The 27th LSI DESIGN CONTEST in OKINAWA 2024

（高信頼知的集積システム研究センターシンポジウム）

2024年コンテストテーマ:
Autoencoder

開催日: 令和6年3月8日(金)
沖縄県石垣市、ユーグレナ石垣港離島ターミナル

主催: LSIデザインコンテスト実行委員会，高信頼知的集積システム研究センター

共催: 琉球大学工学部，九州工業大学情報工学部

協賛: 電子デバイス産業新聞（旧半導体産業新聞），ギガファーム株式会社，
電子情報通信学会スマートインフォメディアシステム研究会，

後援: 九州職業能力開発大学校，沖縄職業能力開発大学校，CQ出版社

http://www.lsi-contest.com

目次

コンテストの概要・目的

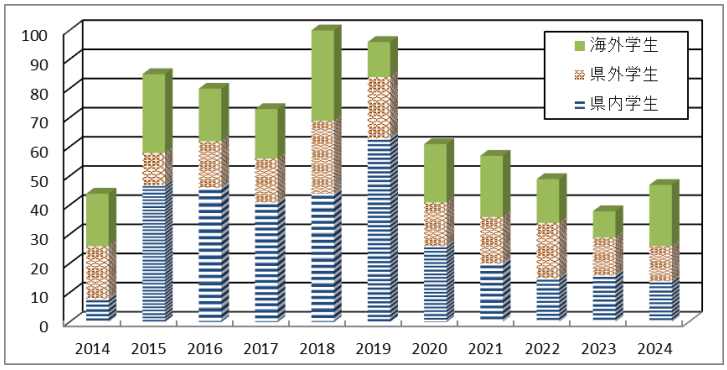
九州・沖縄さらに東南アジア地域の半導体産業および組み込み機器等のエレクトロニクス産業の振興を目的に、標題の学生向けの設計コンテストを毎年実施しております。主催は、琉球大学および九州工業大学の教員で構成するLSIデザインコンテスト実行委員会で実施しております。2024年は、国内外を含め45名を超えた応募がありました。

東シナ海および南シナ海沿岸の LSI 産業拠点地域（九州、韓国、台湾、中国、シンガポール、フィリピン、マレーシア）は、世界の半導体市場の実に 40 %を越える高いシェアを有しており（九州以東の日本本土は含みません！）、その中心に位置する沖縄はそのビジネスチャンスが大変高いものと期待されます。そうした地理的好条件の沖縄にて、学生向け LSI 設計コンテストを実施し、学生すなわち未来のエンジニアの設計スキルを上げることにより、将来的に沖縄や東南アジアでの企業誘致やベンチャー起業につなげたいと考えております。

今回は、AI の一つである「Autoencoder」がテーマになります。処理の高速化，回路規模の削減を目指したハードウェア設計など、いろいろなアイデアが生まれることを期待しております。

本コンテストの主旨をご理解頂き、多くの学生の皆様（高専、大学、大学院生）の参加を期待しております。

学生参加数の推移



国内

琉球大学、千葉大学、九州工業大学、会津大学、大分県立工科短期大学、沖縄職業能力開発大学校、大阪工業大学、大阪大学、京都大学、近畿大学、高知工科大学、神戸大学、芝浦工業大学、職業能力開発大学校、東海大学、東京工業大学、東京都立科学技術大学、東京理科大学、東北大学、徳島大学、豊橋技術科学大学、長崎大学、新潟大学、広島大学、法政大学、山形大学、横浜国立大学、立命館大学、早稲田大学、沖縄高専、有明高専、木更津高専、久留米高専、豊田高専、北陸先端科学技術大学院大学

海外

Bandung Institute of Technology 、 Institut Teknologi Telkom 、 Telecommunication College Bandung 、Telkom University （インドネシア）、Univ. of Science、Ho Chi Minh city、Hanoi Univ. of Science （ベトナム）、Chosun 大学（韓国）、Univ. of Valladolod（スペイン）、Univ. of Kaiserslautern（ドイツ）、NTI（エジプト）、McMaster 大学（カナダ）、南西大学（中国）

これまでの設計課題

- 2001 年： 「デジタル CDMA レシーバ」
- 2002 年： 「差集合巡回符号エラー訂正回路」
- 2003 年： 「静的ハフマン符号用の可変長デコーダ」
- 2004 年： 「共通鍵暗号 AES 用 SubByte 変換回路」
- 2005 年： 「デジタル FM レシーバ」
- 2006 年： 「2 次元積符号繰り返しデコーダ」
- 2007 年： 「64 点高速フーリエ変換」
- 2008 年： 「RSA 暗号デコーダ」
- 2009 年： 「Small RISC Processor」
- 2010 年： 「エラー訂正：BCH 符号」
- 2011 年： 「DCT」
- 2012 年： 「16/64/128-point Flexible FFT」
- 2013 年： 「SW・HW 協調設計を用いたノイズ除去システム」
- 2014 年： 「SW・HW 協調設計を用いたノイズ除去システム」
- 2015 年： 「正弦波発生回路」
- 2016 年： 「人物検出用パターンマッチング回路」
- 2017 年： 「人物検出動画像処理システム」
- 2018 年： 「ニューラルネットワーク（バックプロパゲーション）」
- 2019 年： 「深層学習（バックプロパゲーション）」
- 2020 年： 「畳み込みニューラルネットワーク（CNN）」
- 2021 年： 「強化学習」
- 2022 年： 「Deep Q-Network」
- 2023 年： 「局所的最大値／最小値」
- **2024 年： 「Autoencoder」**

開 催 概 要

名　　称　：「第 27 回 LSI デザインコンテスト in 沖縄 2024」
実行委員長　：　九州工業大学大学院　情報工学研究院　情報・通信工学研究系　教授　尾知博
主　　催　：　LSI デザインコンテスト実行委員会、高信頼知的集積システム研究センター
　　http://www.lsi-contest.com/
共　　催　：　琉球大学工学部、九州工業大学情報工学部
協　　賛　：　電子デバイス産業新聞（旧半導体産業新聞）、
　　ギガファーム株式会社、電子情報通信学会スマートインフォメディアシステム研究会、
後　　援　：　CQ 出版社、
　　九州職業能力開発大学校、沖縄職業能力開発大学校
実行事務局　：　九州工業大学情報工学部情報・通信工学科尾知研究室
　　LSI デザインコンテスト実行委員会事務局
日　　時　：　2024 年 3 月 8 日（金）　13:00～18:00
会　　場　：　〒907-0012　石垣市美崎町 1 番地
　　ユーグレナ石垣港離島ターミナル
目　　的　：　実践的な課題を用いた学生対象のデジタル集積回路設計のコンテストであり、半導体集積回路設計能力の向上とともに国際的に交流することで学生の工学に関する視野を広めることを目的としている。
対　　象　：　国内大学・大学院生、高専学生、アジア地域大学生
来場予定者　：　100 名（入場無料）
募集方法　　：　各大学・高専へのパンフレット送付、LSI 関連雑誌等へのリリース
ホームページ：　http://www.LSI-contest.com/

【審査員】
実行委員長：　九州工業大学　尾知 博
審査委員長：　琉球大学　和田 知久
東京大学　藤田 昌宏
大阪大学　尾上 孝雄
千歳科学技術大学　宮永 喜一
鳥取大学　伊藤 良生
九州工業大学　黒崎 正行
九州工業大学　Leonardo Lanante Jr.
琉球大学　吉田 たけお
ディポネゴロ大学　Wahyul Shafei Amien
バンドン工科大学　Trio Adiono
九州職業能力開発大学校　岡田正之
沖縄職業能力開発大学校
他協賛企業・法人審査員　　（敬称省略・順不同）

連絡先
住　所：〒820-8502 福岡県飯塚市川津 680-4　九州工業大学大学院情報工学研究院情報・通信工学研究系
電　話：0948-29-7667　　Email: support@LSI-contest.com
LSI デザインコンテスト実行委員会委員長
担当者　尾知　博

【設計テーマ】　Autoencoder

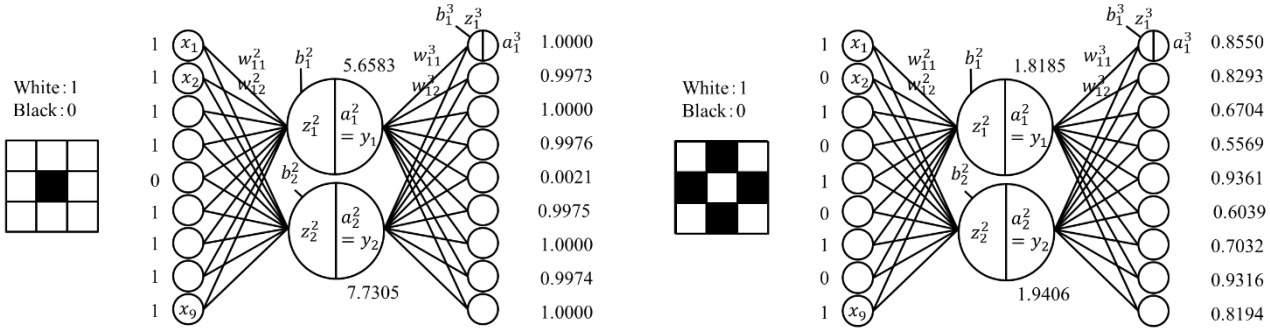
第27回のテーマは、AIの中でも異常堅守や雑音除去などに応用されているオートエンコーダーです。今回は「Autoencoder」をテーマとして、処理の高速化、回路規模の削減を目指したハードウェア設計を行うことを目的とします。Level1の課題では、3×3の○×を判定する回路を設計します。Level2の課題では、画素数を増やした任意の画像に対するAutoencoderの回路を設計します。Level3の課題では問題や構造はすべて自由とし、よりユニークなAutoencoderを使用したシステムを設計します。

要求されている設計は、HDL（VHDLもしくはVerilogHDL）による設計と論理合成および検証結果です。

実装アルゴリズムと環境

■　Autoencoder の概要

Autoencoder は次元を圧縮するようなエンコーダーと復元を行うデコーダーとで構成されています。
例として，3×3 の画像における Autoencoder の構成図を示します。



■　実装環境

Synopsys® Synplify Pro® /Premier
Synopsys® Design Compiler®
Mathworks® MATLAB® /Simulink®
Xilinx Vivado® Design Suite
または設計環境に応じて、Synplify Pro/Premier or その他論理合成ツール、RTL hand coding(VHDL or Verilog-HDL)などの設計環境が挙げられます。

課題

- 1. Level1:初心者向け

３×３の○×を判定せよ

・ハードウェア設計のアウトライン

・シミュレーション用の verilog-HDL　ファイル

・ハードウェア設計のアウトラインの続き(ブロックの詳細)

- 2. Level2:中級者向け

素数を増やして任意の画像（黒白、グレースケール)

・ハードウェア設計のアウトライン

・シミュレーション用の verilog-HDL　ファイル

・ハードウェア設計のアウトラインの続き(ブロックの詳細)

- 3. Level3:上級者向け

unlimited (VAE なども歓迎いたします)

・ハードウェア設計のアウトライン

・シミュレーション用の verilog-HDL　ファイル

・ハードウェア設計のアウトラインの続き(ブロックの詳細)

審査：JUDGE

■　審査メンバーによる以下の 4 項目各 10 点で審査を実施（10 point each）

1）アカデミック的、新奇アイデア的な観点（Academic, New Idea）

2）実用設計、産業面応用的な観点（Used in real life, Good for industry）

3) FPGA 等の実装レベルの観点（Good prototype by FPGA etc.）

4）プレゼンテーションの観点（Good presentation）

表彰：AWARD

■　優勝（電子情報通信学会 SIS 賞）SIS AWARD

【アカデミック的、新奇アイデア的な観点】の BEST

■　その他、2)、3)、4)の観点からも賞を贈呈します。

自家用車の異常音検知システムの開発

チーム名：まっちゃ
具志堅 胡春，　知花 樹，　與那覇 孝矢，　平良 智輝

Abnormal Sound Sensing System for Cars

Koharu Gushiken,　Tatuki Tibana,　Kouya Yonaha,　Tomoki Taira
Department of Advanced Electronics Information Technology for Production System,　Okinawa Polytechnic College
2994-2 Ikehara, Okinawacity Okinawa
Email address：j2321307@okinawa-pc.ac.jp　j2321314@okinawa-pc.ac.jp
J2321318@okinawa-pc.ac.jp　j2221313@okinawa-pc.ac.jp

Abstract: This system records the sound of a car engine, determines whether there is anything wrong with the engine sound, and notifies the driver.
To keep ambient noise from affecting the judgment result, noise elimination is performed using MATLAB, and FPGA is used to determine whether it is normal or abnormal.
Keyword　:　Car Engine,　Noise Elimination,　MATLAB,　FPGA

1. はじめに

現在，沖縄では主な移動手段が自家用車であり，日々様々なドライバーが運転している．沖縄では自家用車以外の交通手段が少ないこともあり，急に故障してしまうと通勤や通学，買い物などの生活に支障をきたしてしまう．そこで私たちは車の異常音を検出し，車に詳しくない人でも事前に異常を予測できるシステムの開発を行うことにした．

2. 開発環境

本システムの開発環境を表 1 に示す．MATLAB でノイズ除去および判定結果の通知を行い，FPGA を用いてエンジン音の正常異常判定を行う．

表 1　開発環境

OS	Windows10
ソフトウェア	MATLAB R2022A VIVADO 2020.3 Vitis Model Composer 2022.2
ハードウェア	ZYBO_Z7 zynq-7010(DIGLENT 社製 FPGA)

3. システム概要

本システムは自動車のエンジン音を録音し，エンジン音に異常がないかを判断して運転手に知らせるシステムである．

環境音が判定結果に影響を与えないように MATLAB でノイズ除去を行う．オートエンコーダを実装した FPGA を使用し，エンジン音の正常と異常を判定する．伝達手段は，MATLAB を用いて音声や文字で出力する．

本システムの全体構成を図 1 に示す．

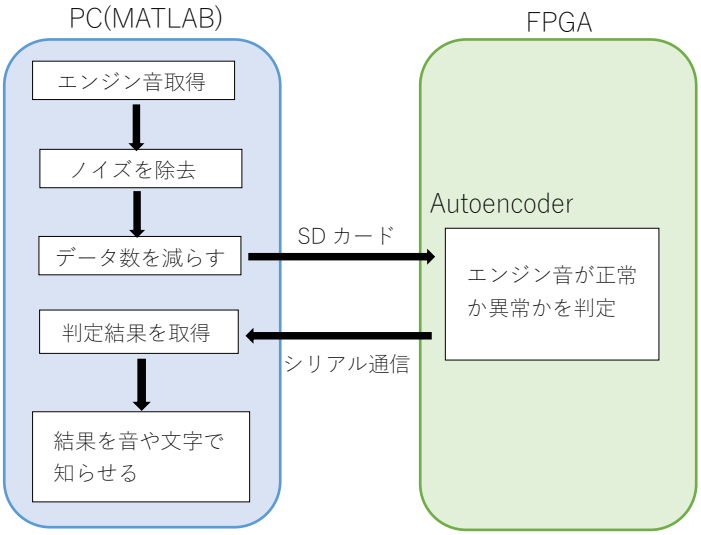


図 1　全体構成

4. システム実装

システムの処理の流れを以下に示す．

4.1. エンジン音の取得

エンジンのフロントグリル部分に取り付けたマイクから日時の違う 8 パターンのエンジン音を取得する．実験ではスマートフォンで録音したエン

ジン音を使用した．データのフォーマットは「.m4a」，エンジンはダイハツの KF エンジンを対象とした．

4.2. エンジン音のノイズ除去及びデータ数の削減

MATLAB を用い，以下の手順でエンジン音データを加工した．

① 録音したエンジン音（データ数約 270,000）を入力する．

② 正確な判定結果となるように，カットオフ周波数を 1000Hz に設定したハイパスフィルタを使用して，風の音や周囲の雑音を除去する．

③ データ数を減らすために，音声データの特定の時間範囲（1.0 秒から 1.5 秒の間の 0.5 秒）を取り出し（データ数約 24,000），8000Hz のサンプリング周波数でフーリエ変換を行う（データ数 8,000）．

④ さらにデータ数を減らすために，フーリエ変換したデータを平均化する（データ数 8）．

⑤ 最終的に 8 パターンの音声データすべてのデータ数を削減し，8×8 のデータにした．

4.3. エンジン音の正常異常判定（FPGA）

入力層 64 層，中間層 6 層，出力層 64 層のオートエンコーダを FPGA に実装し，以下の手順で検証を行った．

① 正常なエンジン音の特徴量をオートエンコーダに通して学習させる．

② 学習した特徴量をもとに，テスト用の異常なエンジン音をエンコード・デコードし，再構築データを生成する．

③ オートエンコーダが生成した正常なエンジン音の再構築データと，テスト用の異常なエンジン音の再構築データの誤差を計算する．

④ 閾値を設定し，誤差の値が閾値よりも小さい場合は正常，大きい場合は異常と判定する．

4.4. シリアル通信・SD カード

PC・FPGA ボード間のデータのやり取りはシリアル通信と SD カードの入れ替えで行う．

ノイズ除去を行ったエンジン音を SD カードに保存，SD カードを FPGA ボードに入れ替え，正常異常の判定結果をシリアル通信で MATLAB に送信することでデータのやり取りを行う．

5. 検証結果

MATLAB を使用し検証を行った際のパラメータとその結果を以下に示す．

5.1. パラメータ

今回の検証で使用したパラメータを表 2 に示す．

表 2　使用したパラメータ

学習率	学習回数	中間層	閾値
0.01	10000	6	0.35

5.2. 正常な音声データ

以下に正常な音声データを学習したオートエンコーダに，正常なエンジン音をテストデータとして入力した際のグラフを示す．

横軸は周波数（平均化により 8000Hz から縮小した）で縦軸はその周波数での強さを正規化したものである．

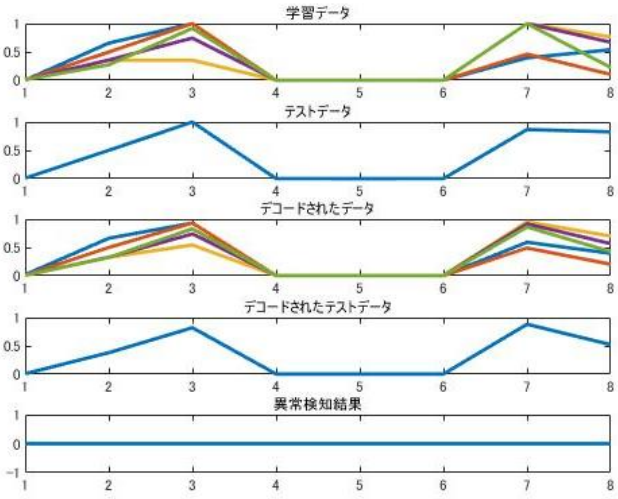


図 2　正常な音声データ

- 学習データ

  学習用として用意した 8 個の音声データを平均化して特徴を取り出した波形
- テストデータ

  学習させていない正常な音声データを平均化して特徴を取り出した波形
- デコードされたデータ

  入力した学習データをオートエンコーダで再構築したデータ
- デコードされたテストデータ

  入力したテストデータをオートエンコーダで再構築したデータ
- 異常検知結果

  再構築したデータの誤差が閾値を超えたかどうか，すなわち異常の有無を図示している

図 2 から，「異常検知結果」において，誤差が閾値を超えた時に出力される 1 が出力されていないことから「正常」と判定していることがわかる．

5.3. 異常な音声データ

正常な音声データを学習したオートエンコーダに Youtube に投稿されていた異常な音声データ[1]を入力した際のグラフである．この図から「異常検知結果」では，誤差が閾値を超えた時に出力される 1 が出力されている箇所がある．これは正常な音声データとの誤差が発生しているということであり，「異常」と判定していることがわかる．

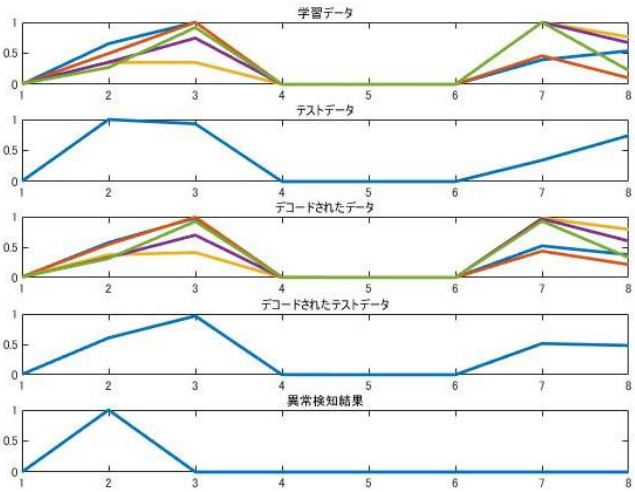


図 3　異常な音声データ

図 4 は図 3 で異常が検知されていた横軸 X の 2 に注目して数値を表示させたものである．

X の 2 の数値ではテストデータは 0.999…となっているが，デコード後は 0.606…となっている．もともとの学習データは X の 2 が高い数値ではないため，その特徴を学習し，テストデータもそのように修正している．

