Table 4.2. Each frame of data was represented by 180000 arrays of size 20, with the second and third rows representing the compression of entire frames of data, while the first row represented an attempt to test an edge case in input size. The smaller input size exhibits a lower throughput due to longer processing time per input. This phenomenon was believed to be primarily caused by the time it takes to send the data to and from the AXI Direct Memory Access. To accurately capture the compression and decompression timing, measurements were taken before sending the data to the model and after receiving the data from the model. This approach better reflected the actual compression process, including any relevant overhead. Given the reduced amount of data to compress, the measured time for compressing one array of size 20 was influenced by the time taken to send and receive data, making it an unfair representation of the actual throughput.

Another noteworthy observation while examining Table 4.2 is the significant difference in throughput between the encoder and decoder for the same amount of input. The primary factor that contributed to this discrepancy was the difference in input size between 20 and 5 for the encoder function and the decoder function, respectively.

DNN Large	Input Size (E:20, D:5)	Time (s)	Throughput (inferences/s)
Encoder	1	0.0028	719
Encoder	180000	1.26	142377
Encoder	720000	5.05	142655
Decoder	1	0.0017	1186
Decoder	180000	0.32	568386
Decoder	720000	1.27	567929

Table 4.2: The results of running different input sizes through the FPGA card for both the encoder and decoder, using DNN Large.

Precision

After testing different input sizes for the same model, another test was conducted: changing the precision for the same model while keeping all other settings consistent. The purpose was to observe if the FPGA would respond differently to computations with either a very high or very low integer bit rate while maintaining the same total bit width of precision.

The precision levels reviewed were <19,1>, <19,5>, <19,10>, <19,14>, and <19,19>, chosen to evenly distribute the results. However, no change in throughput was observed for any of the precision settings, most likely due to the total bit width remaining the same from <19,1> to <19,19>. Therefore, the original precision of <19,10> was retained for further tests, as it provided the most precise representation of the model for the FPGA.

Model Size

The subsequent step in the implementation phase involved altering the model size to assess how resource utilization on the card affected data throughput. Adjusting the model size, particularly reducing it, would inevitably affect the model's performance. However, given that the original model was already constrained by its size and required reduced precision to fit on the FPGA card, the primary objective of this test was not to generate an accurate model but rather to evaluate the FPGA's capabilities and explore Baler's bandwidth compression possibilities.

The initial step involved removing the largest layer from the DNN Large model to substantially reduce its size and computational load. This revised model, referred to as *DNN Reduced* henceforth, is depicted in Figure 4.6 and comprises only 7250 multiplications, in accordance with previous calculations. This represented a significant reduction of 75%, compared to the 29250 multiplications performed by the DNN Large model. Consequently, the amount of resources and DSP slices required on the board was drastically reduced.

According to previous research, the size of the model, or any solution applied to an FPGA, greatly affects the performance of the FPGA. By reducing the model size, theoretically, the FPGA should work more efficiently, as long as the original model does not exceed the card's capacity. In this case, the DNN Large uses a reuse

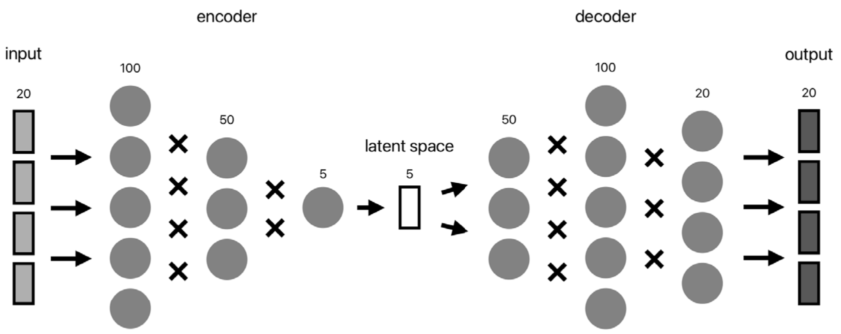

Figure 4.6: Reduced DNN Model used for FPGA implementation.

factor of about 22 for each DSP slice, indicating that the model is theoretically 22 times too large for the FPGA card, and would ideally need to be 22 times smaller to fit onto the card without any compromises.

To maintain the utilization of DSP slices at approximately 80%, as discussed in Section 4.2.2, the reuse factor for the DNN Reduced model only needed to be around 7, compared to 22 for the DNN Large. This reduction significantly eased the strain on the FPGA, potentially increasing the throughput, as the number of computations to perform for the DSP slices was fewer for each run of the model.

The tests were performed for both the encoder and decoder and the results are presented in Table 4.3 and include the same models run on a CPU of type Intel Core i7-2600K 3.40 GHz x 8 [30]. The final comparison boils down to CPU vs FPGA. Without reducing the model size, the CPU and FPGA performed similarly for the encoder, while the decoder on the FPGA exhibited faster performance even for the DNN Large model. However, these results will be discussed further in the next chapter.

Model	Processing Unit	Time (s)	Throughput (inferences/s)
DNN Large (E)	CPU	0.95	189473
DNN Large (E)	FPGA	1.26	142377
DNN Large (D)	CPU	1.21	148760
DNN Large (D)	FPGA	0.32	568386
DNN Reduced (E)	CPU	0.94	191489
DNN Reduced (E)	FPGA	0.23	768481
DNN Reduced (D)	CPU	1.24	145165
DNN Reduced (D)	FPGA	0.059	3039308

Table 4.3: The results of comparing the implementation of different model sizes for both the encoder and decoder.

To further assess the capabilities of the FPGA card, one more model was developed. Similar to DNN Reduced, the primary goal of this model, referred to as DNN Tiny, was not to achieve significant compression but rather to explore the potential of custom-fitting an ML model to the FPGA card. DNN Tiny consisted of four fully connected layers for both the encoder and decoder, mirroring the structure of the original DNN Large model. However, the layer sizes were drastically scaled down, with each layer containing 32, 16, 8, and 5 nodes for the encoder, and 5, 8, 16, and 32 nodes for the decoder. The complete model architecture is depicted in Figure 4.7.

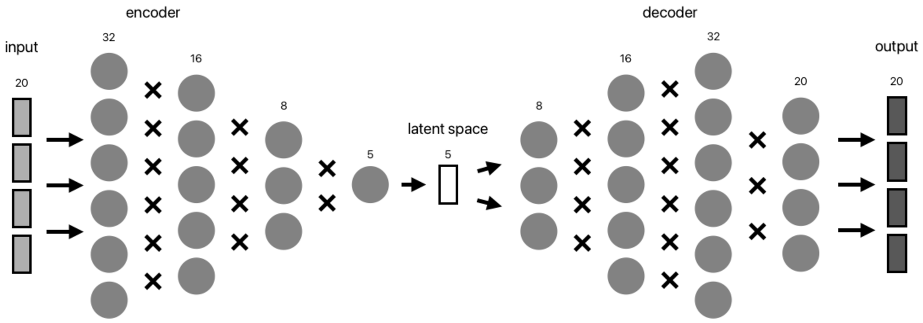

Figure 4.7: Tiny DNN Model used for FPGA implementation.

DNN Tiny comprised a total of 1320 multiplications, which allowed for a reuse factor of 1 without fully maximizing the utilization of resources on the FPGA. Consequently, this model had the potential to produce even better results than the previous two that had been tested. However, as shown in Table 4.4, the speed and throughput did not change considerably between DNN Reduced and DNN Tiny. This was somewhat perplexing, suggesting that the FPGA might have reached a limit of its capabilities. Despite both models fitting onto the FPGA with different reuse factors, it appeared that there were additional factors influencing the throughput of the model when implemented on the FPGA.

DNN Tiny	Processing Unit	Time (s)	Throughput (inferences/s)
Encoder	CPU	0.86	209302
Encoder	FPGA	0.23	763297
Decoder	CPU	1.20	149340
Decoder	FPGA	0.059	3040027

Table 4.4: The results of comparing the implementation of different model sizes for both the encoder and decoder.

Clock Period & Frequency

The last factor considered to influence the model's throughput was the clock period and frequency. The clock period, denoted as T, in a digital system, represents the interval between two consecutive rising or falling edges of the clock signal. It determines the maximum speed at which the system can reliably operate. Conversely, clock frequency, measured in Hertz (Hz), represents the number of clock cycles per unit of time, expressed as $f = \frac{1}{T}$. In essence, clock period and frequency are inversely related, with a shorter clock period allowing for higher clock frequencies.