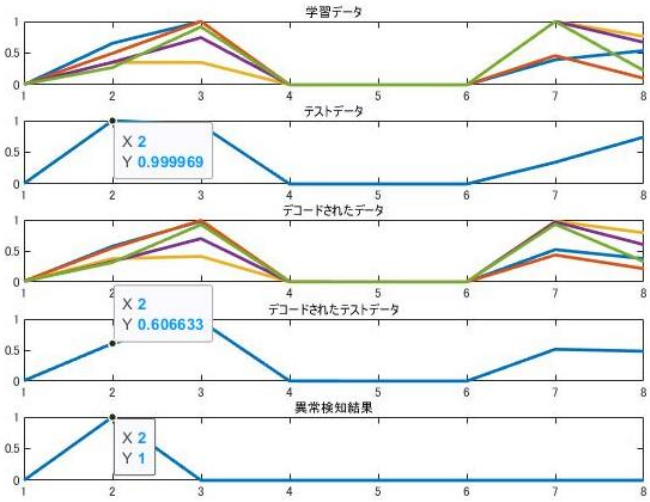


図 4　異常な音声データ（数値）

図 5 はテストデータの値とデコード後の値の差を求めたグラフである．X の 2 が 0.393…となっており，これは先ほどの 0.999 と 0.606 を引いた値である．今回は閾値を 0.35 としており，その数値を上回ったため，異常と判定された．

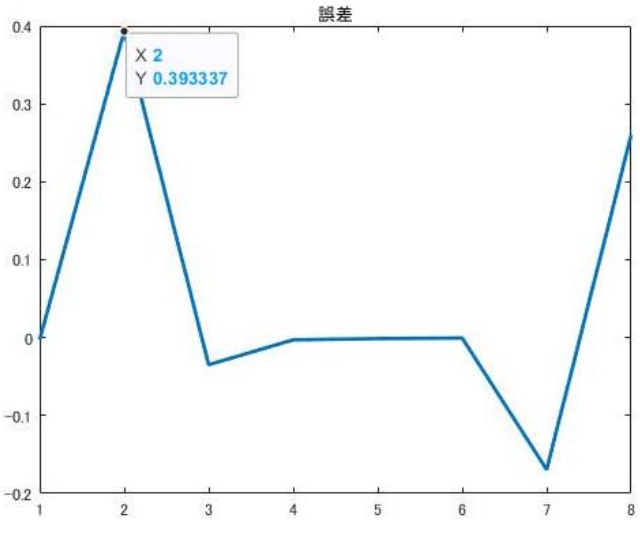


図 5　誤差

5.4. シミュレーション検証結果

MATLAB でシミュレーションした結果を表３に示す．この表は，学習したオートエンコーダに，正常音と異常音のテストデータを入力した場合の異常検知の成功率を表している．それぞれ 10 個のテストデータを用意し，検証行ったところ，正常音，異常音どちらも 70%の成功率となった．

表３　シミュレーション検証結果

テストデータ	正常音	異常音
判別成功	7/10	7/10
成功率	70%	70%

6. おわりに

今回，オートエンコーダを用いたエンジン音の異常判定を行うことができた．しかし，現在は MATLAB 上で音声を加工し異常検知も行っているため，発表までに異常検知を FPGA ボードへ実装し，検証きるよう進めていきたい．

参考文献

[1]KF エンジン異音（エッセ編走行 8 万ｷﾛ）/Youtube
https://youtube.com/shorts/r8GV_INNQ98?si=SKMXGIufJPgPV5L8

学習プリント採点システムの開発

チーム名：複素数
伊禮 海都　上江田 勇得　新里 友大

Development of learning print scoring system

Kaito Irei，　Samueru Ueda，　Tomohiro Shinzato,
Department of Advanced Electronics Information Technology for Production System，
Okinawa Polytechnic College
2994-2 Ikehara, Okinawacity, Okinawa, 904-2141, Japan
Email address：j2321303@okinawa-pc.ac.jp　j2321304@okinawa-pc.ac.jp
j2321311@okinawa-pc.ac.jp

Abstract: Teachers at schools and cram schools need to grade worksheets, but the large amount of worksheets and illegible characters make grading time consuming. To solve this problem, we developed a study worksheets grading system that uses an autoencoder to identify the characters in the answers and reduce the grading time.

I. 初めに

学校や塾などで教師は学習プリントを採点する必要があるが, プリントの量が多いことや読み取りにくい文字があるといった原因で採点の作業に時間がかかってしまう.

そこで採点時間を短縮するために, 我々は Autoencoder を用いて解答の文字判別を行い, 学習プリントの採点システムを作成した.

II. システムの概要

システムの入力は生徒が記入した解答の画像とし, 後述する画像処理を Matlab 上で行い, 値の抽出後に二値化と圧縮を行う. その後 FPGA ボードに値を送信し, FPGA ボード上で Autoencoder による文字判別を行う. 判別した数字を Matlab へと送信し, 学習プリントの正誤判定を行う. 解答のデータは, テキストで保存され, 学生の学習記録として残す.

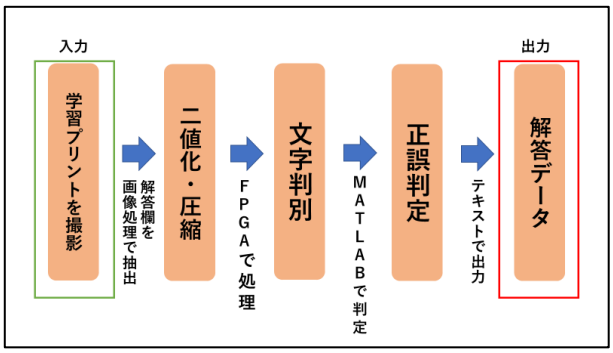


図 1　システムの概要

III. 開発環境

本システムの開発環境を表 1 に示す.
シミュレーションに MATLAB, 評価ボードは ZYBO Z7 zynq-7010 , ハードウェア設計に MATLAB, Vivado を使用する.

表 1. 開発環境

OS	Windows11
開発ソフト	MATLAB 2022a Vitis 2022.2 Vivado 2022.2
評価ボード	DIGILENT 社製 ZYBO_Z7 zynq-7010
言語	MATLAB C 言語

IV. 画像処理

学習プリントの回答を所定の場所に置き, WEB カメラで学習プリントを撮影する. Matlab で画像処理により解答箇所を抽出し, 画像データとして保存する.

学習プリントの解答は選択形式を想定し, 数字の 1～5 を解答として記入することとする. 取得したプリント解答の画像は, 数字の判別を行いやすくするため Matlab 上で文字の黒色と, プリントの白色を色反転させ 2 値化を行う. 数字を表現することができる最小のデータ量である 5x7 画素の画像に圧縮する. 画像処理の流れを図 2 に示す.

図 2　画像処理の流れ

V. AUTOENCODER による機械学習

A. *教師データとテストデータ*

Autoencoder による学習は, FPGA ボード上で行う. 実験として Matlab 上で数字 1 と数字 2 の画像を機械学習させた. 数字 1 を例に機械学習の結果を説明する. 入力データは全て 35(5x7)画素で行った.

以降の図で表している画像データは, 各画素が 0~1 の範囲の値となっており, 白が 1, 黒が 0 を表す.

図 3 は数字 1,数字 2 の教師データをそれぞれ示す.

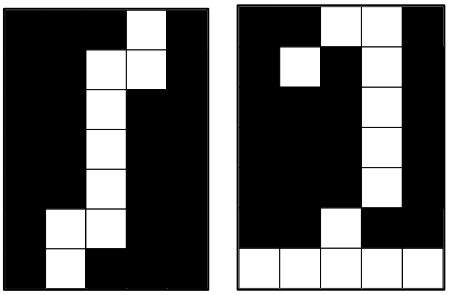


図 3　数字 1,数字 2 の教師データ

作成した数字 1 のテストデータを図 4 に示す. 図 4 にある 4 つの画像データは, 数字 1 の正しい画像データから 1 画素が欠けている誤りのデータである.

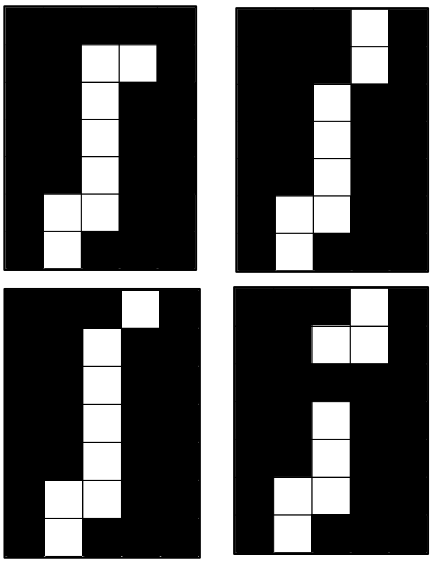


図 4　数字 1 のテスト用誤りデータ

B. *学習のパラメータについて*

Autoencoder を実行させる際に最適なパラメータを導出するため実験を行った. 対象のパラメータは, epoch(学習回数), eta(学習率), 中間層の要素数である. 学習後に画像データが明確な数字として出力された時の値を用いることにした.

1) epoch(学習回数)

epoch の値を以下の表 2 のように変化をさせた.

表 2. epoch の変化(eta=0.03, 中間層の要素数=3 の時)

1000	5000	8000	15000	20000	30000

まず epoch=1000 に設定すると数字と認識しにくい画像データが出力される。図 5 は epoch=10000 の機械学習ののち, 図 4 の誤りのテストデータを渡したときの出力結果である。

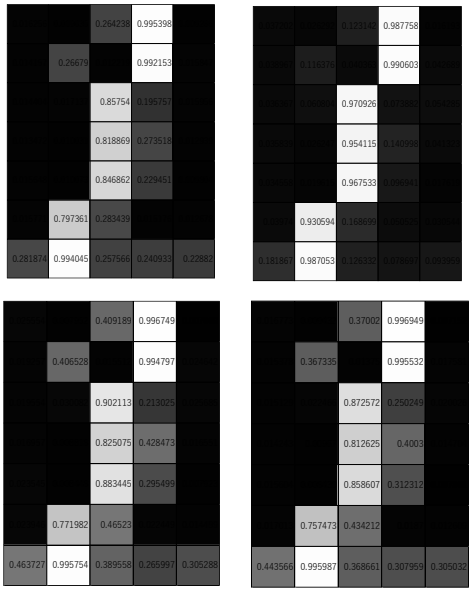


図 5　epoch=1000 の時の出力結果

epoch は値が大きいほど, 画像データの数字がより鮮明に表示された. しかし、不必要な学習回数をさせないため, 十分に数字として認識できる epoch= 5000 に値を設定した.

2) eta(学習率)

eta の値を以下の表 3 のように変化をさせた.

表 3. eta の変化(epoch=5000, 中間層の要素数=3 の時)

0.005	0.007	0.01	0.02	0.03	0.04

eta は値を大きくするほど, 画像データの数字がより鮮明に表示されたが, 0.04 にすると, ノイズが入り, 数字として認識しにくい画像データが出力された.

3) 中間層の要素数

中間層の要素数を以下の表 4 のように変化させた.

表 4. 中間層の要素数(epoch=5000, eta=0.03 の時)

2	3	4

中間層の要素数は, 3 つにした時が最も数字に見える出力結果になった. よって, この実験結果に基づいて, 各パラメータの値は上記の通りが最適であると考えた.

C. 選定したパラメータによる機械学習

上記の実験をもとに選定した epoch, eta, 中間層の要素数それぞれのパラメータは以下の通りとなった.

epoch(学習回数) = 5000
eta(学習率) = 0.03
中間層の要素数 = 3

選定した上記のパラメータの値で Autoencoder を設定したのち機械学習を行った. 機械学習後, 図 4 のテストデータを渡したときの出力結果は以下の図 6 に示す. 図 4 の誤りのテストデータに対して復元ができていることが確認できる.

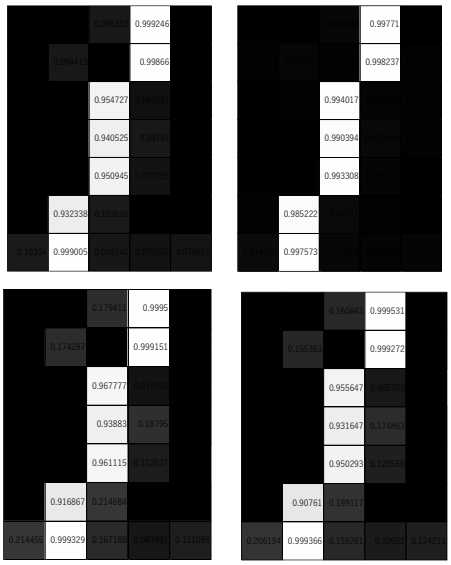


図 6　機械学習後の数字 2 の出力結果

VI. 正誤判定

FPGA ボードで文字判別後, 取得した値はテキストへ保存される.その値を 正答データのテキストと照合し, 正答数や間違えた問い, 正答を指定の場所に保存する.以下の図 6,7 に正誤判定の実行結果を示す.

```
正解数: 4
不正解の問題: 問5
不正解の問題とその番号を Incorrect.txt に保存しました。

ans =

    '正解数: 4'
```

図 7　正誤判定プログラムの実行結果

```
不正解の問題とその番号:
問5: 1 (答え: 5)
```

図 8　保存させるテキストファイル

VII. ・評価

今回作成するにおいて, 画像処理,機械学習,正答判定の三つに分けて作業を進め, 以下の表に制作物の評価を記載する.各評価について, 画像処理と正答判定に関しては問題なく動作しているので評価を○とした.

機械学習に関しては, 現状数値が数字 1 と 2 のみでの学習実験しか行っていない為に評価を△とした.

各システムへの値の受け渡しについては時間の都合上実装できておらず評価は×とした.

FPGA の実装について, 重みや入力などのゲートウェイを設定している段階なので, △としている. Matlab 上での機械学習の試験的な実装に時間がかかったため, FPGA への実装が遅れてしまった.

表 5 制作物の評価

評価	
画像処理	○
機械学習	△
正答判定	○
各システムへ値の受け渡し	×
FPGAへの実装	△

VIII. ・終わりに

LSI コンテストに向けてのシステム開発を通して, Autoencoder についてより理解を深めることができた. 時間的な都合により, 画像処理, 機械学習, 正答判定の統合はできなかったが,それぞれの機能を MATLAB 上でシミュレーションし動作を確認することはできた. 今後は, すべての機能を統合した上で GUI の追加など利用者がよりシステムを使いやすくなるような工夫を施していきたい.

Development of Music Note Analysis and Performance System

―Music Information Processing and Integration of Autoencoders―

1st Karin Kanamaru
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
1665-1 Shii Kokuraminam-iku
Kitakyushu-shi Fukuoka, Japan
2217104@kyushu-pc.ac.jp

2nd Nanami Yoneoka
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
1665-1 Shii Kokuraminam-iku
Kitakyushu-shi Fukuoka, Japan
2217132@kyushu-pc.ac.jp

I. はじめに

音楽教育における読譜指導は、生徒の音楽に対する興味関心を喚起し、音楽的素養を高める上で欠かせない要素である。しかし近年、教育現場における時間的制約や教員の意識の低さなどの要因により、読譜教育が後回しにされる状況が生じている。こうした課題に対処するべく、本研究では効率的な読譜学習を可能にする手法の開発を目的とする。

具体的には、低年齢層を対象とした、興味関心を引き出しやすい簡便な学習システムの構築を試みる。そのためにオートエンコーダを応用し、楽譜画像を入力データとして用い、これをもとに自動演奏を生成する機構を設計した。このようなインタラクティブな学習法は、楽しみながら読譜能力を習得できることが期待される。

本研究のもう一つの目的は、LSI デザインコンテストへの出場である。コンテストではデータ圧縮・復元、特徴抽出の効率化が求められていることから、入力された楽譜画像から特徴を抽出し、適切な音を出力する高度なオートエンコーダを開発した。これは、オートエンコーダの応用可能性を広げるとともに、音楽教育への寄与も期待できる意欲的な試みである。

II. 概要

本研究の目的は、音楽への関心を高めることができる学習支援機器の開発にある。そのためにオートエンコーダを応用し、楽譜画像を入力データとして用い、これを解析することで自動演奏を実現するシステムを構築した。

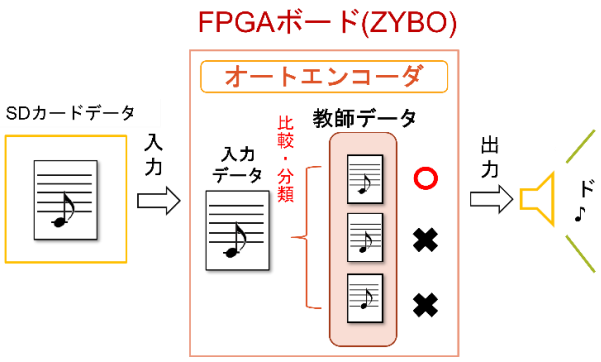
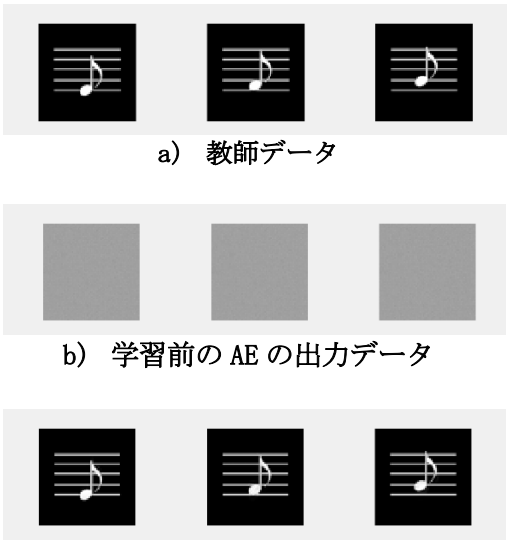


図 1. 動作の流れ

当初は単純な音符画像のみから音の特性を識別するプロトタイプの開発を想定していたが、より高度な認識を実現するため、本研究では音符と対応する音高情報をペアで含む画像データを採用した。これらの楽譜画像は SD カード経由でシステムに入力され、事前に準備された教師データとの比較に基づいて、最適な音が決定される。

出力にはスピーカーを用いた聴覚的表現と、ピアノ型キーボードの LED ライトによる視覚的表現の 2 つを実装した。(図 1 参照)



a) 教師データ

b) 学習前の AE の出力データ

c) 学習後の AE の出力データ

図 2. 学習前後の比較画像

オートエンコーダの性能向上を図るため、教師データである楽譜画像に対して前処理を施し、解析・整形を行った。図 2 は、教師データ、学習前後のモデル出力結果を示したもので、学習の効果を視覚的に確認できる。図 2(c)から、出力画像が教師データに近づいていることが分かり、モデルの学習能力と画像再現精度が向上したことが検証された。

このように、音楽と情報工学を融合した本研究は、教育応用と技術革新の両面で意義深い成果が期待できる。

III. 演奏機器装置

開発した演奏支援機器の構造と機能について、図 3 に示した。本装置はピアノ型のデザインを採用しており、FPGA ボード、アンプ内蔵スピーカー、LED 基板などのコンポーネントを内包している。

外形や鍵盤の設計には 3D CAD ソフトを用い(図 4 参照)、アクリルと 3D プリント技術によって製作を行った。鍵盤は、解析された楽譜データに基づいて対応する LED が点灯する仕組みで、どの音が演奏されているかを視覚的に確認できる。LED 基板は鍵盤下部に配置し、精密な位置調整を行った。

LED への入力信号はソレノイドにも供給可能であるため、鍵盤を自動的に打鍵させる応用が可能である。これにより単なる視覚補助的機能にとどまらず、自動演奏システムとしての高度な音楽表現能力が実現できることが示唆された。この柔軟で高機能なシステムは、音楽教育だけでなくエンターテイメント分野への応用展開も期待される。

図 3. 作成した演奏機器

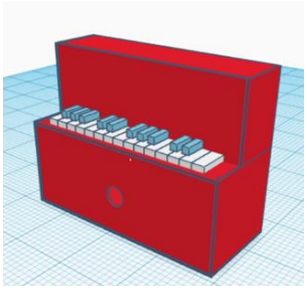


図 4. CAD 設計図

製作した LED 基板は鍵盤の下に LED を配置するために部品の位置を正確に調節し、設計した。LED 基板の回路図のうち 1 枚とアンプ付きスピーカー基板を図 5、図 6 に示す。

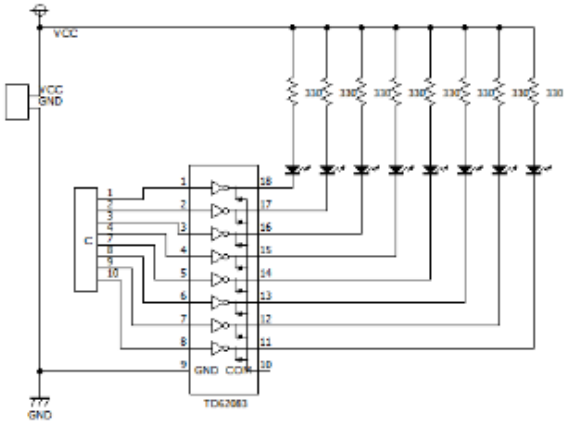


図 5. LED 基板回路図

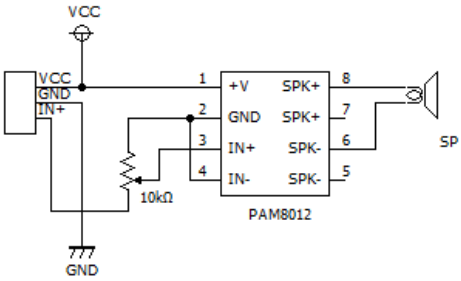


図 6. アンプ付きスピーカー基板回路図

IV. 制御装置

A. *開発環境*

今回の製作にあたって使用した開発環境を以下に示す。

- FPGA ボード(Field-Programmable Gate Array)
  ZyboZ7(Z7-020)
- ソフトウェア：
  vivado 2022.2
  Xilinx Vitis 2022.2
  MATLAB R2021b
  tinkerCAD
- MicroSD カード