Vivado and Vivado HLS employ the clock period as the standard format rather than the clock frequency. Consequently, throughout the remainder of this thesis, the primary measurement will be discussed in terms of clock period rather than frequency. For the DNN Large hls4ml conversions to the HLS project, the target clock period was set to 3 nanoseconds to ensure the model could fit onto the FPGA without timing and latency problems. Initially, with the smaller models, it was decided not to alter the clock period to maintain a fair comparison to the original model. However, as the focus turned to assessing the capabilities of the FPGA card for Baler's compression algorithms, the final test runs involved comparing different clock periods for the DNN Tiny model IP.

To achieve the optimal performance configuration on the FPGA card, adjustments were made to the clock period, and the IP was resynthesized in Vivado HLS. This process was repeated multiple times to compare the utilization summary for each configuration. The results of this experiment are shown in Table 4.5 and Table 4.6, where both the target and estimated clock periods are displayed, along with the latency. The latency is presented in cycles, indicating the estimated number of clock cycles it takes for the IP to produce an output. It is important to note that these results were obtained through a tool and do not represent the outcomes of an actual implementation of the IPs. Therefore, there could be variations between these results and those observed in an actual implementation.

Theoretically, the execution time could be calculated from these values using the formula $E_t = p * l$, where E_t represents the execution time, p denotes the clock period and l indicates the latency. However, if the values in Table 4.5 and Table 4.6 were a 100% accurate representation of reality, the implementation of the model IPs would have been unnecessary. Since this is not the case, tests were conducted with different clock periods for DNN Tiny to gain clarity on how the period affects the actual throughput.

The clock periods of 3 nanoseconds, 10 nanoseconds, 18 nanoseconds, 19 nanoseconds, and 50 nanoseconds were particularly intriguing for investigation due to their significance in comparison to the larger models. In the preceding tables, clock periods of 19 nanoseconds and above yielded an estimated timing of 16.052 for both the encoder and decoder. Therefore, it was compelling to measure the actual throughput obtained for both 19 nanoseconds and 50 nanoseconds to assess any differences. The choice to test 10 nanoseconds and 18 nanoseconds was informed by the latency data shown in Table 4.5 and Table 4.6. Notably, for the encoder, the drop from 19 to 18 nanoseconds resulted in a latency almost double in size, theoretically, it should result in halving the throughput when

DNN Tiny	Target Timing (ns)	Estimated Timing (ns)	Latency (cycles)
Encoder	1	1.316	341
Encoder	3	2.605	121
Encoder	5	4.297	81
Encoder	10	8.748	41
Encoder	15	10.741	41
Encoder	16	10.741	41
Encoder	17	14.836	41
Encoder	18	14.836	41
Encoder	19	16.052	21
Encoder	20	16.052	21
Encoder	35	16.052	21
Encoder	50	16.052	21
Encoder	100	16.052	21

Table 4.5: The results of comparing the implementation of different clock periods for DNN Tiny encoder.

DNN Tiny	Target Timing (ns)	Estimated Timing (ns)	Latency (cycles)
Decoder	1	1.316	460
Decoder	3	2.605	137
Decoder	5	4.297	97
Decoder	10	8.748	70
Decoder	15	10.741	49
Decoder	16	10.741	49
Decoder	17	14.836	49
Decoder	18	14.836	49
Decoder	19	16.052	49
Decoder	20	16.052	49
Decoder	35	16.052	49
Decoder	50	16.052	49
Decoder	100	16.052	49

Table 4.6: The results of comparing the implementation of different clock periods for DNN Tiny decoder.

implemented on the FPGA. Following the earlier mentioned formula for E_t, the execution time would be doubled along with the latency. Since the throughput is strictly connected to the execution time, an increase in execution time would result in a decrease in throughput. Similarly, for the decoder, although the latency remained consistent between 18 and 19 nanoseconds, at 10 nanoseconds, the latency increased to 70 cycles from 49, which should theoretically impact the throughput. Each new clock period was tested for both the encoder and decoder to maintain a comprehensive test bench.

The conclusive tests involving varied clock periods provided significant insights into important factors regarding timing and latency when implementing solutions onto the FPGA. These findings are encapsulated in Table 4.7, revealing the correlation between time/throughput and latency, as compared to the data presented in Table 4.5 and Table 4.6. As predicted, the throughput was affected greatly by the latency even though the clock period, or timing, was changed.