B. *制御システム*

本稿で開発されたシステムは、SD カードから供給される画像データに基づいて音楽データを生成し、その再生を可能とするものである。このシステムは、AE を応用して、画像データから音の種類と長さを認識し、これらの情報を用いて LED 表示と音声出力を実現する。(図 7 参照)

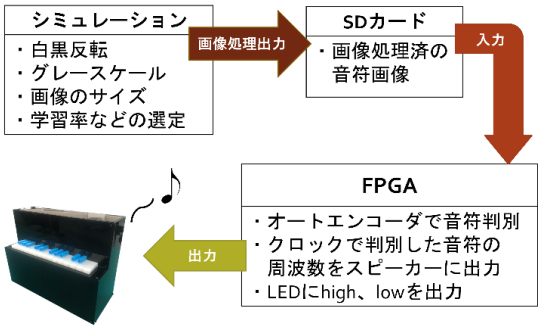


図 7. 処理の流れ

また、FPGA(Field-Programmable Gate Array)の構成を図 8 に示す。FPGA の構成は、左上が CPU、右上がオートエンコーダのハードウェア処理を行う IP で右下が今回 LED と音に変換する自作 IP である。その外にもいくつかブロックがあるがこれらはそれぞれの IP を繋ぐバスの役割になっている。AE の学習部分は、ソ

フトウェアで処理、スピーカー及び LED への出力部分はハードウェアで設計した。

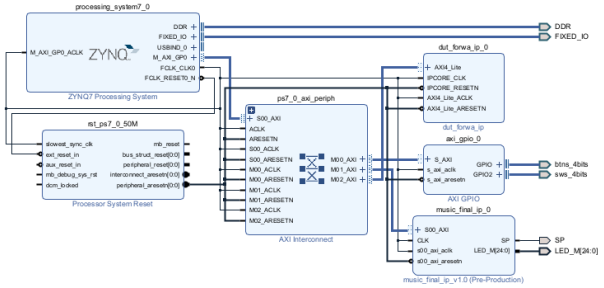


図 8. FPGA の構成図

図 9 に作成した自作 IP のブロック図を示す。自作 IP は、vivado の簡易 IP 作成機能「Create a new AXI4 peripheral」を選択し、AXI4 のインターフェース仕様はデフォルトの AXI4_Lite、Slave、32 ビット幅、レジスタ数 4 で作成した。またさらに内部に自作 Module music を追加した。
Module music では入力された CLK を分周し、スピーカーに出力する周波数を作成する。また、slv_reg0 と接続した IN からの入力データをもとに、対応した周波数、LED ポートを選択し出力する。

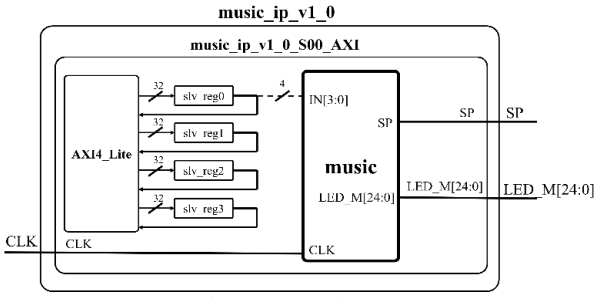


図 9. 自作 IP のブロック図

今回、設計したハードウェアサイズを表 1 に示す。

表 1. ハードウェアサイズ

Resource	Utilization	Available	Utilization
LUT	4099	53200	7.70
LUTRAM	62	17400	0.36
FF	4206	106400	3.95
BRAM	2.50	140	1.79
DSP	72	220	32.73
IO	12	125	9.60
BUFG	1	32	3.13

また、今回実験により導いた AE のモデルを図 10 に示す。入力層、出力層を 63×63(3969)で中間層は 128 構成に収束することができた。

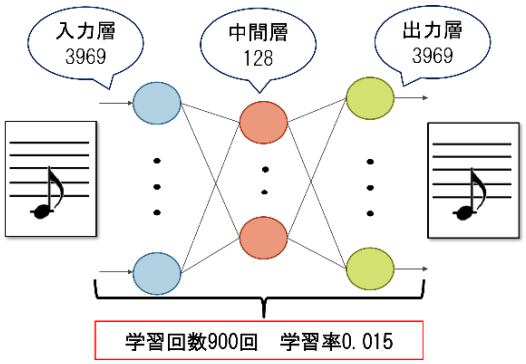


図 10. AE モデル

C. 周波数生成と音長制御

今回作成した自作 IP の Module music では先述の通り①音高に対応した周波数を作成②LED に出力、の二つの動作を行っている。今回コードを書くにあたって参考文献[3]のコードを参照した。まず参考文献では使用する周波数が 64kHz になっていたため Zybo の周波数である 125MHz から 64kHz へ分周を行った。

回路を組むために必要不可欠な基本データとなる音階と周波数の関係を表 2 に示す。音階の中央のラの周波数は 440Hz であり、この 1 オクターブ低いラで式は、音階の周波数＝220×2^(n/12)で求めることができる。また、n は 1 から 12 の整数を入れる。音階が 1 オクターブ高い時にはこの分周は半分の値になり、1 オクターブ低い時にはこの分周は二倍になる。
また、本研究では当初自動演奏の機能の実装を考えていたことから音の長さを表す周波数も作成している。本稿では 16 分音符を一つ分として考え、0.25 秒とした。表 2 のように音高を決める周波数を、音長を決める周波数に合わせ、呼び出し回数を増やすことで音の長さを調整することが可能になる。例えば 8 分音符だと 16 分音符 2 個分であることから二回、4 分音符だと 4 個分であることから四回、2 分音符だと 8 個分のため 8 回呼び出している。Module music の構成図を図 11 に示す。

表 2. 音高と対応する周波数［2］

音階の番号	音階	平均律音階の周波数	分周比 (CLK=64000)	分周比の半分	オクターブ高いときの分周
0	ラ	220	290.91	145	72
1	ラ#	233.08	274.58	137	68
2	シ	246.94	259.17	129	65 (si)
3	ド	261.63	244.62	122(do)	61 (octdo)
4	ド#	277.18	230.9	115	57
5	レ	293.66	217.94	109 (re)	54 (octre)
6	レ#	311.13	205.7	102	51
7	ミ	329.63	194.16	97 (mi)	48 (octmi)
8	ファ	349.23	183.26	92 (fa)	46 (octfa)
9	ファ#	369.99	172.98	86	43
10	ソ	392	163.27	82 (so)	41 (octso)
11	ソ#	415.3	154.11	77	38
12	ラ	440	145.45	73 (ra)	36

まず音階テーブルに音楽のメロディになるデータを入れておく。これは楽譜の役割をしている。先述の通り音の長さの最小単位である 0.25s ごとにこのテーブルからデータを読み出し，音階生成カウンタ（musicosc）に音階生成データとして与える。カウント数を変化させることで音階の周波数を作り出すという仕組みになっている。
inclk は，64kHz のクロックを 16000 分周して，0.25s（4Hz）のタイミングを作っている。cts025 は，音階テーブルのアドレスを作るカウンタで，4Hz で動作する。cts025 の出力信号は，テーブル melody のアドレスになり，0.25s ごとにインクリメントする仕様になっている。音階テーブル melody のアドレスは，0.25s ごとに変化するため，出力の outmusic も同様のタイミングで変化する

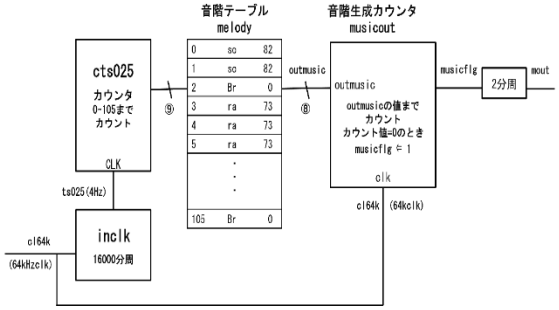


図 11.Module music の構成図

音階生成カウンタの musicosc は，outmusic の値によって，カウントする値が可変可能である。64kHz のクロックの分周比を outmusic の値によって変えることで、カウンタ出力の周波数に変化をつけて，音程の周波数を作り出している。
音の出力の原理を図 12 に示す。例えばテーブルの値がラの場合、outmusic=73 になり、カウンタ musicosc は 73 進のカウンタになる。musicosc が 0 になった際に musicflg が 1 になり、つまりは 73 個に 1 回 High になる。これを 2 分周するため mout は 146 個分の周期になり、ラの周波数のちょうどデューティ比 50%の音階の発振出力 mout になる。
mout はスピーカーに接続されているため、その振動が音になり聞こえる。またテーブルの値が Br の時は outmusic が 0 になるため musicflg は常に High になり、発振しない。これは所謂休符と同じ役割になっている。

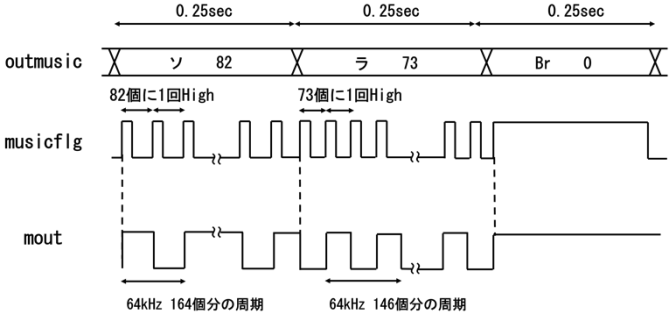


図 12. 音の出力原理

D. 画像生成

本研究の使用する画像では、AE に前処理された画像データを供給するために、以下の画像処理技術が適用される。

① 白黒反転
デジタル画像データにおける色の表現を、黒を 0 の数値で、白を 1 で表されるように反転させることで、黒色が占める領域を強調し、後続の演算手順を容易にするために使用される。

② グレースケール処理
入力される楽譜画像は、本質的にモノクロだが、多くの場合カラー画像フォーマットで読み込まれる。これらにグレースケール処理を施すことで、画像データをモノクロに変換し、AE モデルが扱いやすい形式に統一する。

③ サイズ調整
画像データのサイズを縮小することによって、データの処理の負荷を減らし、効率的な計算を実現するための措置を行う。

V. 実験

A. 実験内容

本実験では、MATLABを用いて教師データを元に入力画像が正確に再現できているか検証する。また、正確に再現できた場合の学習率、学習回数の値を用いてFPGAボードで再度AEを行う。事前処理として、教師データと入力データに白黒反転、サイズの調整、グレースケール変換を行った。使用した教師データを図13に示す。入力データは教師データと同じデータを使用した。

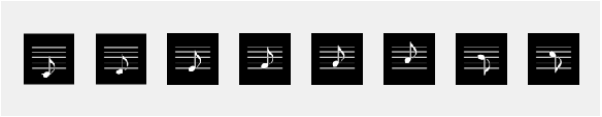


図13. 教師データの概略図

B. 実験結果

実験によって得られた結果を、教師データと比較して示した図14では、学習率は0.1、学習回数は500で演算を行っており、生成された画像の不安定さが確認される。また学習する過程での正確さを表すグラフを図15に示す。このグラフは、教師データと出力画像の誤差を示しており、縦軸のエラー数が0に近づくほど正確さが増す。図15では最終的なエラー値が約25.8になった。

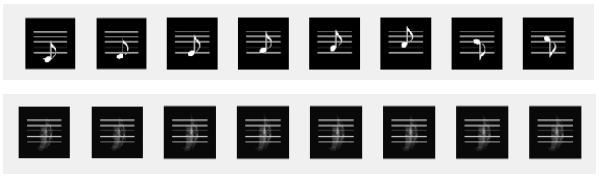


図14. 実験結果1

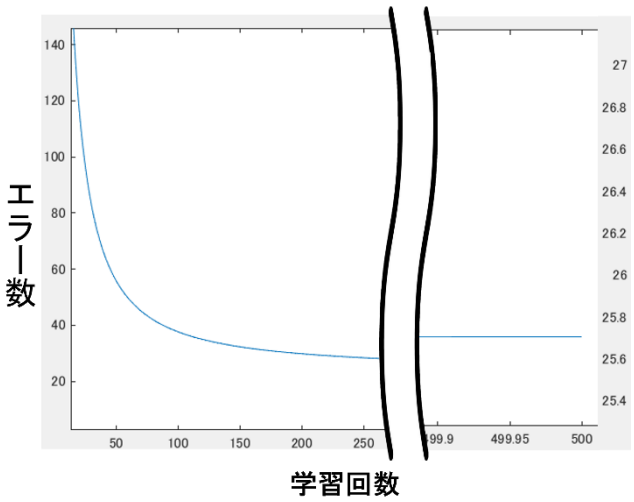


図15. 出力画像と教師データとの誤差1

この問題は、教師データの数に対して学習率が大きすぎることが一因とされる。そこで、学習率を下げる調整をして再度実験を行い、改善された結果を図16で示す。

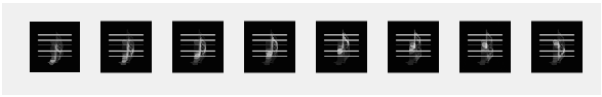


図16. 実験結果2

図16は、学習率は0.02、学習回数は500で演算を行った。確認すると、かなり改善は見られたが、いくつかの不安定な要素まだ見受けられる。特に、最終的な2つの画像が同じ音を示していることや、音符が重複し余分な画像が残っている。この要因として、画像の特徴を捉えるための十分な時間が確保できていないことが一因として考えられる。この問題に対処するため、学習回数の調整を再度実施した。その後、誤差を表すグラフを見ながら都度調節し、最終的に学習率は0.015、学習回数900で演算を行うことで、教師データに更に近似した。結果を図17で示す。また、図18では最終的なエラー値が約6になっており、初期の実験より誤差が減少していることが確認できる。

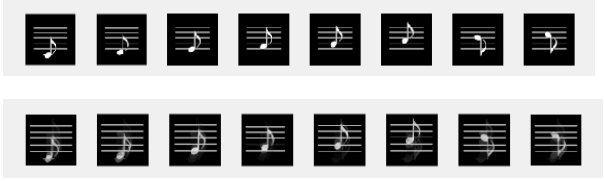


図17. 実験結果3

さらに、最終的に得た学習率、学習回数を用いてCPUで演算を行い、結果をMATLABに返すことで演算結果を画像出力した結果を図19に示す。

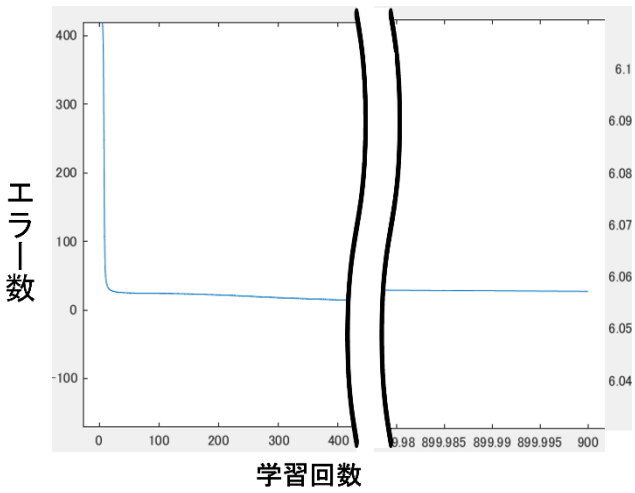


図18. 出力画像と教師データとの誤差2

確認すると、CPU内でも教師データと近似した画像が出力できていることが分かる。

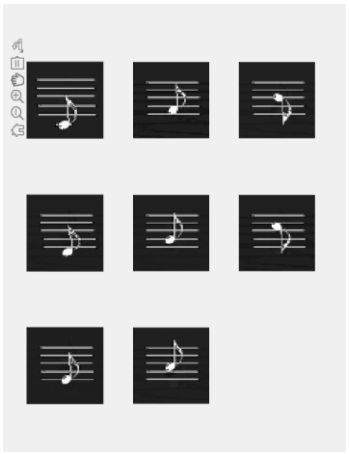


図19. CPU内の演算結果

これらの結果から、入力データと教師データが最も類似している状態のデータを選出し、それをFPGAボードに送信することで、ボードを通じてスピーカーやLED等の出力デバイスに目的のデータを正確に再現できることを確認した。

VI. オートエンコーダのフォワード演算

オートエンコーダの設計にあたり、Level 1の課題である3×3マトリックスのパターン認識を参考にした。具体的には、図8に示すように、入力層と出力層をそれぞれ9ノードとし、中間層を2ノードとするシンプルな構成を採用した。

ただし、入力画像が63×63ピクセルであることから、入力層と出力層のノード数を3,969とし、この数値が9で割り切れるよう調整した。これにより、中間層の2ノードを用いた64回の演算で、楽譜画像から音楽情報へのマッピングを実現できる。

この比較的シンプルで直感的なネットワーク構成は、実装と処理が容易である一方、高度な音楽情報の抽出を可能にする柔軟性を備えている。

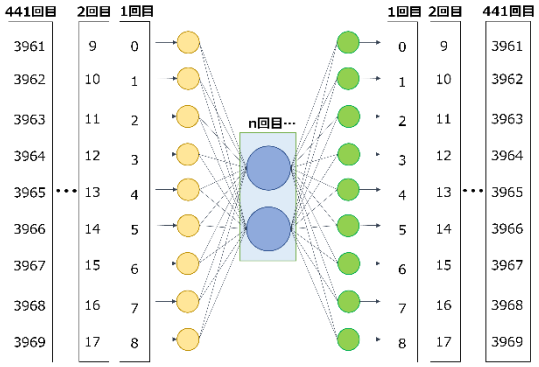


図20. 小規模FPGAでのフォワード処理

VII. アピールポイント

本稿の試みは、画像と音の間の新しい変換メカニズムを見出し、AEの応用範囲を広げると共に、音楽分野におけるAI技術の可能性を広げるという革新的試みである。将来的には、この技術の発展により完全自動の演奏機器が実現されることとなる。これは、芸術と技術が融合する未来への一歩となると同時に、エンジニアリングとデジタルクリエイティビティの邂逅点という学問領域の開拓にもなり得る。

VIII. おわりに

本研究では、音符画像を読み取り、音を出力できる機器をテーマに進めた。MATLABでシミュレーションを行いながら画像処理とオートエンコーダの各種パラメータを取得するとともにFPGAのモジュール作成も行った。また、入力データが教師データに近い画像データを生成できることを確認した。その生成されたデータで、入力画像データに応じた音をスピーカーとLEDに出力することが達成された。

ただし、本研究では一度に一つの音しか読み込んで出力できない制限が存在する。今後の課題として、楽譜全体を入力データとして扱い、並んでいる複数の音符を一つずつ認識できるような処理を実現することが望まれる。

また、処理の高速化を図るために、白黒反転や画像サイズの変更などを実施したが、教師データが多いことによりまだ処理が遅いと感じられる。今後、音の種類を増やしていくと、教師データも増加し、処理の負荷が増すことは自明である。そのため、処理の高速化に向けた新たな手法やアルゴリズムの検討が課題となる。

参考文献

[1] LSI Design Contest. (2023). The 26th LSI Design Contest in Okinawa. Retrieved January 10, 2023, from http://www.lsi-contest.com/

[2] 香西久美子. 子どもの読譜力の発達に関する研究, http://www.art.hyogo-u.ac.jp/hrsuzuki/students/peco99.pdf

[3] 佐々木淑恵. (2003). 電子オルゴールの製作. 『2003年1月号付属CPLD基板』関連制作レポート. Retrieved from https://www.cqpub.co.jp/dwm/turnout/200301/03/

Design and implementation of an assisted wayfinding support system using Braille blocks

－Improved mobility assistance and spatial awareness through integration of smart technologies－

1st Keita Yamada
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
1665-1 Shii Kokuraminami-ku
Kitakyuushuu-shi Fukuoka, Japan
2217129@kyushu-pc.ac.jp

2nd Mako Fukuda
Kyusyu Politechnic College
Department of Electronics and Information Tecnology
1665-1 Shii Kokuraminami-ku
Kitakyuushuu-shi Fukuoka, Japa
2217118@kyushu-pc.ac.jp

I. はじめに

本研究では、視覚障碍者の移動支援を目的とした技術開発に取り組んだ。視覚障碍者は移動時に不安を抱えることが多く、この課題の解決が求められている。

そこで点字ブロックを利用した自律移動ロボットの開発を提案する。本システムは専用の点字ブロック上を走行する小型ロボットで、カメラによって路面の状態を検知しながら自動運転を実現する。障害物検知時にはブザーで使用者に警告を発するなど、安全性にも配慮した機能を搭載している。

このように、情報工学と福祉工学の融合によって、視覚障害者の自立した移動の支援を目指す本研究は、高い社会的意義を有している。

A. ロボット概要

図 1 に示したロボットは、現時点では概念実証を目的としたプロトタイプである。本ロボットは MATLAB 上で模擬した点字ブロックのみを走行可能であり、実環境下での利用を想定したものではない。

次段階として、実際の点字ブロックでの走行を可能とする実機ロボットの開発が必要不可欠である。実点字面上での安定した走行性能を確保することで、はじめて視覚障碍者の移動を支援できる実用的なシステムが構築できるからである。

このように段階的に研究を発展させることで、福祉工学分野における本研究の意義はさらに高まると考えられる。

図 1. ロボットの外観

II. システム概要

本ロボットの点字ブロック認識および走行制御のために、オートエンコーダを応用したアルゴリズムを設計した。 具体的には、USB カメラから取得した路面画像を入力データとしてオートエンコーダに供給する。オートエンコーダは、事前に学習させた点字パターンの教師データと入力画像を比較し、点字の種類を識別する。

次いで、オートエンコーダの出力値に基づいて、ロボットの走行方向や速度といった制御信号を生成する。 この一連の処理によって、ロボットは点字ブロックを認識しながら自律走行を実現できる。

このように、情報工学の最新技術を取り入れることで、視覚障害者の移動支援システムの高度化・実用化が期待できる。

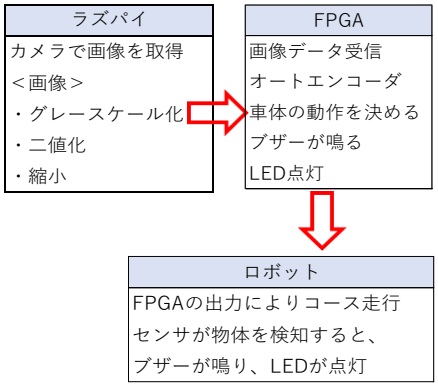


図 2. システム構成図

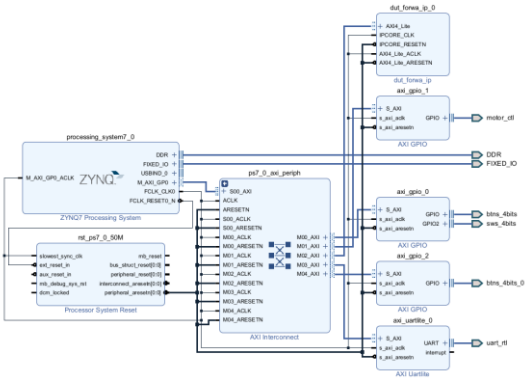
本システムの構成を図 2 に示す。主要な構成要素は以下の 3 つである。

1) 画像入力部:USB カメラにより路面の画像データを取得する。

2) オートエンコーダによる認識部:入力画像から点字パターンの特徴抽出と認識を行う。

3) ロボット制御部:オートエンコーダの出力に基づき、ロボットの走行制御を行う。

これら 3 つのコンポーネントが連携することで、カメラ入力から走行制御までの一連の情報処理が実現され、点字ブロック上を自律走行可能なシステムが構築できる。本システム構成は拡張性と汎用性にも優れており、今後の発展が期待できる。



①オートエンコーダの処理を行うブロック
②モータを制御するためのブロック
③FPGA 内部のデバック用の SW,BTN のブロック
④FPGA 内部のデバック用の LED のブロック
⑤UART 通信を行うためのブロック

図 3. ロボットのシステム構成

A. 画像入力部

USB カメラから取得した画像に対して、Raspberry Pi 上で前処理として 2 値化と画像縮小を施している。これにより、後段のオートエンコーダにかかる処理負荷を軽減し、特徴抽出の効率化を図っている。

具体的には、2 値化処理によって点字面のみを切り出し、画像縮小によって処理する画素数を低減する。これらの前処理により、オートエンコーダの学習スピードと実行速度の向上が期待できる。

このように適切な前処理を行うことで、制約のある組み込みシステム上でも、高度な画像認識を実現できる。前処理段階での工夫はリソース制限環境での効率的な認識アルゴリズム構築に重要である。

(a) 取得画像

(b) 二値化画像

図 4. 入力画像

B. 処理制御部

Raspberry Pi による前処理後、画像データは UART 通信経由で FPGA ボードに送信される。FPGA 上のオートエンコーダがこの画像を入力として点字パターンの認識を実行する。

オートエンコーダは事前の学習に基づき、入力画像から点字の種類を識別し、直進や停止などの制御命令を生成する。これらは最終的にモータ制御信号として出力される。

なお図 5、6 に示したオートエンコーダの構造と学習データは、アルゴリズムの検証のために MATLAB 上で作成したものである。本番環境への実装時には、FPGA 上で同様の学習と認識処理を実現する必要がある。

このように適切に処理を分割することで、組み込みデバイスの特性を生かした効率的な実行が可能となる。

図 5. 入力データ

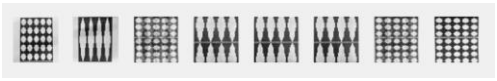
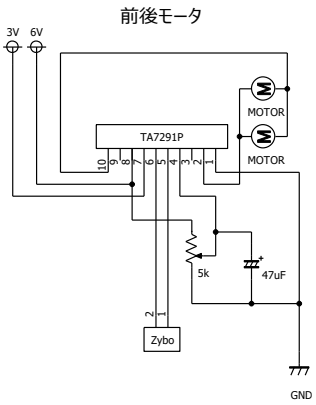
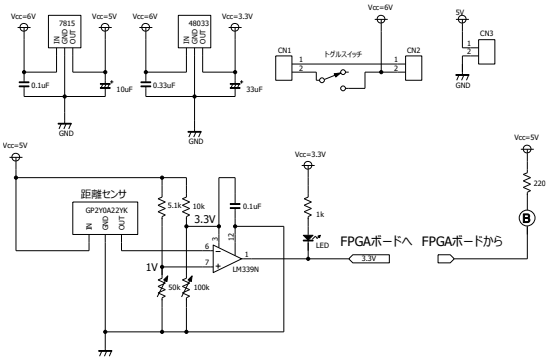


図 6. 出力データ

安全性を考慮し、センサから障害物を検知した場合の緊急停止機能を実装した。



(a) モータ制御回路



(b) センサー・ブザー制御回路

図 7. 制御回路

具体的には、センサで障害物が認められた時点で、ロボットの走行中であっても直ちにモータを停止させる。加えてブザーによる音声アラートを発し、ユーザに注意喚起を行う。これにより接触事故を未然に防ぐことができる。

モータ制御回路とセンサー・ブザー制御回路は図 7 のように実装した。このような多層的な安全機構の導入は、実用化を考えた場合に必須の要素である。ハードウェアとソフトウェアの両面から安全性確保を図ることで、信頼性の高いシステムを構築できる。

C. ハードウェア化

オートエンコーダのハードウェア実装にあたり、入力層 9 ノード、中間層 2 ノード、出力層 9 ノードの基本構造を FPGA 上に構築した。これに 54×54 画素の 2 値化入力画像を分割して供給することで、MATLAB 上で導出した 100 層中間層モデルと同等の認識処理を実現できる。

このアプローチにより、小規模 FPGA を用いた場合でも、一部の層をハードウェアで実装することが可能となり、これにより、ソフトウェアのみの場合と比較して処理速度の大幅な向上を図ることができる。

このように、利用可能なハードウェアリソースに応じた最適化設計は、組み込みシステム設計において極めて重要である。ハードウェアとソフトウェアの協調的な実装が、効率的かつ高性能なシステム実現の鍵となる。

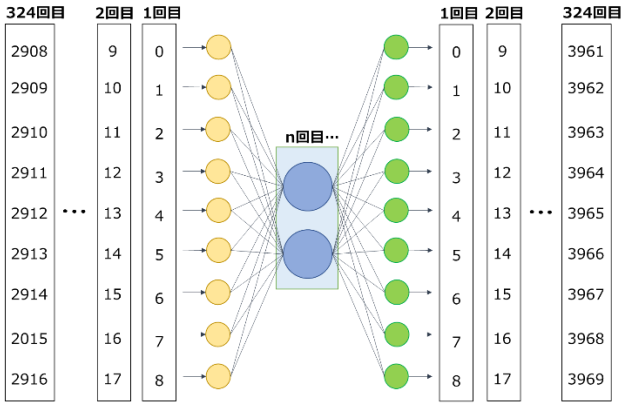


図 8. ハードウェア化

III. 実験

A. 実験内容

本研究では、Raspberry Pi による画像前処理と、FPGA 上のオートエンコーダによる点字認識・判別を組み合わせることで、ロボットの自律走行制御を実現した。

オートエンコーダのアルゴリズム構築・検証は、MATLAB 上でのシミュレーションにより行った。画像の前処理としてサイズ調整や 2 値化を施し、点字面のみを抽出した。次いで、図 9(a)(b)に示す直進と停止を指示する 2 パターンを教師データとして学習を行った。これらの教師データは、実画像に合わせブロックの形状・サイズを標準化して作成した。

このように適切な前処理と教師データの構築により、実環境での点字認識・走行制御を高い精度で実現できた。ソフトウェアとハードウェアの協調的な活用が、実用的なシステム開発の肝となる。

(a)直線　　(b)停止

図 9. 教師データ画像

オートエンコーダの学習と評価は以下の手順で実施した。

① 教師データ自体を入力した場合に、元のパターンが正確に再現・出力されるかを確認。

② 教師データにない入力パターンを順次与え、最も近い教師データに分類・マッピングできるかを評価。

この時、入力パターン数を徐々に増やしながら実験を重ねた。

以上のプロセスを通じ、オートエンコーダの汎化性能を定量的に評価し、パラメータチューニングを行った。教師データの再現性と汎化性能の双方を担保することで、実環境下でのロバストな認識が可能となる。

B. 実験結果

図 10 と図 11 は、教師データとオートエンコーダの出力結果を比較したものである。 最初の出力結果では、直進と停止の教師データが混合した中間的なパターンが生成されており、目的の出力を再現できていなかった。 これは、設定した中間層の次元数が過大であることが原因として考えられる。

中間層が過大な次元を持つと、出力結果が教師データの中間特徴に収束してしまいがちである。 そこで次元数を適切に削減することで、教師データを正確に再現できることが期待される。

このように教師データとの比較検証を繰り返しながらモデルを改善していくプロセスが、オートエンコーダに限らず機械学習一般において非常に重要である。

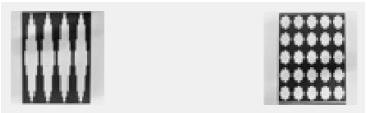
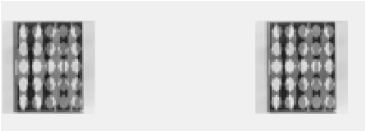
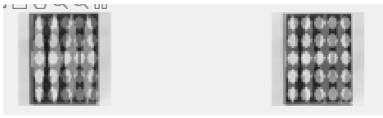


図 10. 教師データ



(a)1 回目の結果



(b)2 回目の結果

図 11. 学習後データ

次に、中間層のユニット数を減らして再学習を行ったが、出力結果に大きな改善は見られなかった。そこで学習の収束状況をモニタリングするため、学習率の推移グラフを確認したところ、誤差が収束せず概ね横ばいで推移していることが判明した。このことから、現在の学習率が小さく、誤差が十分最小化されないまま学習が終了している、つまり学習が不足していることが考えられる。図 12 に図 11 で示したデータの学習推移を示す。

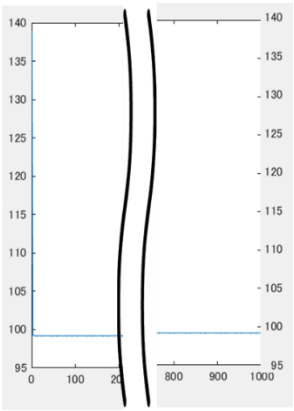


図 12.　学習率のグラフ

そこで学習率を高めた値に更新して再学習を試みたところ、図 13 に示すように 2 種類の教師データが明瞭に識別された出力が得られるようになった。これにより適切な学習率の設定によって目的の出力が生成可能であることを確認した。

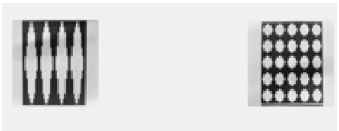


図 13.　2 パターンの学習結果

学習率を増加させた結果、図 13 のように目的の出力が得られるようになった。さらに学習の収束状況を確認するため、学習率の推移グラフを図 14 に示す。1 回目の学習時と比較して、今回は誤差が収束し 0 に近づいていることが確認できる。

以上から、適切なハイパーパラメータ設定によって、オートエンコーダが点字パターンを識別し、目的の出力を生成できることが実証された。

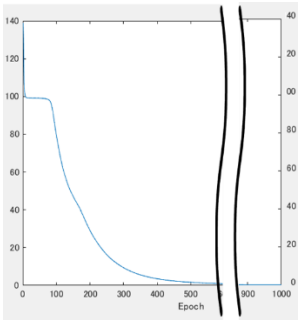


図 14. 学習率を増加のグラフ

教師データに対する出力が適切に得られることを確認した後、入力パターン数を増加させることでオートエンコーダの汎化性能を評価した。図 15 に示すように、教師データとは形状の異なる複数の点字画像を新たに入力データとして加えた。

その結果、図 16 に示す出力結果のように、一部の点字形状では目的の出力が生成できず、オートエンコーダの汎化性能に限界があることがわかった。このことから、教師データの種類や量をさらに増加させることで、未知の入力パターンに対する頑健性・柔軟性を高められると考えられる。

図 15. 入力データ

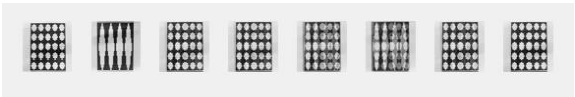


図 16. 出力データ

出力結果を詳細に解析したところ、教師データの中には正確に再現されているものと、誤った出力が生成されているものが混在していた。

この原因は、教師データ自体の量が不足していることにあると考えられる。教師データが少ない状態では、モデルがパターンの多様性を十分に学習できていない可能性が高い。

そこで、教師データの種類と量を追加して再学習を行うことで、モデルがパターンの幅広い特徴を捉えられるようになり、安定した認識性能が得られることが期待される。十分な教師データの確保とそれに基づく学習は、汎化性能向上の鍵となると考えた。

図 17. 増やした教師データ

図 18. 学習後データ

出力結果から、停止を指示する丸型の点字パターンが不鮮明に再現されていることが確認できた。この原因として、教師データ量の増加に伴いオートエンコーダが過学習状態となり、汎化性能が低下した可能性が考えられる。

そこで対策として、学習率を低下させることで再学習を試みた。学習率を下げることにより過剰な学習を抑制し、未知の入力に対する頑健性を取り戻すことを狙った。

図 19.　学習後データ

学習率を適切な値に下げることで、図 19 に示すように目的の出力を再現できるようになった。

	中間層	学習率
1 回目	180	0.01
2 回目	100	0.01
3 回目	100	0.02
4 回目	100	0.02
5 回目	100	0.01

表 1. 中間層・学習率の変化

適切な実験結果が得られたため、これを利用しリアルタイム画像をオートエンコーダし、ロボットを走行させることを試みた。

しかし、安定して上手く出力させることができなかった。学習率などの条件を変化させてみたが、結果は変わらなかった。

原因は、リアルタイムで画像を取得した際、移動することにより明るさが変化し、画像の二値化が上手くいっておらず、入力画像を判別できていないことにあった。上手くいく場合もあったが、少し位置をずらすだけで、明るさが変わり本来白ではない部分が白になるなど上手く二値化できない場合が多かった。

そこで、二値化を正確に行うために、白黒から黄黒で表現の画像へと変更し、再びロボットの走行を試みた。

その結果、図 4 のように二値化は上手く処理させることに成功した。また、USB カメラで画像を取得するプログラムとシリアル通信を行うプログラムを並列に動作させることで、画像取得をスムーズに行えるようにした。

しかし、ロボットは走行しなかった。オートエンコーダの出力結果を見てみると、入力データが進行であるのに対し出力データは停止と判断され、識別が上手くいっていなかった。

原因として考えたことは、教師データを白黒の画像のままで行っていることだった。黄黒にしたことで、白黒のままでは判別ができないため、教師データを変更することにした。

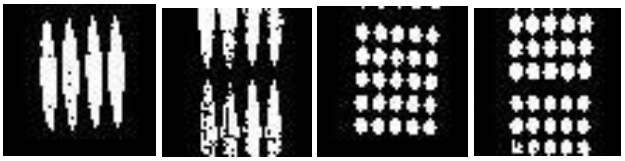


図 20. 変更後の教師データ

教師データを図 20 に示す。これは UART 通信後のデータである。この教師データを MATLAB で先ほどと同じ手順でシミュレーションし、ロボット走行を試みた。

その結果、上手くオートエンコーダで画像判別ができ、ロボットを走行させることができた。

最終的な中間層・学習率の値を表 2 に示す。

中間層	学習率
150	0.0015

表 2. 最終的な中間層・学習率の値

IV. おわりに

本研究の動機づけは、視覚障害者の移動支援とアクセシビリティ向上を実現する技術の開発である。今回は実環境下ではなく模擬点字ブロック上を対象としたが、得られた自律走行制御の基本原理は実用化につながる。

点字ブロックの形状認識および走行ルート決定のアルゴリズムは汎用性が高く、テクスチャー認識や迷路解決といった他タスクへの応用が期待できる。また、明暗変化への追従性を向上させることで、野外での利用も可能となる。

しかしながら、実用化に向けて以下の点を改善・拡張する必要がある。

- 停止後の再走行開始にはユーザの音声指示が必要で自動制御化されていない。複数カメラやセンサーにより前方が開通しているかを判断し、自律的に走行を再開できるようにする。

- 照明環境等により取得画像にバラつきが生じ、認識性能が不安定となる。データ拡張や CNN 活用等の手法でロバスト性を向上させる。

実際の点字ブロックや屋外環境での検証が不十分である。引き続き様々な環境下での評価が必要となる。

参考文献

[1] LSI Design Contest. (2023). The 26th LSI Design Contest in Okinawa. Retrieved January 10, 2023, from http://www.lsi-contest.com/

Auto Encoder を用いた印鑑の判別

Signature Stamp Identification using Auto Encoder

Team: Black Russian

Yume Nagata

Department of Artificial Intelligence

Kyushu Institute of Technology

nagata.yume269@mail.kyutech.jp

1. Introduction

日本の契約においては、仕事上のみならず、銀行、郵便、自動車、住宅などの様々な場面で印鑑（signature stamp）が重要である。このプロジェクトでは、取り込んだ印鑑の画像データと登録データで、電子的に AI 判定する LSI デザインを開発する。

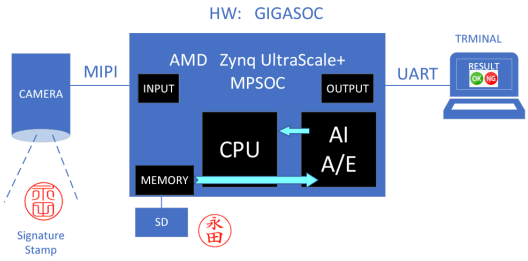


Figure 1　Image of System

2. Autoencoder

本システムでは、機械学習の手法の一つである Autoencoder(AE)を適用した。

今回、計算値が大きくなりすぎることを防ぐために、ReLU 関数ではなく Satlin 関数(1)を用いた。

$$f(x)=\begin{cases}0, & x<0\\ x, & 0\le x\le 1\\ 1, & x>1\end{cases} \qquad (1)$$

また、デコーダーの活性化関数には Sigmoid 関数(2)を、誤差関数には平均二乗誤差(3)を用いた。

$$f(x)=\frac{1}{1+e^{-x}} \qquad (2)$$

$$f(x)=\frac{1}{n}\sum_{i=0}^{n}(y_i-\hat{y}_i)^2 \qquad (3)$$

3. Development environment

A. PC environment for compilation

i. Ubuntu 18.04.6 LTS

ii. Vivado 2022.2 / Xilinx Vitis 2022.2

B. PC environment for IP creation

i. Windows 11

ii. MATLAB 2021b

C. FPGA

i. DIGILENT 社製 Zybo Z7-10 (ZYNQ-7010 Development Board)

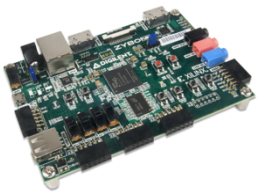


Figure 2　Zybo Z7-10

ii. Gigafirm 社製 GIGASOC

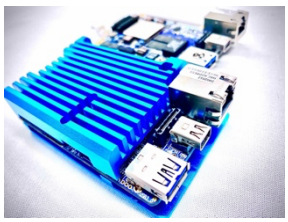


Figure 3　GIGASOC

4. System overview

A. How the system works

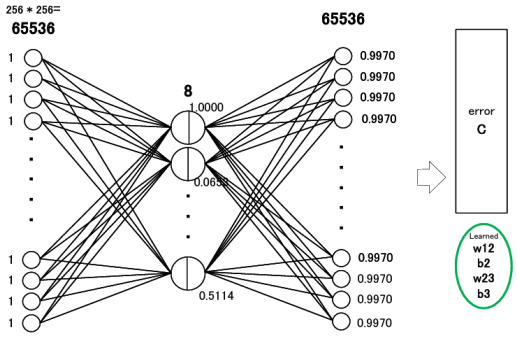


Figure 5　Training Image

初めに元となる印鑑の画像を入力し、AE を学習させておく(Figure 5)。この際に、学習回数分の誤差と、学習済みの重みとバイアスを取得する。

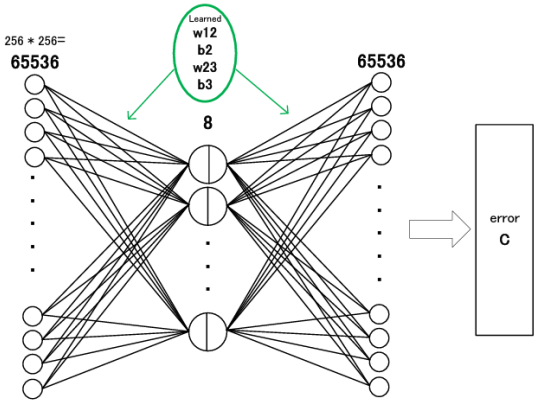


Figure 7　Target Image

その後、ターゲットとなる画像を同じように AE に入力する(Figure 7)。この際に、先ほど取得した学習済みの重みとバイアスを用いることとする。この際にも誤差を取得する。