The cause for this will be further discussed in the upcoming chapter. However, it is worth highlighting that these concluding experiments have yielded valuable data, potentially serving as a building block for any future work related to ML-applications applied to FPGAs. The information gathered from these tests could inform subsequent research directions, guiding the refinement and optimization of FPGA implementations.

DNN Tiny	Target Timing (ns)	Time (s)	Throughput (inferences/s)
Encoder	3	0.23	763297
Encoder	10	0.085	2121365
Encoder	18	0.085	2122641
Encoder	19	0.051	3472422
Encoder	50	0.054	3319379
Decoder	3	0.059	3040027
Decoder	10	0.031	5804204
Decoder	18	0.023	7655339
Decoder	19	0.023	7744267
Decoder	50	0.024	7413874

Table 4.7: The results of using different clock periods for DNN Tiny and running it on the FPGA card.

Chapter 5
Discussion

The objective of this chapter is to examine and interpret the outcomes of the implementation, addressing the research questions posed. Moreover, any potential avenues for future exploration, along with conclusions of the thesis work.

5.1 Evaluation of Results

In the preceding chapter, the task of implementing a few of Baler's ML-based compression models on the available FPGA card was successful. This required extensive preparation and necessitated adaptations as the work progressed. The final outcome of these efforts proved to be interesting in several ways, key notes and takeaways from this are described below.

1. The average FPGA card does not have the capacity to support larger ML-models, posing challenges for practical implementation.
2. The simplicity of implementing anything on the FPGA should not be taken for granted.
3. Under the right conditions, FPGAs do have the potential to enhance throughput and, consequently, the bandwidth compression of Baler's algorithms.
4. Under unfavourable conditions, pursuing FPGA implementation might result in an expensive and time-consuming venture with no tangible improvement in compression throughput.

The most evident challenge encountered during the implementation phase was the disparity in size between the original DNN Large model and the FPGA's capacity. The DNN Large model consumed more resources than the FPGA could accommodate, necessitating the primary objective of fitting any model onto the FPGA for testing purposes. Through adjustments and appropriate configurations, this goal was achieved, largely owing to the flexibility of hls4ml and the FPGA card. By utilizing a reuse factor of 22 and selecting a suitable precision level, the DNN Large model was successfully implemented on the FPGA. While this solution was not ideal, the constraints imposed by the computational demands and hardware limitations left little room for alternative approaches.

Model Constraints

The decision was made to pursue a smaller model to explore the extent of the FPGA's capabilities without being restricted by the FPGA's resources. To provide a more balanced assessment of the FPGA's potential without the size constraints posed by DNN Large, the development of DNN Reduced and DNN Tiny ensued. The DNN Reduced managed to go through the same amount of data as DNN Large in less than 1/5th of the time for both the encoder and decoder. Interestingly, the model size did not seem to be the predominant factor impacting throughput beyond a certain point. Both DNN Reduced and DNN Tiny achieved equivalent throughput when utilizing a clock period of 3 nanoseconds, despite DNN Reduced having a reuse factor of 7 and DNN Tiny having a reuse factor of 1.

Increase in Throughput

The primary factor contributing to the notable speedup and higher throughput achieved with the DNN Tiny model appears to be the adjustment in the clock period. Intuitively, one might expect that a shorter clock period or a higher clock frequency would result in a greater throughput, as the card can operate at a faster pace, thereby producing results more quickly. However, optimizing for performance involves more nuanced considerations than simply accelerating compilation speed and hoping for optimal results.

The top-performing implementation on the FPGA was DNN Tiny with a clock period of 19 nanoseconds, as illustrated in Table 4.7. This configuration, referred to as DNN Tiny19, demonstrated only a marginal improvement in speed compared to the same model using a clock period of 50 nanoseconds, designated as DNN Tiny50. This disparity can be explained by examining the latency tables in Table 4.5 and Table 4.6. It is evident that shorter clock periods correspond to higher latencies. This correlation is logical, as reducing the clock period results in increased latency. The faster one clocks a circuit, the less time the circuit has to produce a result before the next clocking occurs.

Therefore, a longer clock period for DNN Tiny resulted in a higher throughput because the card had more time to finish each cycle without increasing the latency. Of course, as can be seen in the tables, there is a limit to this correlation as well. At a certain point, the higher clock period results in a longer time, and this is why it is important to conduct experiments for any implementation on the FPGA.