教師データの学習から得られた誤差の下位 5%を閾値に設定する。ターゲットの画像を入力した時の誤差の値は、同一の印鑑の画像であれば閾値を超えない。しかし、異なる印鑑の画像を入れると誤差がこの閾値を超えてしまうため、判定が可能となる(Figure 4)。

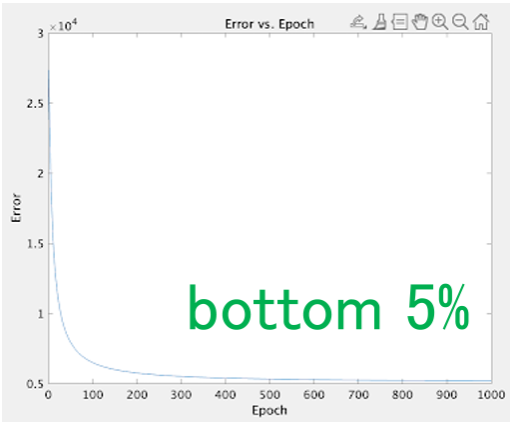


Figure 4　Error vs. Epoch

B. Inside the FPGA

今回はターゲットの FPGA のリソースを鑑み、Encoder 部分の行列計算を行う IP を作成し、これを FPGA に実装した。

i. Zybo Z7-10

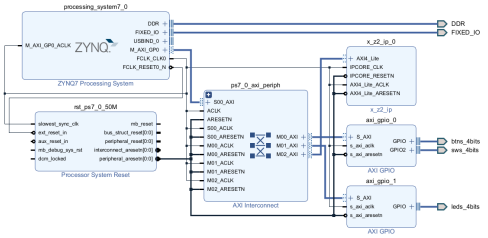


Figure 6　Zybo Z7-10 Block Design

まず、開発の容易な Zybo Z7-10 を用いてシステムの作成を行った(Figure 6)。この FPGA の場合、DSP 数が最大 80 個と非常に少ないため、4 入力の行列計算を行う IP を作成し、これを用いて FPGA 内部の作成を行った。

ii. GIGASOC

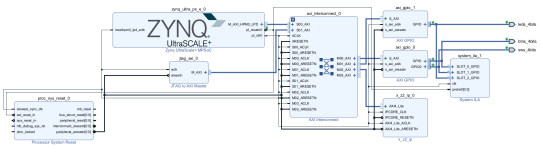


Figure 8　GIGASOC Block Design

Zybo ではシステムの実行に時間が

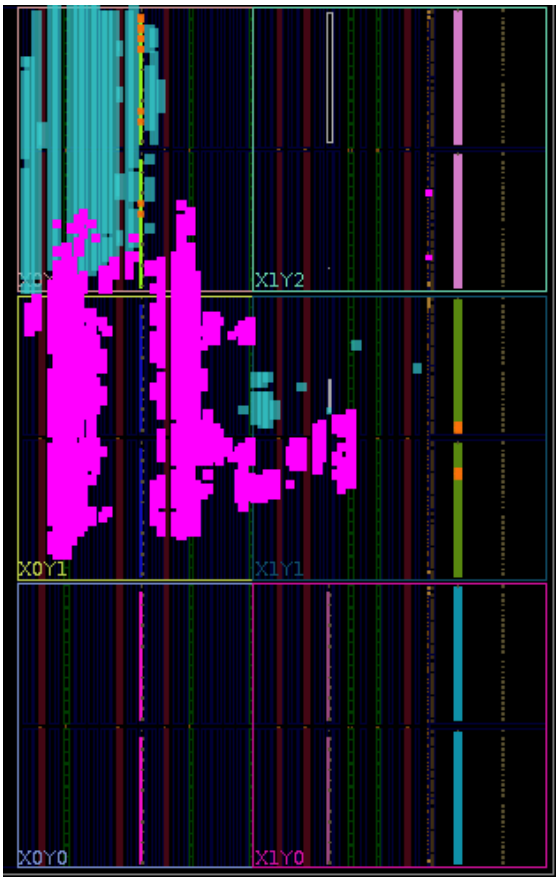
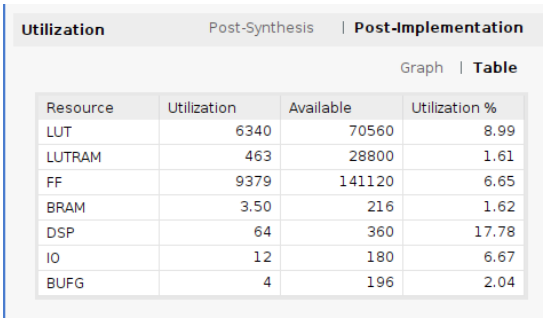


Figure 9　Where to use additional modules



Utilization　Post-Synthesis | Post-Implementation

Graph | Table

Resource	Utilization	Available	Utilization %
LUT	6340	70560	8.99
LUTRAM	463	28800	1.61
FF	9379	141120	6.65
BRAM	3.50	216	1.62
DSP	64	360	17.78
IO	12	180	6.67
BUFG	4	196	2.04

Figure 11　Resource utilization

かかるため、Gigafirm 社製の GIGASOC を用いてシステムの作成を行うこととした(Figure 8)。Figure 9 は GIGASOC の内部使用状況を示したものである。このうち、ピンクで示された箇所が追加した IP の占めるところである。また、Figure 11 は GIGASOC のリソースの使用状況を示したものである。GIGASOC は DSP 数が 360 個と Zybo と比較して非常に大きい。また MPSoC(Multi-Processor System on Chip)であるために処理能力の向上が図られる。

C. Image of operation

今回は、Figure 10 のように、Tera Term に結果を出力した。

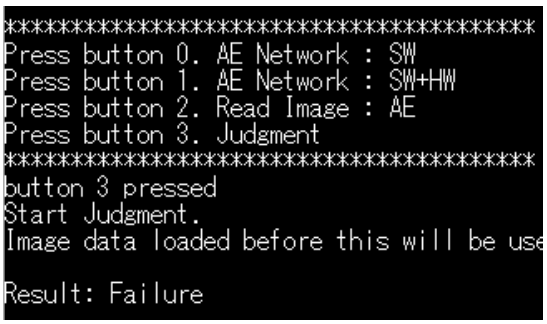


Figure 10　Result Image

システムを 4 段階に分けている。

i.　教師データの読み込み

button 2 を押すことで教師データのイメージを読み込む。

ii.　AE

button 0 もしくは 1 を押すことで AE の学習を行う。

iii.　ターゲットデータの読み込み

button 2 を押すことでターゲットデータのイメージを読み込む。

iv.　判定

button 3 を押すことで判定する。

4. Future challenges

今回のシステムには、いくつかの改良可能な点が存在する。

A.　IP scale up

GIGASOC 等の容量の大きな FPGA を用いることによって、並列処理が可能な範囲が増加し、AE 学習の高速化が図れ

る。

B. How to obtain image data

今回は SD カードに予め印鑑の画像データを入れておき、その中からスイッチ操作によって選択した画像で判定を行う方式にしている。今後は、取得方法を MIPI Camera に変更し、より実用的なシステムに変更することを考えている。

C. How to obtain judgment results

今回は Tera Term での出力であったが、今後はシステムを SD Boot することを考えている。そのため、FPGA にディスプレイを接続し、PC から切り離して FPGA 本体のみで起動できるようにしたいと考えている。この際は、結果を画像出力することが望ましいと考える。

D. Judgment other than signature

今回の判別システムは、うまく改良すると印鑑以外の画像データの判別を行うことができる。FPGA を用いた認証システムとしての使用が期待される。

5. Impressions

今回は、まず大規模化することを考えて $256 \times 256 = 65536$の入力が可能な IP を作成したが、これを入れる FPGA が存在しない(CARRY 4 が 350000 以上必要)ことに気づくまでに時間がかかってしまった。このため、時間と人的資源の限られた中でのシステム開発であったため、何をどこまで頑張るかというところが難しかった。

DSP 数などのリソースに関する問題が最大の悩みどころであったが、Gigafirm 株式会社様のご厚意により GIGASOC を使用することができたため、この問題に糸筋が見えた。

私は、AI に関する学科にいながら機械学習等の AI の実装を行ったことがなかったため、今回の AE に関する学習を含め、参加は非常に良い経験になったと思う。今後も、このシステムの改良や、他の機械学習についても学習を進めていければと思う。

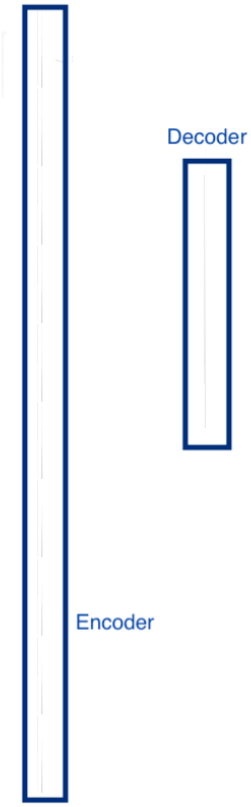


Figure 12　IP that can input 65536

HANOI UNIVERSITY OF SCIENCE AND TECHNOLOGY
SCHOOL OF ELECTRICAL AND ELECTRONIC ENGINEERING

LSI CONTEST

Convolutional AutoEncoder

EDABK

Tuan Khoi Nghiem
khoi.nt192941@sis.hust.edu.vn

Xuan Son Dao
son.dx203763@sis.hust.edu.vn

Dai Duong Truong
duong.td203687@sis.hust.edu.vn

Hoang Kien Trann
kien.th193302@sis.hust.edu.vn

Hanoi, March 5, 2024

INTRODUCTION PAGE

In today's digital age, image processing has become an important field with widespread applications in various sectors such as healthcare, automation, and transportation. Convolutional Neural Networks (CNNs) have demonstrated their power in solving image processing tasks. In this context, the autoencoder, an application of CNNs, has been chosen for practical use. The autoencoder not only has the ability to compress and reconstruct data but also serves as a powerful tool for automatic representation learning and feature extraction from image data.

This report focuses on optimizing hardware for the autoencoder image decoder, particularly in the context of image processing. We will delve into hardware optimization methods to enhance the performance of the autoencoder image decoder, including hardware architecture, network architecture optimization, and the utilization of new sensor technologies. We will also discuss the challenges in hardware optimization for the autoencoder image decoder and the future opportunities it presents.

I would like to express my gratitude to Mr. Nguyen Duc Minh and Ms. Hoang Phuong Chi for their enthusiastic assistance and support throughout the process of developing the idea and implementing this project. I also sincerely thank my colleagues in the EDABK laboratory at the University of Engineering and Technology, Hanoi University of Science and Technology, for their help during the project implementation.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

Due to recent advances in digital technologies, and availability of credible data, an area of artificial intelligence, deep learning, has emerged, and has demonstrated its ability and effectiveness in solving complex learning problems not possible before. In particular, convolution neural networks (CNNs) have demonstrated their effectiveness in image detection and recognition applications. In this article, we talk about how to implement Deep Learning CNN algorithm to create encoders that compress dimensions and decoders that restore them by leveraging the power of parallel computing inside the FPGA fabric and speeding up the development process using High-Level Synthesis (HLS).

CHAPTER 1. INTRODUCTION

With the explosion of big data, efficiently managing and processing information has become an increasingly significant challenge. Hence, our project focuses on developing efficient encoders and decoders capable of compressing information into low-dimensional space and accurately restoring the original data.

We will proceed with designing and training encoder models with the ability to compress information to the smallest dimensional size. Simultaneously, we will create decoders that can restore data with minimal loss. These encoders and decoders will not only reduce data size but also ensure the quality and accuracy of the data after restoration.

By combining in-depth knowledge of Convolutional Neural Networks (CNN), C++ programming, and the high-level synthesis tool Vitis_HLS, we are committed to creating a comprehensive, innovative, and flexible solution. Our project not only promises to win awards in the LSI Design Contest but also contributes to the progress of the data processing and embedded systems field. Let's together conquer the challenge and make unique contributions to this captivating theme!

The chapters that this project document contains are briefly described below:

- **Proposed algorithm**: This is the core chapter of the project. In this chapter there is first a description of the architecture of the our project.
- **Results**: The research results collected in this chapter include summarized results.
- **Conclusions and future development**: The conclusions of the project are described in this chapter including some future developments are commented to motivate future thesis or research projects.

At the end of the project, there is detailed the literature consulted in this thesis, as well as the project examples and trainings followed in the Bibliography chapter. And finally, there are a set of Appendixes and a section defining the Acronyms and Glossary appearing in the report.

CHAPTER 2. PROPOSED ALGORITHM

The development of the project is divided into different steps in order to easily focus each part of the whole system and make it work itself as a smaller system. This way, at the end of the project, the whole system will be merged using the acquired know-how from the previous steps. This working methodology has been selected because of the complexity of the work field and the lack of knowledge about it from the company.

Accordingly, this section is composed by first a description of the final system and its architecture and then all the steps required to complete the work.

2.1 Specifications and flow design

Specifications:

- Input image: 28×28.
- Design environment: Vitis_HLS and Vivado

Flow design:

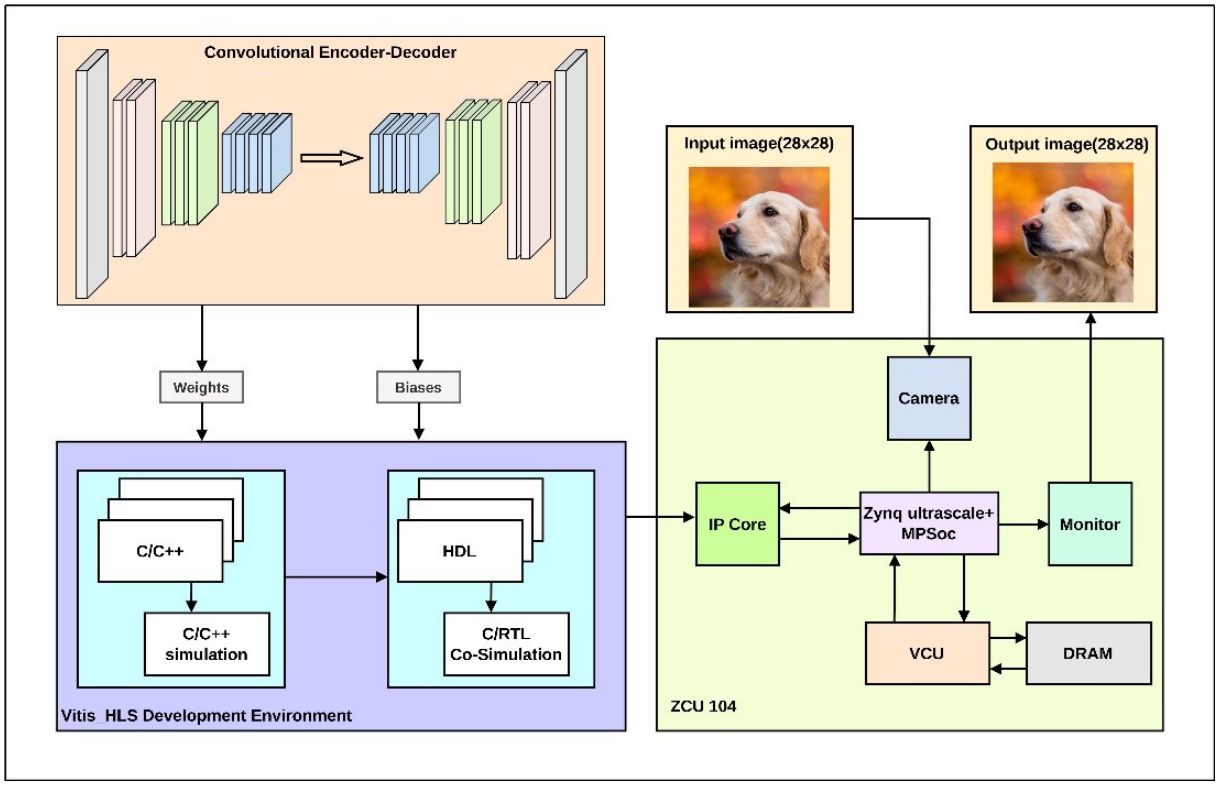


Figure 2.1 Flow design of our project

Our project comprises three main components, all aimed at optimizing the image

compression process through an efficient Convolutional Neural Network (CNN) system. The initial step involves designing and training a CNN model to achieve optimal performance in image compression. This requires careful consideration of network architecture, hyperparameters, and appropriate training datasets.

Upon successful model training, we will generate parameter files, including weights and biases, from the trained model. These parameters play a crucial role in replicating the model on a different platform: the C++ programming language. We will convert the model from the Deep Learning model language to C++ source code to seamlessly integrate it into embedded systems.

The next step in the project is high-level synthesis using the Vitis_HLS tool. Through this process, C++ source code will be transformed into Verilog, a popular language in integrated circuit design. This transformation results in an Intellectual Property (IP) block with the functionality to perform image compression based on the constructed CNN model.

Finally, this IP block will be integrated into the ZCU 104 controller using the Vivado tool. This integration will create a complete system capable of high-performance image compression and flexible integration into various embedded applications. Our project promises to deliver an effective solution with deep integration for the image compression challenge in embedded systems and mobile devices

2.2 Design Convolutional Neural Network for image compression

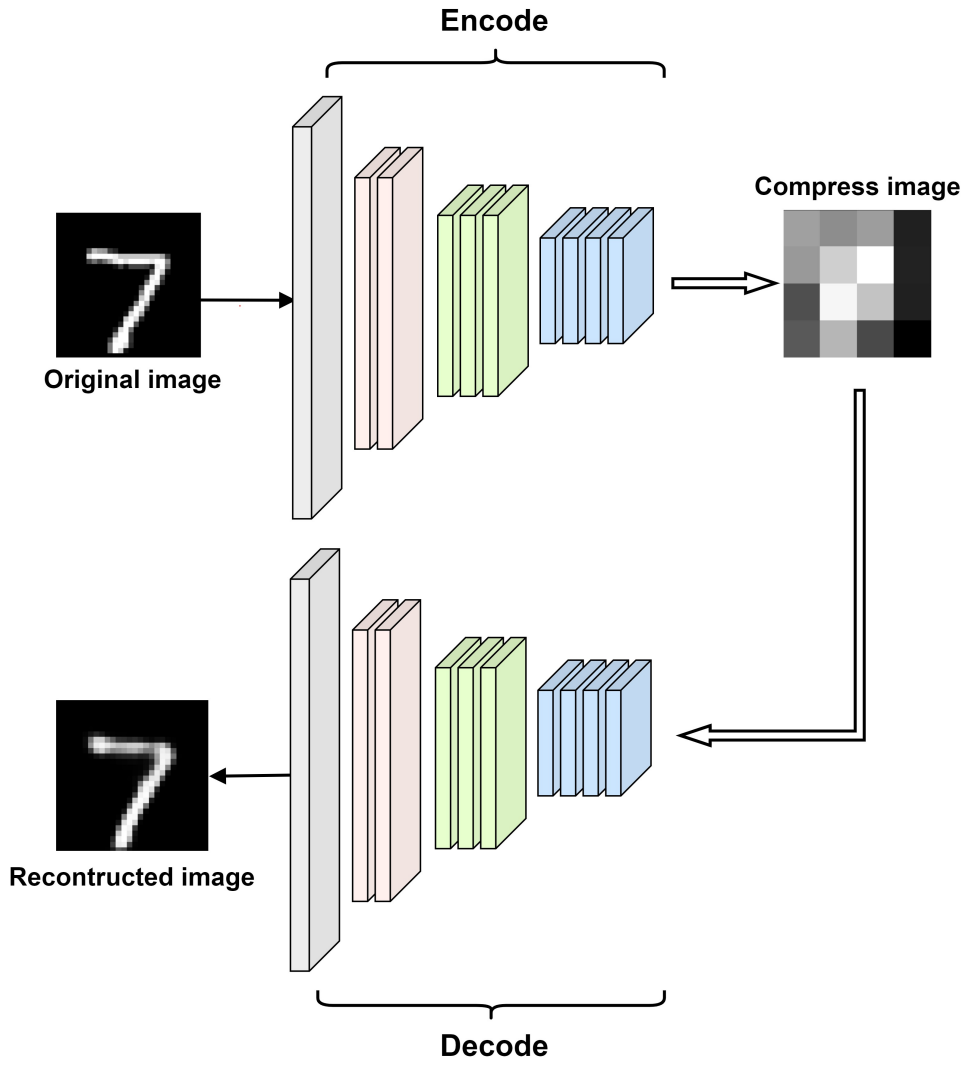


Figure 2.2 Convolutional flow design

To perform the task of image compression and decompression, we constructed an Autoencoder network based on convolutional layers. The architecture depicted in Figure 1 comprises two distinct parts - ComCNN and RecCNN. The initial part of the network consists of several convolutional layers to learn low-level features of the image. Small filters are utilized to capture local features, followed by Maxpooling2D layers. The responsibility of ComCNN is to compress these images in a manner such that the resulting images can be efficiently reconstructed by the decoding network to closely resemble the original images.

The decoding part, RecCNN, includes convolutional layers and Upsampling2D layers. The entire model is trained on the MNIST handwritten digit dataset, undergoing 500 training iterations. The trained weights are then transferred to C code using the High-Level Synthesis (HLS) framework of Vitis. Subsequently, the generated Verilog code,

along with the hardware description, is synthesized into a bitstream using Vivado. This bitstream, along with the Petalinux platform, creates an image file along with the root filesystem.

The generated bitstream, along with u-boot and the ELF file, is compiled into a boot.bin file. All these files are then written to an SD card for loading onto the ZCU104 board.

In summary, the process involves training the autoencoder model on the MNIST dataset, converting the trained model to C code using HLS, generating Verilog code, synthesizing a bitstream, creating a boot image with Petalinux, and finally, writing the resulting files onto an SD card for deployment on the ZCU104 board.

2.3 Traning models AI

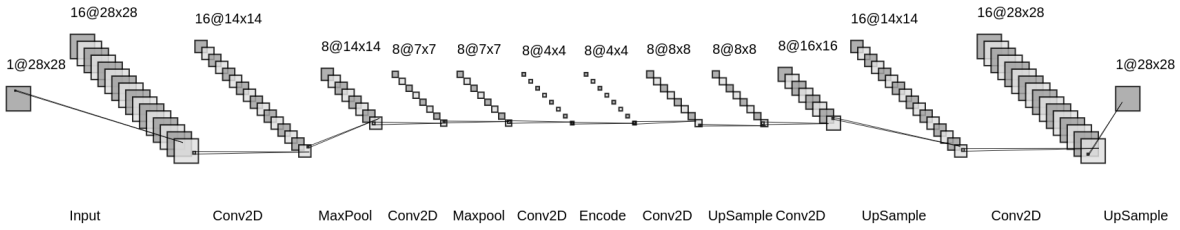


Figure 2.3 Auto Encoder model

Inspired by the Keras Autoencoder, our team constructed an Autoencoder model based on the Convolutional Neural Network (CNN) architecture. Using the Keras framework, we designed the network with two main components: Encode and Decode. The model was trained on the Mnist handwritten digit dataset, where the input consists of grayscale images with dimensions of 28×28 pixels.

The Encode portion comprises three Convolutional layers with a 3×3 kernel size each and three Maxpooling layers with a 2×2 size. The encoded image is obtained by passing it through the third Maxpooling layer with a size of $4 \times 4 \times 8$. Subsequently, this encoded image undergoes four Convolutional layers and three Upsampling layers in the Decode section to reconstruct the compressed image.

The model optimization was performed using the Adam optimizer, employing binary crossentropy 2.1 as the loss function. Evaluation metrics included two parameters: Structural Similarity Index (SSIM) and Peak Signal-to-Noise Ratio (PSNR). The training process involved 500 epochs with a batch size of 120. The formula for Mean Squared Error (MSE) 2.3 is described below.

$$-\frac{1}{N}\sum_{i=1}^{N}[y_i \cdot \log(p(y_i)) + (1-y_i) \cdot \log(1-p(y_i))] \tag{2.1}$$

The SSIM formula 2.2 relies on three parameters for comparison: luminance, contrast, and structure.

$$\text{SSIM}(x,y) = [l(x,y)]^{\alpha} \cdot [c(x,y)]^{\beta} \cdot [s(x,y)]^{\gamma} \tag{2.2}$$

The Mean Squared Error (MSE) 2.3 is defined as

$$MSE = \frac{1}{mn}\sum_{i=0}^{m-1}\sum_{j=0}^{n-1}[I(i,j) - K(i,j)]^2 \tag{2.3}$$

where I and K are the original and synthesized images, respectively.

PSNR 2.4 is determined through MSE for a two-dimensional image of size $m \times n$, and is given by

$$PSNR = 20 \cdot \log_{10}\left(\frac{\text{MAX}_i}{\sqrt{MSE}}\right) \tag{2.4}$$

In this context, MAX_i represents the maximum pixel value in the image. When pixels are represented by 8 bits, its value is 255. In the general case, when the signal is represented by B bits per sample, MAX_i is equal to $2^B - 1$.

2.4 Implementation using high-level synthesis by Vitis High level systhesis

In this section, we will move on to the deployment process of the previously designed algorithm. We will program to create an IP block capable of accurately executing the mentioned algorithms. In this process, we will use the C++ programming language and the High-Level Synthesis (HLS) 2.4 library to generate an IP block implemented in VHDL.

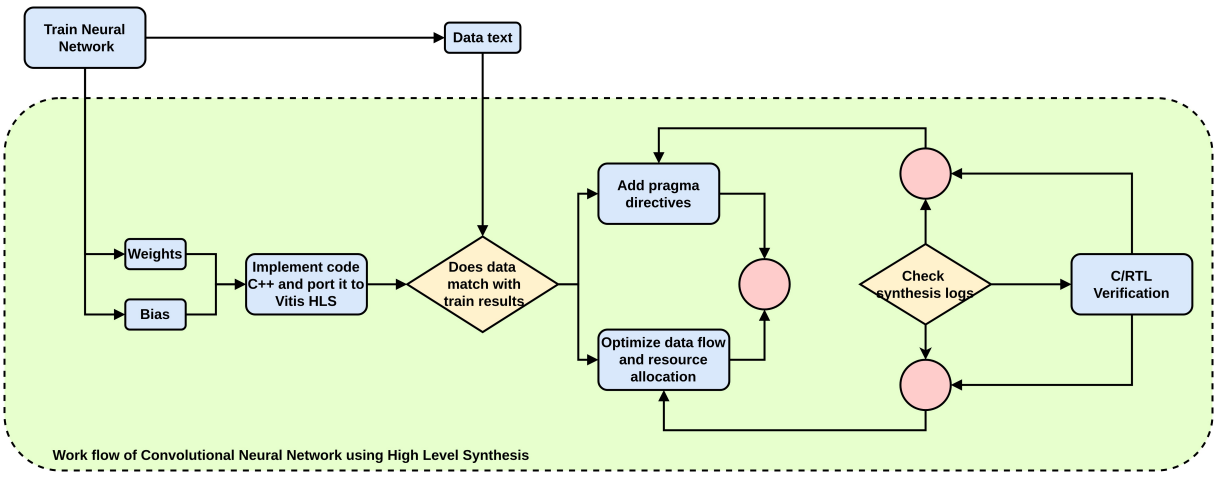


Figure 2.4 Workflow of Convolutional Neural Network using High Level Synthesis

2.4.1 Techniques used in the model

2.4.1.1 Linebuffer in Image Processing

Linebuffer is a data structure used to temporarily store a line of data from an image during processing. It is structured as a temporary memory, capable of maintaining one or multiple lines of data at a time. Operation Process of two linebuffers, in each processing cycle, new data from the image is input into the Linebuffer

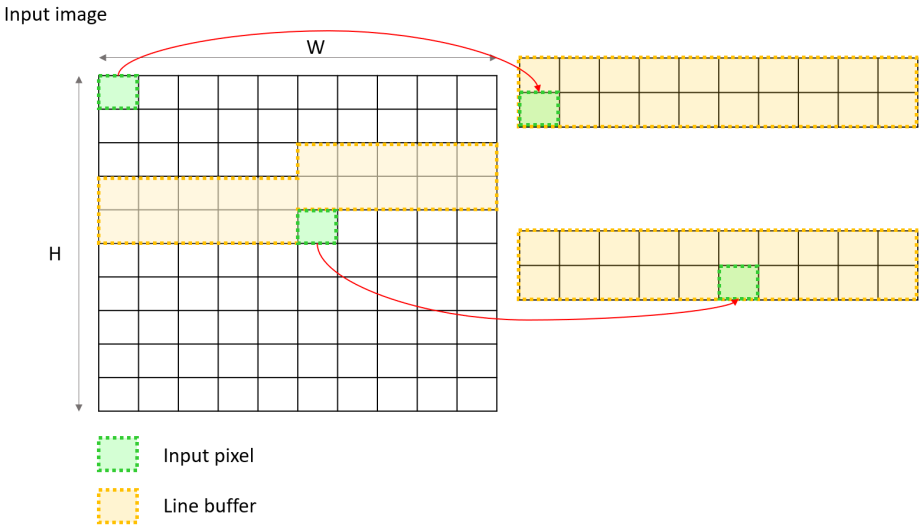


Figure 2.5 Two line bufer

The Linebuffer stores the previous line of data and provides it for the next processing step.

Each input data will be stored in row 2 of the buffer with the column corresponding to the input data (for example, if the column of the input pixel is 5, the newly stored data will be in column 5, row 2).

Before receiving additional input data, the linebuffer shifts the data of the column being examined up to row 1, as shown in the figure. Similarly, this continues until the last input of the image 2.5, 2.6.

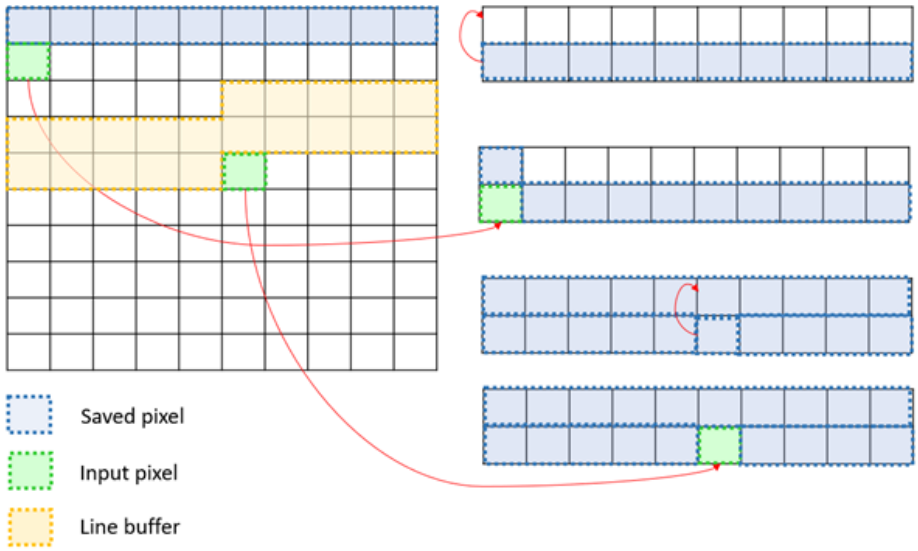


Figure 2.6 Two line bufer flow

Advantages of using 2-line buffer:

During convolution, linebuffer helps reduce the time to access data from the image, increasing the algorithm's efficiency. Maintaining the previous line of data helps reduce latency when applying convolution weights to small parts of the image.

The use of Linebuffer not only helps reduce latency but also optimizes performance by efficiently maintaining and organizing data. This structure is particularly important when deployed on FPGA hardware, where resources are precious, and memory access can be costly.

2.4.1.2 Window in Image Processing

The window 2.7 is a data structure or strategy used to extract small portions of an image during processing. In the context of image processing, the window is often used to apply convolution and pooling operations to small regions of the image. Operation Process of the Window:

The window typically moves over the image with a certain step (stride) to continuously extract small portions.

The size of the window can be adjusted depending on the requirements of the specific algorithm and the size of the region of interest.

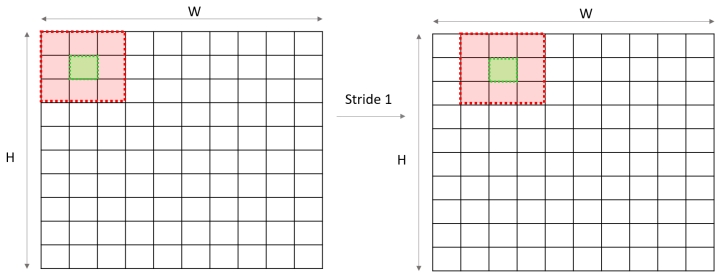


Figure 2.7 Window for image processing

The value of the window is updated from the window and input data. When the window shifts with a stride of 1, the value of the window will always be updated. In row 2, the value will be updated from the buffer in the above paragraph and the input pixel value as shown in the figure. Then when the window moves away, the window value is updated by shifting the data to the left by 1 column, and at column 2, the data is updated similarly to the above. Impact of the Window in Convolution and Pooling 2.8:

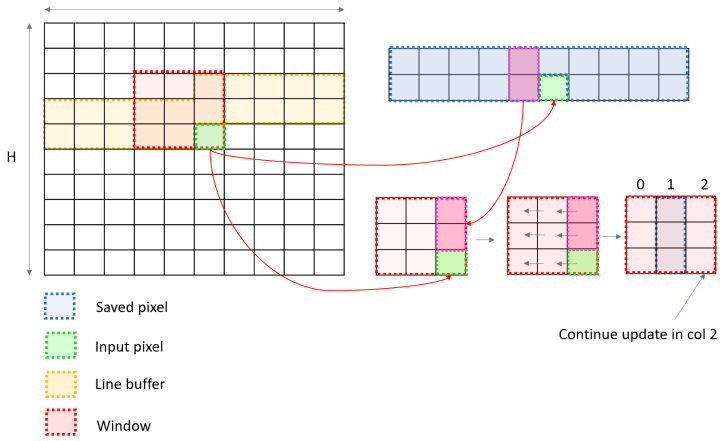


Figure 2.8 Window update data

2.4.1.3 Padding

Padding 2.9 is a technique of adding zero values around the edges of an image to maintain the size of the image across layers of a CNN model.

In the deployment process, adding zero values is performed as part of the data preprocessing to prepare the input for convolution and pooling layers.

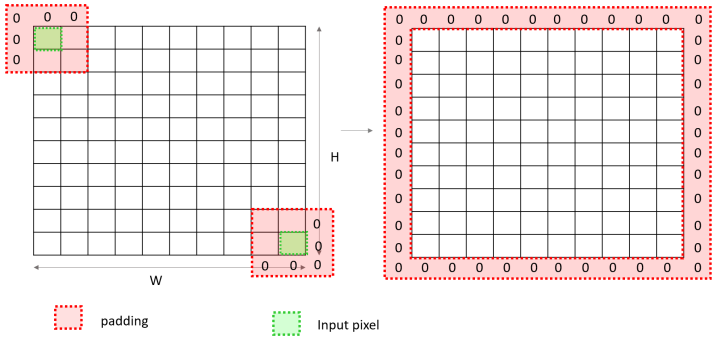


Figure 2.9 Padding

Padding increases the size of the image representation, preserving information at the edges and helping avoid the loss of important information. In convolution layers, padding helps minimize edge effects and maintains the size of the output.

2.4.2 Optimize fix-pointed representation

The second technique I employ is optimized fixed-point representation, a strategy crucial for efficiently utilizing system resources while maintaining desired performance levels. In this context, I focus on tailoring fixed-point representations to suit the capabilities of the ZCU 104 platform.

One key consideration is the length of the bit width, which directly impacts resource allocation. By carefully setting the fixed-point representation, I can strike a balance between resource utilization and achieving satisfactory image results. This involves adjusting the number of integer bits in each layer to refine the data range and select an appropriate width for representation 2.1.

Through experimentation and analysis, I've found that transitioning from a 32-bit to a 24-bit representation yields significant benefits. While reducing the bit width, I ensure that the quality of the images, as measured by metrics such as Peak Signal-to-Noise Ratio (PSNR), remains consistent. By fine-tuning the integer bits within each layer, I optimize resource utilization without compromising image quality 2.2.

In summary, optimizing fixed-point representation involves strategically selecting bit widths and adjusting integer bits to maximize efficiency while preserving image quality 2.3. This approach enables us to achieve a desirable balance between resource utilization and performance on the ZCU 104 platform.

Table 2.1 Set interger bits in layers

Layer	Conv1	Pool1	Conv2	Pool2	Conv3	Pool3	Conv4	Upsa4	Conv5	Upsa5	Conv6	Upsa6	Conv7
Range (min)	-1.51	0	-4.43	0	-1.15	0	-2.16	0	-0.74	0	-1.9	0	0.66
Range (max)	1.12	1.15	0.99	3.13	0.83	6.64	5.67	5.67	1.15	7.86	19.51	19.51	0.98
Interget bit	**3**	**3**	**4**	**3**	**3**	**4**	**4**	**4**	**3**	**4**	**3**	**3**	**4**

Table 2.2 Resources used corresponding with bit width

Bit width	PSNR (dB)	FF	LUT	DSP	Latency (ms)
32	68.1	410249	458106	6172	0.155
31	68.2	379368	427192	5126	0.155
30	67.6	347151	392675	4577	0.155
29	65.4	337869	356504	3822	0.150
28	64.5	308463	292575	3194	0.150
27	64.1	261412	231142	2256	0.150
26	63.4	232594	184695	1975	0.150
25	63.4	217641	160482	1723	0.150
24	**62.8**	**170542**	**111408**	**1611**	**0.150**

Table 2.3 Resources used in ZCU104

	FF	**LUT**	**DSP**
Available (ZCU104)	460800	230400	1728
Used	170542	111408	1611
Utilization (%)	37	48	93

2.4.3 Application of Algorithms in CNN

2.4.3.1 Convolution

Convolution Neural Network (CNN) 2.10 is designed to extract important features and structures from image data.

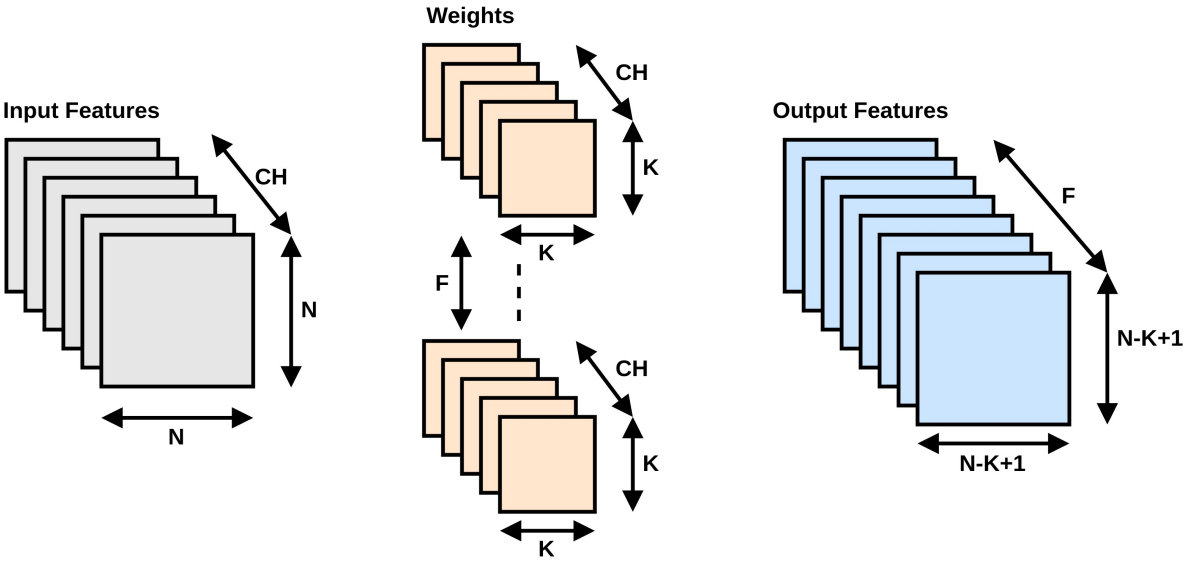


Figure 2.10 Convolution neural network

To implement this algorithm, I use line buffer and window to perform convolution with the kernel. This process is carried out through the following steps:

1. **Initialize Initial Parameters**:

```
int acti, padding, fil_in, fil_out, width, height;
T *linebuf, *win, *bias2, *ker;
hls::stream<T> &src, &dst;
```

- acti: Set the activation function corresponding to convolution (0 for ReLU and 1 for sigmoid).
- fil_in: Depth of the input channel.
- fil_out: Depth of the output channel.
- width: Width of the input channel.
- height: Height of the input channel.
- linebuf: Line buffer array.
- win: Window array.
- ker: Kernel data array (weights).

- bias2: Bias for the convolution layer.
- src: Input data stream.
- dst: Output data stream.