However, with the optimal configurations, a throughput of 16.9 times the throughput of the available CPU was acquired, which is a significant speedup indeed. This highlights the potential of FPGA-based implementations for accelerating ML-based compression algorithms, provided careful optimization and configuration.

5.2 Regarding the Problem Statement

The primary goal of this thesis was to implement Baler's compression algorithms on an FPGA to assess the possibilities and challenges associated with ML-based compression on dedicated hardware.

5.2.1 To what extent can the deployment of Baler's compression algorithms on FPGAs improve its bandwidth compression?

Research question one explores the possibilities associated with ML-based compression, particularly Baler's compression, on an FPGA. The deployment of Baler's compression algorithms onto an FPGA was successful, achieving a compression speed increase of up to 16.9 times compared to a CPU. The time and throughput results, compared to a CPU, for the three different models used are depicted in Table 5.1. This was accomplished by using a model that leveraged the FPGA's adaptability and parallelism, fitting onto the card without exhausting all its resources.

Model (Encoder)	Processing Unit	Time (s)	Throughput (inferences/s)
DNN Large	CPU	0.95	189473
DNN Large	FPGA	1.26	142377
DNN Reduced	CPU	0.94	191489
DNN Reduced	FPGA	0.23	768481
DNN Tiny	CPU	0.86	209302
DNN Tiny	FPGA	0.05	3472422

Table 5.1: The results of comparing the implementation of different model sizes for the encoder.

The extent to which Baler's bandwidth compression can be improved is significant, though limited to some degree. These tests focused on the actual compression speed and did not account for the overhead that might occur due to other components in a full-scale FPGA implementation for compression. The tests were conducted to ensure equal conditions for both the CPU and the FPGA regarding data compression. In a real-world scenario, data would need to go through some intermediate stages before compression, applicable to both CPUs and FPGAs. Therefore, pure compression throughput was considered a more appropriate metric for testing. Nonetheless, the highest-performing model on the FPGA was 16.9 times faster than the CPU, representing a substantial speedup even when accounting for potential overhead.

5.2.2 What are, if any, the limitations of deploying Baler's compression algorithms on FPGAs?

Research question two addresses the challenges associated with performing Baler's ML-based compression models on an FPGA. During the implementation phase, several limitations emerged related to the model's size, the FPGA's capacity, the precision level, and the card's clock frequency. These factors significantly influenced the final solution and were challenging to anticipate in advance.

The initial plan was to use the DNN Large model on the FPGA, as it demonstrated decent compression and decompression performance on the CPU. However, this plan had to be revised when it became evident that the model was too large to fairly represent FPGA's capabilities. The size of the model and the limitations of the FPGA's resources posed a significant challenge that needed to be addressed.

The next stage of the implementation used a smaller reduced model and a tiny model to assess the FPGA's capabilities when utilizing fewer resources. During the preparation phase, it became apparent that the precision level influenced the strain on the FPGA's resources, thereby limiting the accuracy of the compression. However, once the resource allocation was determined during synthesis and implementation in Vivado, the throughput was not significantly impacted.

5.3 Future Work

5.3.1 Larger FPGAs

One of the most evident areas for future research is the exploration of larger FPGAs, which could potentially handle larger models, thereby enabling more accurate compression and decompression. Currently, the largest FPGA in production by AMD supports 3840 DSP slices [5], which is roughly double the 1728 DSP slices available on the FPGA used in this research. While this is not an enormous difference, it highlights the challenges inherent in developing computationally demanding projects, such as an ML-accelerator on an FPGA.

In the future, access to even larger FPGAs will likely improve, but current hardware limitations present significant challenges for implementing ML-based data compression solutions on FPGAs. With double the capacity, a wider range of projects and models could fit onto the FPGA card, potentially yielding even more promising results. This expanded capacity could enable the handling of more complex models and datasets, providing deeper insights into the capabilities and performance enhancements possible with FPGA-based ML accelerators.

5.3.2 Optimized Model

Another angle for future development could be the optimization and compilation of the model to run faster, thereby potentially accelerating the compression process. PyTorch has recently introduced a new method to compile models trained using their libraries, allowing them to run faster by compiling PyTorch code with optimized kernels [47]. This approach could be explored to enhance the performance of Baler's compression algorithms on both CPUs and FPGAs, potentially providing a speedup and making the solution more efficient in real-time applications.