2. **Set Up Line Buffer and Window**:

```
**For Line Buffer:**
// Shift values in line buffer vertically
ShiftLineBuf: for (int i = 0; i < K-1 ; i++) {
   #pragma HLS PIPELINE
   T temp = (i < K - 2) ? linebuf[num_fil1*(K-1)*width+(i+1)*width+
      pool_col] : in_val;
   linebuf[num_fil1*(K-1)*width+i*width+pool_col] = temp;
}
```

Input variables are updated at row 1 and the current column in the input data. Before fetching new data, the value at the current column is shifted up to the upper column. This process is then repeated to maintain the line buffer.

```
**For Window:**
if (pool_row >= K - 1) {
   // Shift values in the window horizontally
   CUpdateWinH: for (int win_row = 0; win_row < K; win_row++) {
       CUpdateWinW: for (int win_col = 0; win_col < K; win_col++) {
           #pragma HLS PIPELINE
           if (win_row < K - 1) {
               win[num_fil1*K*K+ win_row*K+ win_col] = (win_col < K -
                   1) ? win[num_fil1*K*K+ win_row*K + (win_col + 1)] :
                   linebuf[num_fil1*(K-1)*width+ win_row*width +pool_col
                   ];
           } else if (win_row == K - 1) {
               win[num_fil1*K*K + win_row*K + win_col] = (win_col < K -
                    1) ? win[num_fil1*K*K+ win_row*K + (win_col + 1)] :
                   in_val;
           }
       }
   }
}
```

When the data reaches row 2 of the input channel, the window starts to update. Column 2 of the window takes values from the line buffer and input pixel. Specifically, row 0 and row 1 take values from row 0 and row 1 of the current column. This process continues to maintain the sliding window over the input data.

3. **Convolution**: When examining a column with a value greater than or equal to 2, a new convolution begins. This operation multiplies each value of the window with the corresponding kernel value and then sums them up.

```
CFilter: for (int num_fil2 = 0; num_fil2 < fil_out; num_fil2++) {
   if (pool_col >= K - 1) {
        CWinH: for (int win_row = 0; win_row < K; win_row++) {
            CWinW: for (int win_col = 0; win_col < K; win_col++) {
                #pragma HLS PIPELINE
                data[num_fil2] += win[num_fil1*K*K + win_row*K + win_col
                    ] * ker[num_fil2*fil_in*K*K+num_fil1*K*K+win_row*K+
                    win_col];
            }
        }
   }
}
```

4. **Apply Activation Function and Save Result**: After obtaining data from the previous step, it undergoes the activation function to normalize the data. Then the result is saved into the output.

```
for(int i=0; i<fil_out; i++) {
   if(acti == 0) {
        dst << relu((data[i]+bias2[i]));
   } else {
        dst << sigmoid((data[i]+bias2[i]));
   }
}
```

An illustrative example is shown in the figure 2.11, 2.12 below.

Figure 2.11 Updating data operation

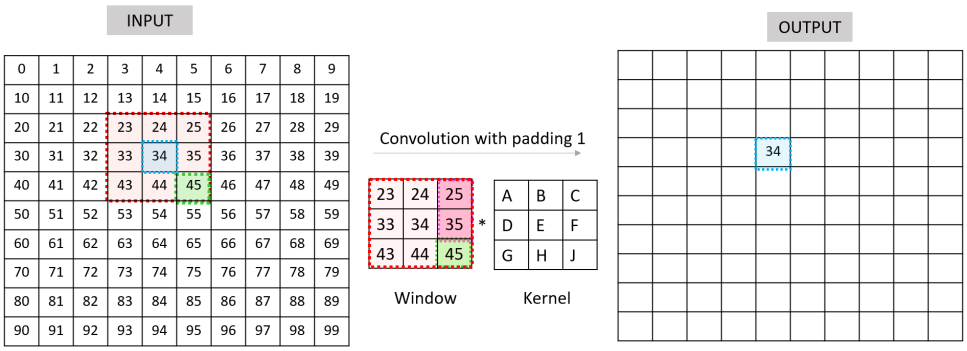


Figure 2.12 Convolution with padding

2.4.3.2 Pooling

Pooling is a crucial component of Convolutional Neural Network (CNN) models designed to reduce the size of representations and enhance important features from image data. Pooling layers typically operate by dividing the image representation into pools and retaining either the maximum value (Max Pooling) or the average value (Average Pooling) of each pool. In the provided code snippet, we implement Max Pooling with a window size of **POOL_SIZE**. This process is carried out through the following steps:

1. **Initializing Parameters**:

```
int padding, int fil, int width, int height, T *pool_buf, T *pool_win,
    hls::stream<T> &in_pl, hls::stream<T> &out_pl
```

   Parameters include padding size, input channel depth, input width, input height, pool buffer array, window array, input stream, and output stream.

2. **Updating the Window**: The 'pool_win' window is updated with data from 'in_pl' and 'pool_buf'. Similar to the kernel part, the values of the window and line buffer are updated. However, unlike the convolution layer, here the line buffer has only one row, and the size of the window is 2×2.

```
if (pool_row >= K - 2) {
    PUpdateWinH:for (int pool_win_row = 0; pool_win_row < POOL_SIZE;
        pool_win_row++) {
        PUpdateWinW:for (int pool_win_col = 0; pool_win_col < POOL_SIZE
            ; pool_win_col++) {
            #pragma HLS PIPELINE
            if (pool_win_row < K - 2) {
                pool_win[num_filters*POOL_SIZE*POOL_SIZE + pool_win_row
                    *POOL_SIZE + pool_win_col] = (pool_win_col < K - 2)
                    ? pool_win[num_filters*POOL_SIZE*POOL_SIZE +
                    pool_win_row*POOL_SIZE + (pool_win_col+1)] :
                    pool_buf[num_filters*width + pool_col];
```

```
            } else if (pool_win_row == K - 2) {
                pool_win[num_filters*POOL_SIZE*POOL_SIZE + pool_win_row
                    *POOL_SIZE + pool_win_col] = (pool_win_col < K - 2)
                    ? pool_win[num_filters*POOL_SIZE*POOL_SIZE +
                    pool_win_row*POOL_SIZE + (pool_win_col+1)] :
                    in_pool_val;
            }
        }
    }
}
```

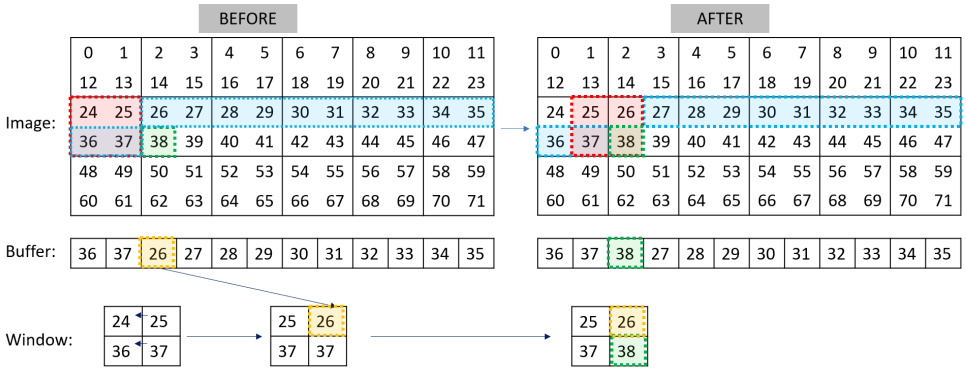


Figure 2.13 Update Convolution

3. **Pooling Operation**: The window is updated until the value is multiplied in an odd column, and then the comparison is initiated to find the maximum value in the window.

```
if (pool_col % (K - 1) == 1 && pool_row % (K - 1) == 1) {
    PWinH:for (int pool_win_row = 0; pool_win_row < POOL_SIZE;
        pool_win_row++) {
        PWinW:for (int pool_win_col = 0; pool_win_col < POOL_SIZE;
            pool_win_col++) {
            #pragma HLS PIPELINE
            if (temp <= pool_win[num_filters*POOL_SIZE*POOL_SIZE +
                pool_win_row*POOL_SIZE + pool_win_col]) {
                temp = pool_win[num_filters*POOL_SIZE*POOL_SIZE +
                    pool_win_row*POOL_SIZE + pool_win_col];
            }
        }
    }
```

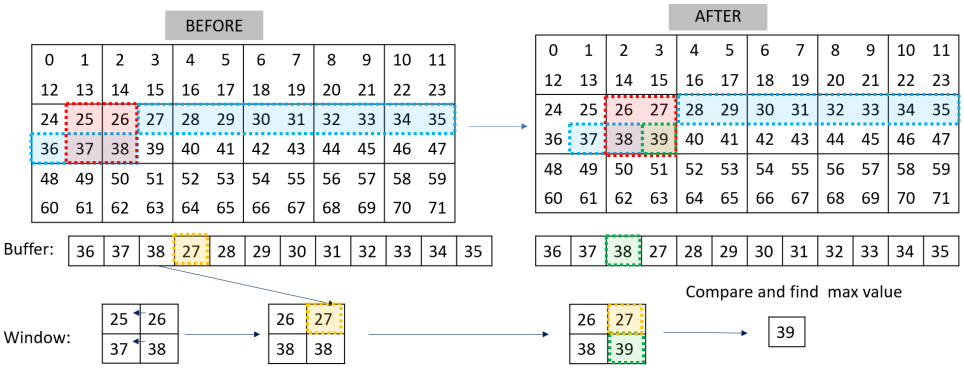


Figure 2.14 Pooling Operator

4. **Data Storage**: The size of the data output will be halved in both width and height after Max Pooling.

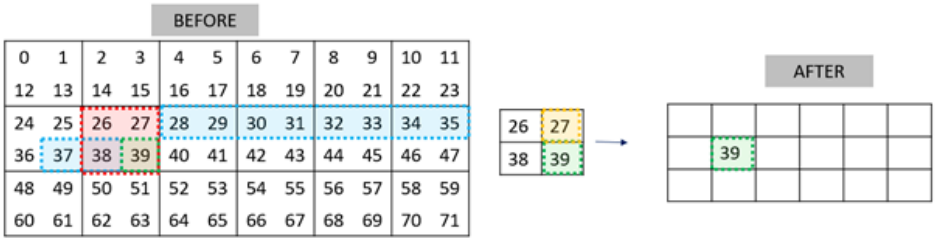


Figure 2.15 Pooling Data storage

The use of pooling layers helps reduce the size of representations and computational complexity in CNN models. It retains essential features by selecting the maximum value within each pool, effectively downsizing the output dimensions.

2.4.3.3 Upsampling

Upsampling with Nearest Neighbor Interpolation serving to increase the size of the output data without sacrificing essential information. This process is commonly used to reconstruct the resolution of images and generate more detailed representations of features.

In this implementation, we utilize Nearest Neighbor Interpolation, a technique that calculates new values based on the values at the nearest grid points.

Upsampling Function

```
template<typename T>
void sp_upsamp(int fil,int width, int height, T *upsam_buf,hls::stream<T> &
    in_us1, hls::stream<T> &out_us1) {
    UHeight:for (int cona_row = 0; cona_row < 2 * height; cona_row++) {
        UWidth:for (int cona_col = 0; cona_col < 2 * width; cona_col++) {
```

```
            #pragma HLS PIPELINE
            UFils:for (int cona_depth = 0; cona_depth < fil; cona_depth++)
                {
                 if(((cona_row % 2) == 0) && ((cona_col % 2) == 0)){
                     T temp = in_us1.read();
                     upsam_buf[cona_depth*width+(cona_col / 2)] =temp;
                 }
                 out_us1 << upsam_buf[cona_depth*width+(cona_col / 2)];
            }
        }
    }
}
```

2.4.3.4 Activation Functions: ReLU and Sigmoid

Activation functions in Convolutional Neural Networks (CNNs) that introduce non-linearity, enabling the network to learn complex patterns and representations. Two commonly used activation functions are Rectified Linear Unit (ReLU) and Sigmoid.

1. **Rectified Linear Unit (ReLU)**: ReLU is a popular activation function that introduces non-linearity by allowing the positive values to pass through while setting all negative values to zero. This simple yet effective function helps CNNs learn intricate features and patterns in the input data.

```
data_t relu(data_t a) {
    return a > (data_t)0 ? a : (data_t)0;
}
```

2. **Sigmoid**: Probabilistic Output: Sigmoid is suitable for tasks requiring probability outputs, making it commonly used in the final layer of a CNN for binary classification.

   Smooth Gradient: Sigmoid provides smooth gradients during backpropagation, aiding in stable training.

```
data_t sigmoid(data_t x) {
    return 1.0 / (1.0 + expf(-x));
}
```

2.4.3.5 High Level Synthesis

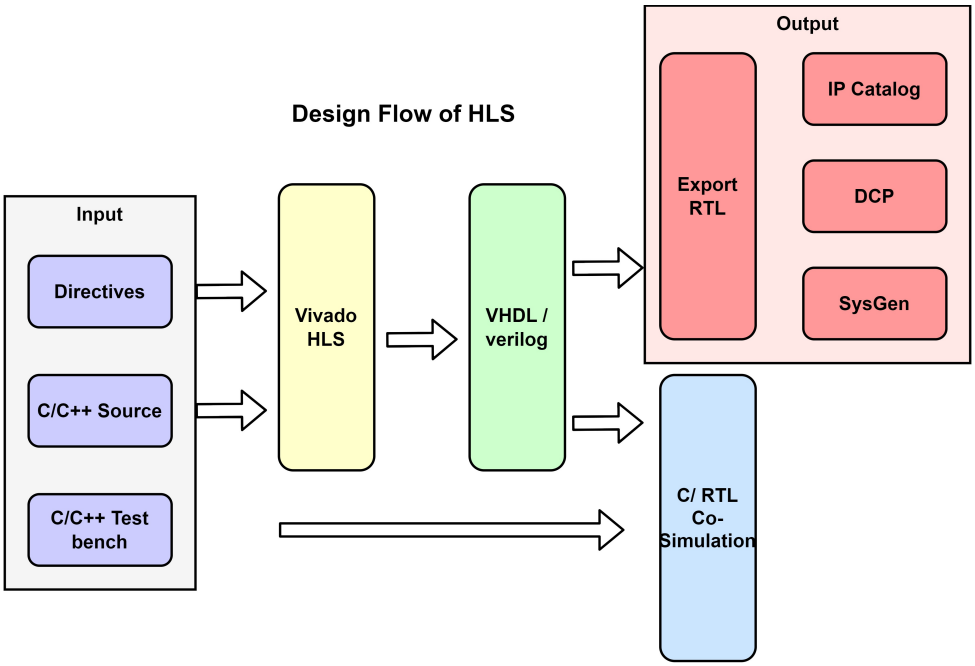


Figure 2.16 Design Flow of HLS

With HLS tools, developers can deploy algorithms and functions from a high-level language such as C or C++, reducing the complexity of the development process and optimizing the hardware performance. It serves various purposes, including:

Accelerating Development Process: Using a high-level programming language helps reduce development time compared to using low-level languages like Verilog or VHDL. Developers can focus more on algorithmic modeling rather than the details of hardware implementation.

Resource Optimization: HLS has automatic optimization capabilities, effectively utilizing hardware resources. This is particularly crucial when working on platforms with limited resources, such as FPGAs.

Flexible Control: Features like stream, AXIS, fixed-point, and pragma provide developers with flexible control during the automatic optimization process. This helps optimize hardware according to the specific requirements of the application.

1. **Stream**:

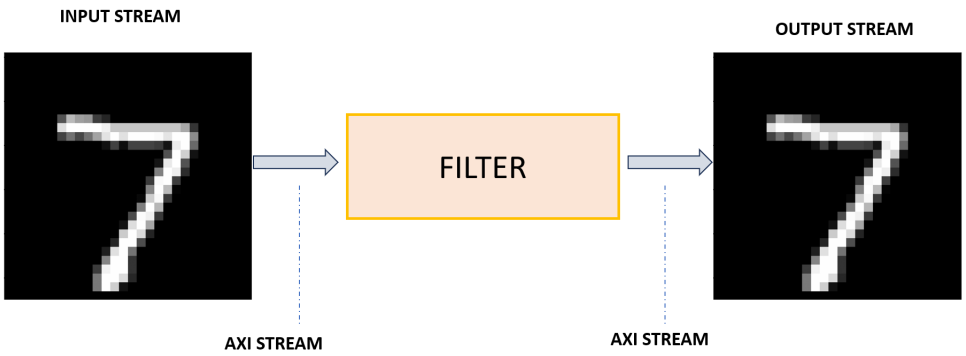


Figure 2.17 Stream HLS

Streaming is a data transfer method where data samples are sent sequentially from the first sample. Modeling designs using streaming data can be challenging in C. The use of pointers to perform multiple read and/or write accesses can introduce issues related to type qualifiers and test bench construction.

Vitis HLS provides a C++ template class ‘hls::stream<>‘ for modeling streaming data structures. These streams have the following attributes:

- In C code, ’hls::stream<>’ acts like an infinite-depth FIFO.
- They are read and written sequentially, meaning that once data is read from an ’hls::stream<>’, it cannot be read again.
- An ’hls::stream<>’ on the top-level interface is, by default, implemented with an ’ap_fifo’ interface for the Vivado IP flow or as an ‘axis‘ interface for the Vitis kernel flow.
- Streams can be named, and the depth of the FIFO can be adjusted.

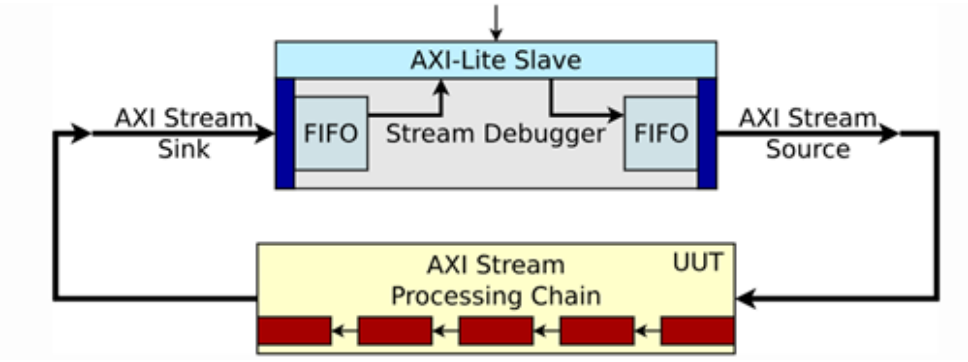


Figure 2.18 Stream HLS

In addition, my project utilizes the AXI Stream interface for handling these streams efficiently. The choice of the AXI Stream interface further enhances the communication between different hardware components, ensuring seamless and standard-

ized data transfer. This approach not only simplifies the implementation of read and write operations but also contributes to the overall optimization of computational processes within the project.

2. **Fixpoint**: The Fixed-Point section in the report plays a crucial role in optimizing performance and resource utilization in the project. They have the following advantages:

   - Reduced LUT Usage:
     - Utilizing Fixed-Point instead of floating-point numbers can simplify the logic circuit complexity and reduce the number of Look-Up Tables (LUTs) used.
     - Enhances the potential for RTL optimization, especially on resource-constrained platforms like FPGA.
   - Performance Optimization:
     - Fixed-Point can reduce a certain level of precision without significantly impacting the results, helping to decrease computation load and accelerate processing.
     - It can enhance computational capabilities and reduce execution time for hardware with limited resources.
   - Resource Savings in HDL Tools:
     - Fixed-Point Representation can decrease the required memory footprint and reduce the amount of combinational resources used during synthesis.
     - The Fixed-Point section in the project not only provides an effective resource utilization approach but also brings significant benefits in terms of performance and optimization throughout the development process.

   In this project, the decision to use Fixed-Point with 40 bits and allocate between the integer and fractional parts is a reasonable choice. You've allocated 10 bits for the integer part and 30 bits for the fractional part, providing a good balance between precision and representation size.

3. **Pragma**: Pragma plays a crucial role in guiding or modifying how High-Level Synthesis (HLS) tools organize and synthesize source code to optimize performance and utilize hardware resources. Below are some pragmas I use in my project:

   - Dataflow Pragma: #pragma HLS dataflow: This pragma is used to create a DATAFLOW region, where all implemented functions or source code blocks operate independently and in parallel. This helps optimize performance by allowing functions to perform computations without waiting.

- Pipeline Pragma: #pragma HLS pipeline: This pragma indicates that loops in the source code can be divided into stages and executed in parallel. This helps optimize the working speed of the logic circuit.

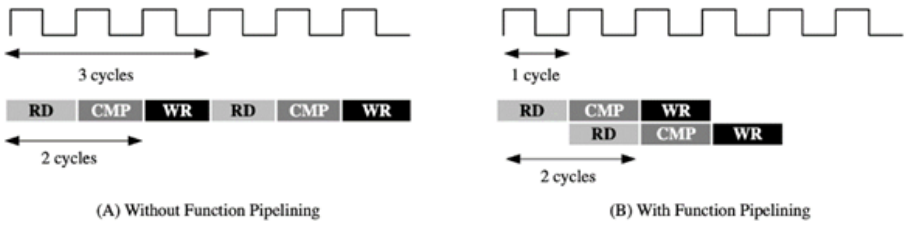


Figure 2.19 Pragma

- Interface Pragma: #pragma HLS interface: This pragma is used to define the module's interface in the project, including AXI4-Stream, AXI4-Master, AXI4-Slave interfaces, and various other interface types.
- Dependence Pragma: #pragma HLS dependence variable: This pragma is used to identify dependencies between variables, helping the HLS tool better understand the dependency relationships between data.

CHAPTER 3. FPGA IMPLEMENTATION

In this section, we implement the CNN algorithm in the previous section in FPGA known as the *Zynq™ UltraScale+™ MPSoC ZCU104* to evaluate with benmarks. This is a very complete board with a large number of capabilities intended to be used in a lot of different applications.