5.3.3 Quantization Aware Training

One promising path for future research could be the integration of Quantization Aware Training (QAT) into Baler's compression models. QAT involves simulating the effects of quantization during the training process, allowing the models to learn and adapt to the lower precision that will be used during inference on hardware such as FPGAs [1]. This approach could potentially enhance the efficiency of the model without substantial loss of accuracy, making it highly suitable for deployment in resource-constrained environments.

5.3.4 Vivado Optimization

There are potential optimizations in Vivado that could be explored for future development of ML models. When using hls4ml, the detailed structure of the resulting project has not been as thoroughly inspected as it possibly could be. As mentioned earlier in the thesis, the hls4ml tool is useful for accelerating the development curve and integrating ML-based models on FPGAs. However, this comes at the cost of a decrease in detail since the project is generated, with custom configurations, but not built from the ground up. Realizing an ML-model in Vivado and Vivado HLS from scratch would be very time-consuming, which is why hls4ml is so beneficial. Nonetheless, undertaking this process could uncover optimizations and potentially increase throughput even further.

5.4 Conclusion

Throughout this thesis, several of Baler's ML-based compression models were successfully implemented on an FPGA, leveraging intermediate elements such as hls4ml and PYNQ. The resulting increase in throughput, up to 16.9 times compared to available CPU resources, underscores the viability of FPGA-accelerated ML solutions. Analysis reveals that key factors influencing optimal FPGA performance include model size, precision levels, and clock period.

While the findings showcase promising advancements, the immediate real-world applicability of these discoveries may be somewhat constrained. The challenge lies in accurately assessing the full extent of Baler's bandwidth compression improvements amidst the complexities of varied real-world scenarios, potentially introducing overhead. Thus, simpler solutions, such as GPUs, may currently hold greater practical appeal within the technical landscape.

Nevertheless, the significant enhancement in throughput achieved through FPGA implementation signals a promising trajectory for future research. If the technology of FPGAs continues to evolve, further exploration and refinement of hardware-accelerated ML solutions hold substantial potential. Future developments in FPGA architecture and optimization techniques may catalyze more seamless integration of ML algorithms with hardware solutions, presenting compelling paths for innovation in this dynamic field.

In conclusion, while the immediate practical applications may be limited, the progress made in improving throughput through FPGA-based compression models provides a valuable foundation for continued research and development. Looking forward, the convergence of ML and FPGA technology holds potential for incremental advancements, contributing to the evolution of computational methods and offering opportunities for more efficient and adaptable hardware-accelerated ML solutions.

References

[1] Thea Aarrestad, Vladimir Loncar, Nicolò Ghielmetti, Maurizio Pierini, Sioni Summers, Jennifer Ngadiuba, Christoffer Petersson, Hampus Linander, Yutaro Iiyama, Giuseppe Di Guglielmo, et al. Fast convolutional neural networks on fpgas with hls4ml. *Machine Learning: Science and Technology*, 2(4):045015, 2021.

[2] Adobe. Lossy vs lossless compression differences and when to use. `https://www.adobe.com/uk/creativecloud/photography/discover/lossy-vs-lossless.html`. Accessed: 2024-02-10.

[3] Muzhir Shaban Al-Ani and Fouad Hammadi Awad. The jpeg image compression algorithm. *International Journal of Advances in Engineering & Technology*, 6(3):1055–1062, 2013.

[4] AMD. Pynq. `http://www.pynq.io`.

[5] AMD. Virtex ultrascale+ vu19p. `https://www.xilinx.com/products/silicon-devices/fpga/virtex-ultrascale-plus-vu19p.html#advantage`.

[6] AMD. Vivado overview. `https://www.xilinx.com/products/design-tools/vivado.html#overview`.

[7] AMD. Zynq ultrascale+ mpsoc zcu104 evaluation kit. `https://www.xilinx.com/products/boards-and-kits/zcu104.html`.

[8] Baler-compressor. Baler. `https://github.com/baler-compressor/baler`, 2024. Accessed: 2024-01-15.

[9] Dor Bank, Noam Koenigstein, and Raja Giryes. Autoencoders. *Machine learning for data science handbook: data mining and knowledge discovery handbook*, pages 353–374, 2023.

[10] Fritjof Bengtsson, Caterina Doglioni, Per Alexander Ekman, Axel Gallén, Pratik Jawahar, Alma Orucevic-Alagic, Marta Camps Santasmasas, Nicola Skidmore, and Oliver Woolland. Baler – machine learning based compression of scientific data, 2023.

[11] Aleksandr Berezkin, Alexey Slepnev, Ruslan Kirichek, Dmitry Kukunin, and Daniil Matveev. Data compression methods based on neural networks. In *The 5th International Conference on Future Networks & Distributed Systems*, pages 511–515, 2021.

[12] Stephen D Brown, Robert J Francis, Jonathan Rose, and Zvonko G Vranesic. *Field-programmable gate arrays*, volume 180. Springer Science & Business Media, 2012.

[13] Srihari Cadambi, Abhinandan Majumdar, Michela Becchi, Srimat Chakradhar, and Hans Peter Graf. A programmable parallel accelerator for learning and classification. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, pages 273–284, 2010.

[14] CERN. Major work starts to boost the luminosity of the lhc. `https://home.cern/news/press-release/accelerators/major-work-starts-boost-luminosity-lhc-0`, 2018. Accessed: 2024-05-29.

[15] Philippe Coussy and Adam Morawiec. *High-level synthesis*, volume 1. Springer, 2010.

[16] Sheng Di and Franck Cappello. Fast error-bounded lossy hpc data compression with sz. In *2016 ieee international parallel and distributed processing symposium (ipdps)*, pages 730–739. IEEE, 2016.

[17] Javier Duarte, Song Han, Philip Harris, Sergo Jindariani, Edward Kreinar, Benjamin Kreis, Jennifer Ngadiuba, Maurizio Pierini, Ryan Rivera, Nhan Tran, et al. Fast inference of deep neural networks in fpgas for particle physics. *Journal of Instrumentation*, 13(07):P07027, 2018.

[18] Farzad Ebrahimi, Matthieu Chamik, and Stefan Winkler. Jpeg vs. jpeg 2000: an objective comparison of image encoding quality. In *Applications of Digital Image Processing XXVII*, volume 5558, pages 300–308. SPIE, 2004.

[19] Farah Fahim, Benjamin Hawks, Christian Herwig, James Hirschauer, Sergo Jindariani, Nhan Tran, Luca P. Carloni, Giuseppe Di Guglielmo, Philip Harris, Jeffrey Krupa, Dylan Rankin, Manuel Blanco Valentin, Josiah Hester, Yingyi Luo, John Mamish, Seda Orgrenci-Memik, Thea Aarrestad, Hamza Javed, Vladimir Loncar, Maurizio Pierini, Adrian Alan Pol, Sioni Summers, Javier Duarte, Scott Hauck, Shih-Chieh Hsu, Jennifer Ngadiuba, Mia Liu, Duc Hoang, Edward Kreinar, and Zhenbin Wu. hls4ml: An open-source codesign workflow to empower scientific low-power machine learning devices, 2021.

[20] Umer Farooq, Zied Marrakchi, and Habib Mehrez. *Tree-based heterogeneous FPGA architectures: application specific exploration and optimization*. Springer Science & Business Media, 2012.

[21] FastML Team. fastmachinelearning/hls4ml. `https://github.com/fastmachinelearning/hls4ml`, 2023. Accessed: 2024-02-11.

[22] A Fidler, B Likar, and U Skalerič. Lossy jpeg compression: easy to compress, hard to compare. *Dentomaxillofacial Radiology*, 35(2):67–73, 2006. PMID: 16549431.

[23] Gnu. Gnu gzip: General file (de)compression. `https://www.gnu.org/software/gzip/manual/gzip.html`.

[24] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016. `http://www.deeplearningbook.org`.

[25] The HDF5 Group. Hdf5. `https://www.hdfgroup.org/solutions/hdf5/`.

[26] Andrei Hagiescu and David Cashman. How oneapi is making fpgas more accessible than ever. `https://www.intel.com/content/www/us/en/developer/articles/technical/accelerate-compression-on-intel-fpgas.html`, 2020. Accessed: 2024-01-12.

[27] hls4ml. Vivadoaccelerator backend. `https://fastmachinelearning.org/hls4ml/concepts.html`.

[28] hls4ml. Vivadoaccelerator backend. `https://fastmachinelearning.org/hls4ml/advanced/accelerator.html`.

[29] Intel. Intel quartus. `https://www.intel.com/content/www/us/en/products/details/fpga/development-tools/quartus-prime.html`.

[30] Intel. Intel® core™ i7-2600k processor. `https://www.intel.com/content/www/us/en/products/sku/52214/intel-core-i72600k-processor-8m-cache-up-to-3-80-ghz/specifications.html`.

[31] JPEG. Overview of jpeg 1. `https://jpeg.org/jpeg/index.html`.

[32] Wei-Meng Lee. *Python machine learning*. John Wiley & Sons, 2019.

[33] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. Error-controlled lossy compression optimized for high compression ratios of scientific datasets. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 438–447, 2018.

[34] Roger Lipsett, Carl F Schaefer, and Cary Ussery. *VHDL: Hardware description and design*. Springer Science & Business Media, 2012.

[35] Zdeněk Matěj, Rajmund Mokso, Krister Larsson, Vincent Hardion, and Darren Spruce. The max iv imaging concept. *Advanced Structural and Chemical Imaging*, 2:1–7, 2016.

[36] Jean Mermet. *Fundamentals and standards in hardware description languages*, volume 249. Springer Science & Business Media, 2012.

[37] NumPy. Numpy array. `https://numpy.org/doc/stable/reference/generated/numpy.array.html`.

[38] PyTorch. Pytorch. `https://pytorch.org`.

[39] C. Raufaste, B. Dollet, K. Mader, S. Santucci, and R. Mokso. Three-dimensional foam flow resolved by fast x-ray tomographic microscopy. *EPL (Europhysics Letters)*, 111(3):38004, August 2015.

[40] RAYPCB. Top 8 fpga manufacturers in the world. https://www.raypcb.com/fpga-manufacturers/.

[41] Khalid Sayood. *Introduction to data compression*. Morgan Kaufmann, 2017.

[42] Tim Smith. An exabyte of disk storage at cern. https://home.cern/news/news/computing/exabyte-disk-storage-cern, 2023. Accessed: 2024-01-11.

[43] Hayden So. Introduction to fixed point number representation. https://inst.eecs.berkeley.edu/~cs61c/sp06/handout/fixedpt.html.

[44] Donald Thomas and Philip Moorby. *The Verilog® hardware description language*. Springer Science & Business Media, 2008.

[45] Jiannan Tian, Sheng Di, Chengming Zhang, Xin Liang, Sian Jin, Dazhao Cheng, Dingwen Tao, and Franck Cappello. wavesz: a hardware-algorithm co-design of efficient lossy compression for scientific data. In *Proceedings of the 25th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '20, page 74–88, New York, NY, USA, 2020. Association for Computing Machinery.

[46] TomoBank. Foam data. https://tomobank.readthedocs.io/en/latest/source/data/docs.data.dynamic.html#foam-data.

[47] William Wen. Introduction to torch.compile. https://pytorch.org/tutorials/intermediate/torch_compile_tutorial.html.

[48] Michael Wu. Why is there so much statistical redundancy in big data? https://www.linkedin.com/pulse/why-so-much-statistical-redundancy-big-data-michael-wu-phd/. Accessed: 2024-02-15.

[49] Xilinx. Adapting computing technology overview. https://www.xilinx.com/content/dam/xilinx/publications/technology-briefs/adaptive-computing-technology-overview.pdf.

[50] Xilinx. Zcu104 evaluation board user guide. https://www.xilinx.com/support/documentation/boards_and_kits/zcu104/ug1267-zcu104-eval-bd.pdf.

[51] Qingqing Xiong, Rushi Patel, Chen Yang, Tong Geng, Anthony Skjellum, and Martin C Herbordt. Ghostsz: A transparent fpga-accelerated lossy compression framework. In *2019 IEEE 27th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 258–266. IEEE, 2019.

[52] Si-Si Zhang, Jianwei Liu, Xin Zuo, Runkun Lu, and Siming Lian. Online deep learning based on auto-encoder. *CoRR*, abs/2201.07383, 2022.

[53] Noah Zipper, CMS collaboration, et al. Testing a neural network for anomaly detection in the cms global trigger test crate during run 3. *Journal of Instrumentation*, 19(03):C03029, 2024.

Printed by Tryckeriet i E-huset, Lund 2024

LUND UNIVERSITY

Series of Master´s theses
Department of Electrical and Information Technology
LU/LTH-EIT 2024-1005
http://www.eit.lth.se