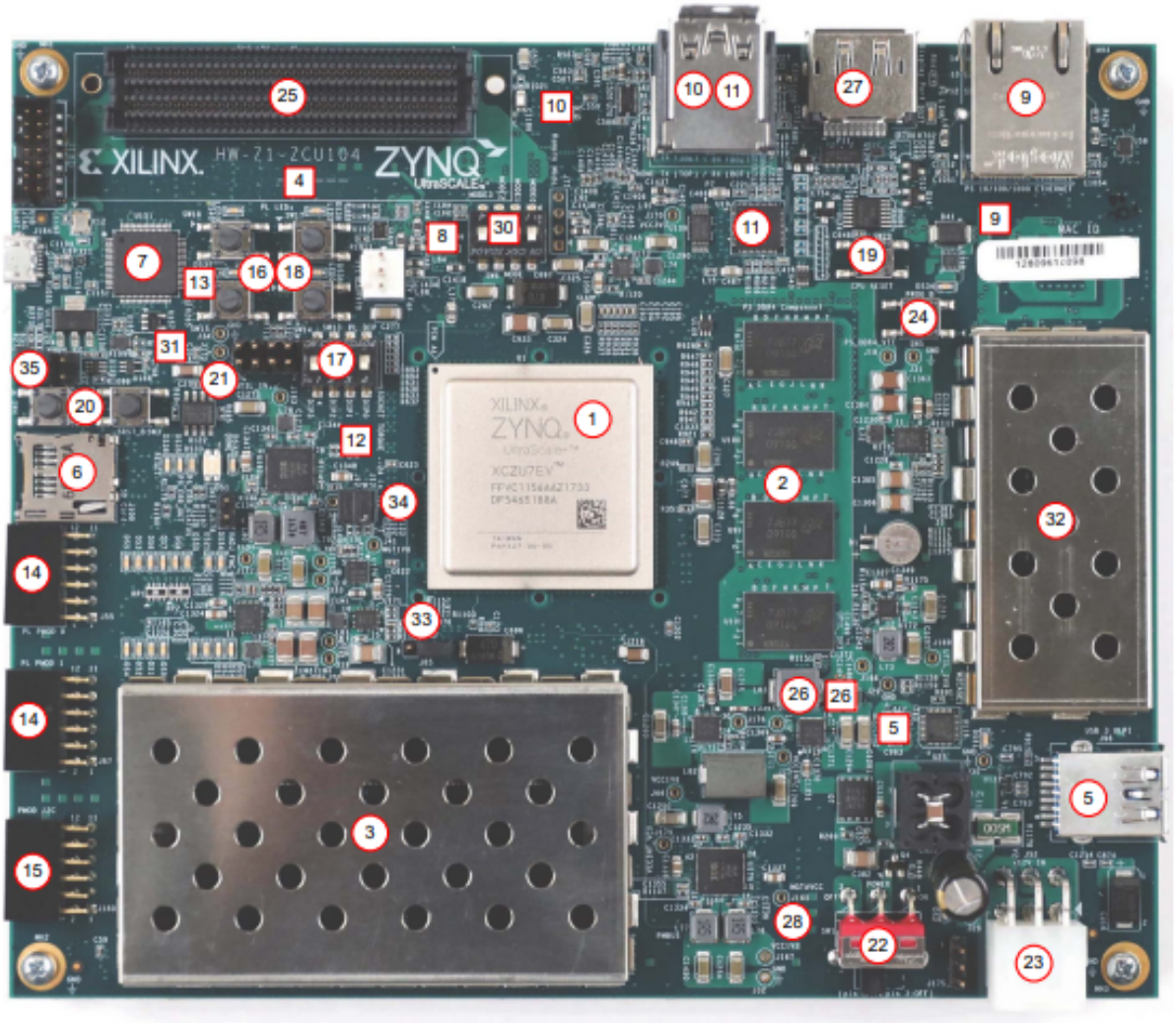


Figure 3.1 Zynq™ UltraScale+™ MPSoC ZCU104

With the most relevant for this project are listed in table.3.1:

1	Zynq UltraScale+ XCZU7EV MPSoC
3	PL-Side SODIMM DDR4
5	USB 3.0 Transceiver and USB 2.0 ULPI PHY
6	SD Card Interface connector
7	Programmable Logic JTAG
27	Display Port connector

Table 3.1 Relevant components for this project

3.1 Data Acquisition System Design

The diagram depicts the data acquisition system. The input is a camera sending a video stream via USB 3.0 to the PS. This communication is facilitated by the USB Video Class driver, transmitted to the PL for data processing. After processing, the results are returned to the PS and transmitted to the Display Port block for external display. The communication between the Display Port and the monitor is handled by the kmssink plugin. Figure.3.2:

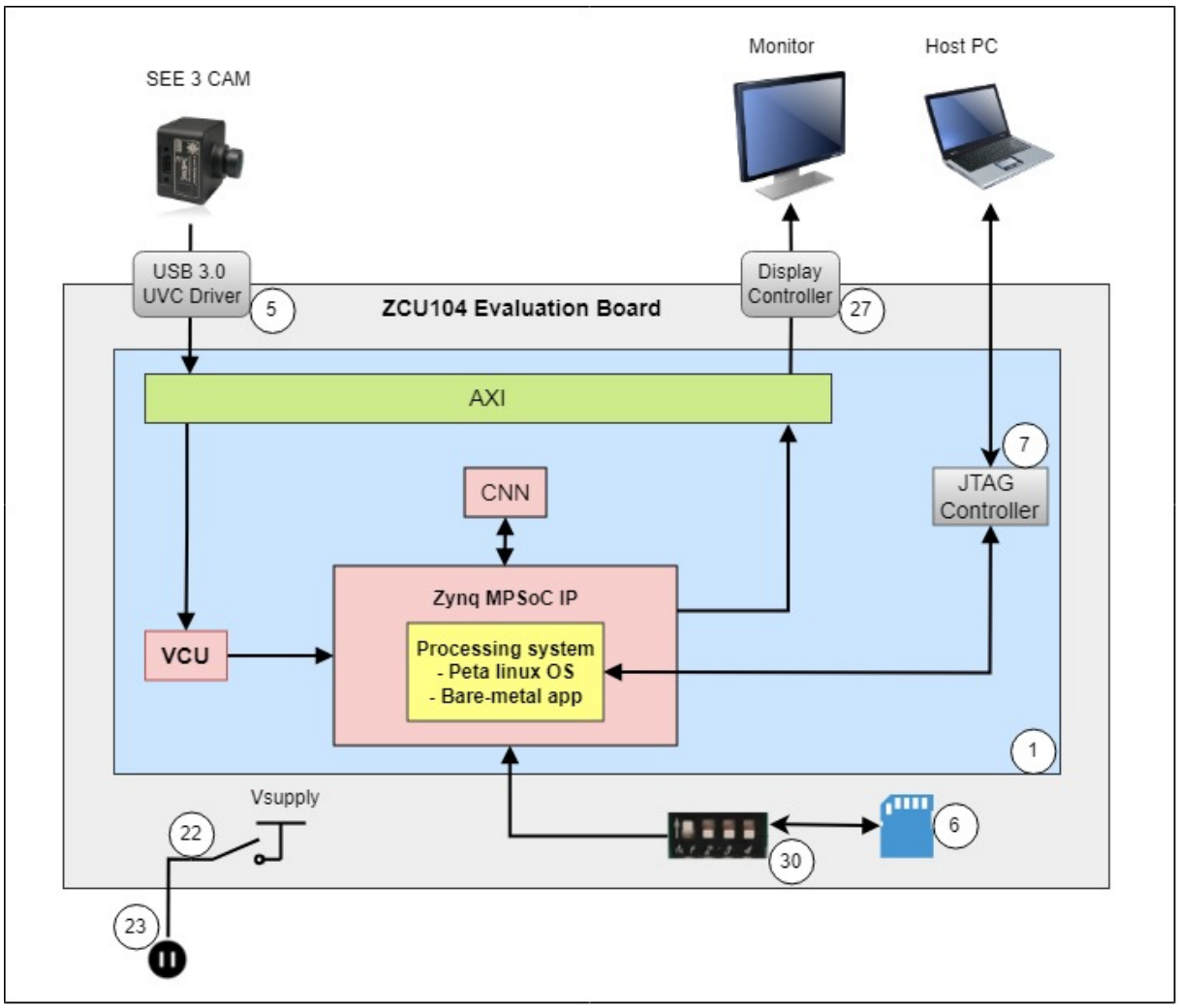


Figure 3.2 System architecture

The input data is transmitted from the camera through the USB3.0 block on the PS to the CNN within the PL. Therefore, an intermediary block connecting the PS and PL is necessary for the camera data to reach the CNN. Hence, the MPSoC IP block needs to be added to the system.

The operating principle of the MPSoC differs from how a processor operates when communication between peripherals (I/O) and memory occurs via DMA. The central processing block, A53 ARM Cortex (APU), retrieves input data from peripherals (I/O) transmitted to the PS memory via DMA to be sent to the PL. After processing in the PL, the data is transmitted back to the PS memory, and then transferred to the peripheral devices (I/O) via DMA.

Additionally, a VCU (Video Codec Unit) block is required to encode the input data from the camera to reduce the stored data size, saving memory space.

However, the transcoding process, switching between encoding and decoding (Transcoding), i.e., switching between the process of writing data to DRAM in the PS performed

by the Encoder and the process of writing data to DRAM performed by the Decoder, may encounter bandwidth congestion at a data rate of 60 frames per second with an image size of 1920x1080. Therefore, the solution to this issue includes:

- The communication and data transmission process after encoding by the Encoder (part of VCU) to the PS's DRAM remains unchanged.
- The communication and data transmission process between the Decoder and the PS's DRAM will pass through transmitting the decoded data to the DRAM block in the PL via the VCU DDR4 Controller block. Then, the data in the PL's DRAM will be copied to the PS's DRAM via the PS's DMA block.

3.2 Integrating CNN into the Data Acquisition System

Since the input for CNN will be a single RGB frame rather than a video stream, there will be some changes compared to the data acquisition system described in section 3.1. In this section, the encoded frame data will be stored in the RAM of the APU block (consisting of 4 Arm Cortex A53 cores) instead of storing it in the DRAM of the PS.

Then, the data of this frame will be decoded, stored in the PS's DRAM, and the DMA block will be used to read the data from memory and transfer it to the CNN. Therefore, the block diagram will be as follows 3.3:

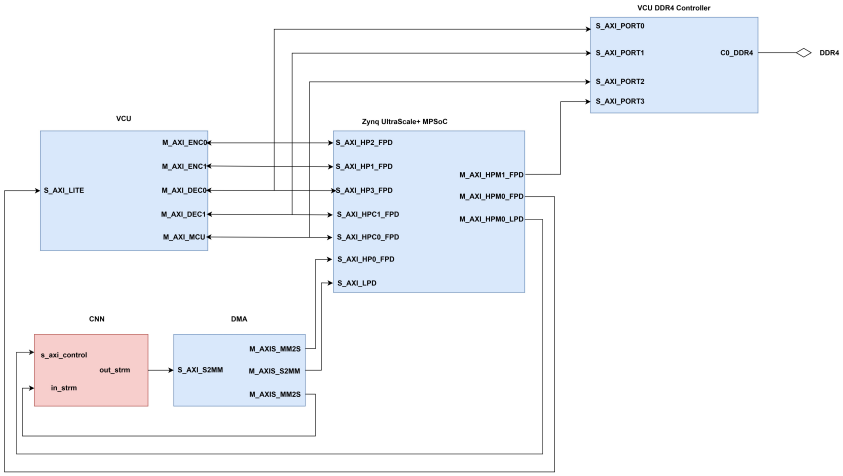


Figure 3.3 System block diagram

So the data flow when deploying Petalinux onto the system will consist of 4 stages:

1. Data captured from the camera is a video streamed into the DRAM on the PS. It is then sent to the VCU for encoding one frame and saved as an H264 encoded .ts file in the /home/ directory of Petalinux.

2. Utilizing the imread() function from OpenCV to read the .ts file saved in the /home/ directory and perform decoding. The data is then converted from YUV color space to BGR and then to RGB using the cvtColor() function. Finally, the decoded image is written into the /dev/mem directory using the imwrite() function.

3. Reading the decoded RGB image file from stage 2 and transferring it to the CNN through the DMA block.

4. After processing in the CNN block, the output image is exported and saved into the /dev/mem directory. /home/ : .ts -> (imread())BGR -> (cvtColor() RGB) -> (imwrite() write in /dev/mem RGB, CNN -> output$_image -> /dev/mem$

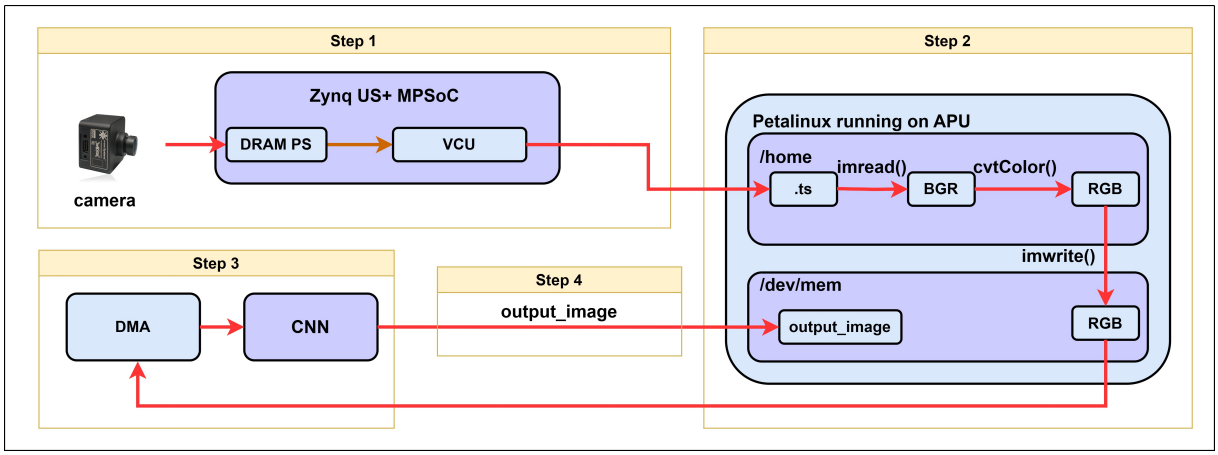


Figure 3.4 Diagram showing data flow

The .ts file is exported and saved in the /home/ directory for easy retrieval and comparison with the original image to calculate the compression ratio. After that, it is decoded and saved in /dev/mem/ to be sent to the CNN via DMA.

3.3 System deployment

An overview of the steps to deploy the system on Zynq MPSoC will be as shown in 3.6

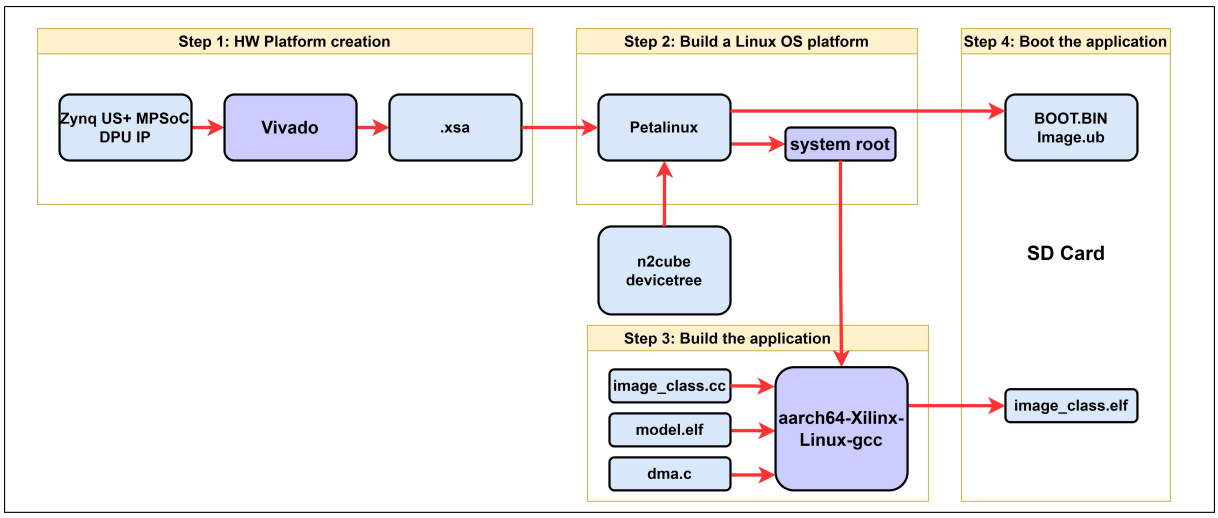


Figure 3.5 Steps to deploy the system

Deploying the system involves four steps. The first step is designing the blocks in Vivado and exporting them to a compressed directory containing the entire hardware description file (.xsa file). The second step is configuring Petalinux to run on the ZCU104 FPGA. This step also involves synthesizing files such as drivers for the DPU block (dpu.ko), the device tree for DPU, the bitstream of the design from step one, and files for the bootloader to generate the final file Boot.bin. The third step is developing the software application for DPU and exporting it to an elf file. The fourth step is loading the files exported in steps two and three onto the SD Card to program the ZCU104 FPGA through the SD Card port. Now, let's delve into the details of each step.

3.3.1 Design blocks on Vivado

The first step is to design the hardware on the PL side and export it as a .xsa file. The blocks will be designed in Vivado as depicted in Figure 3.3. However, it will only include blocks with primary functions, along with AXI Interconnect, Reset, Clock, and other necessary blocks.

Then, perform the steps of synthesis, implementation, and generation of Bitstream for the design. Afterward, export the hardware design as a .xsa file. Next, proceed with deploying Petalinux onto the designed system.

3.3.2 Build application

In this project, we utilize the GStreamer framework as a multimedia tool compatible with Petalinux to control the data acquisition process and support data storage. GStreamer includes drivers for USB, Display Port, and VCU. Hence, data acquisition from the camera and its storage in the PS's DRAM, VCU operations, and data transmission from the PS's DRAM to the Display Port are all supported by GStreamer.

Storing data after encoding to .ts files in the /home/directory can be done through Unix Shell using command-line operations.

Therefore, the application needs to be built with two processes:

- Reading the encoded .ts file in the /home/ directory, decoding it back to RGB format, and then writing it to the /dev/mem/ directory.
- Initializing DMA, setting parameters for DMA, and reading RGB image files stored in /dev/mem/ and transferring them to CNN.

Then, the application will be compiled with the aarch64-xilinx-linux-gcc toolchain to generate the executable file. Copy the executable file to the SD Card.

Finally, the files generated during the Vivado, Petalinux, and Application building processes will be copied to the SD Card as follows:

- PL bitstream: The file generated from the design circuit after running the Synthesis, Implementation, and Generate Bitstream processes.
- FSBL: The first and crucial part of the boot process into the SoC.
- U-Boot
- ATF (Arm Trusted Firmware): Provides security software for the ARMv8 architecture. This file is specific to Zynq MPSoC.

These four files are merged into the Boot.bin file and copied to the SD Card. In which, Image.ub and Boot.bin, along with the application file, are loaded into the boot area of the SD Card, while Rootfs is loaded into the root area of the SD Card.

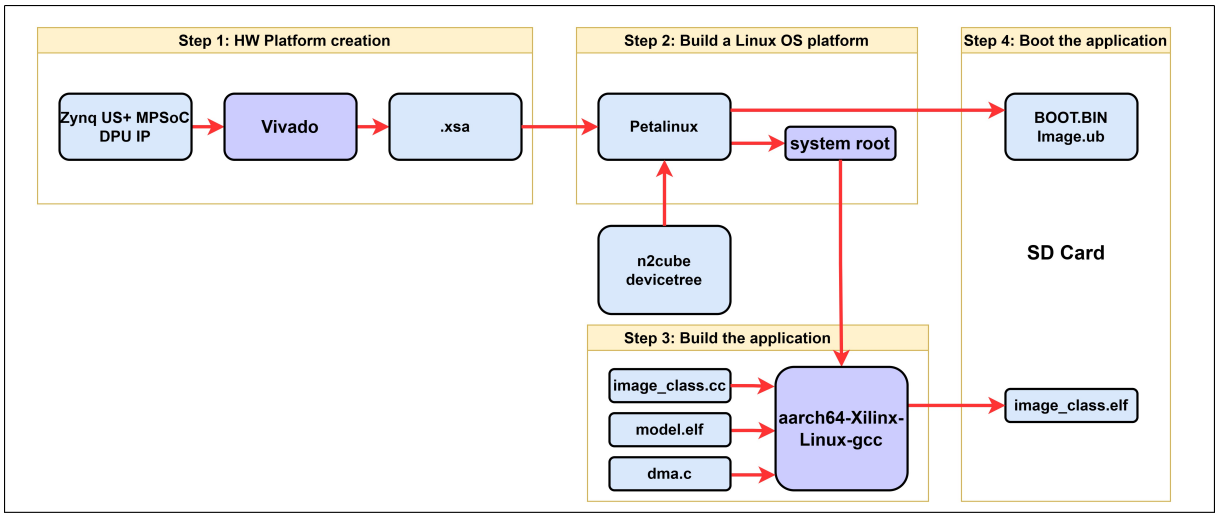


Figure 3.6 Steps to deploy the system

CHAPTER 4. RESULTS

4.1 Python Results

Firstly, we evaluated our model in Python, assessing the model based on MSE loss, PSNR, and SSIM. The results obtained from training the model with the MNIST dataset for 100 epochs were MSE loss = 0.008, SSIM = 0.889, and PSNR = 69.30. The simulation results through python are depicted in the figure 4.1

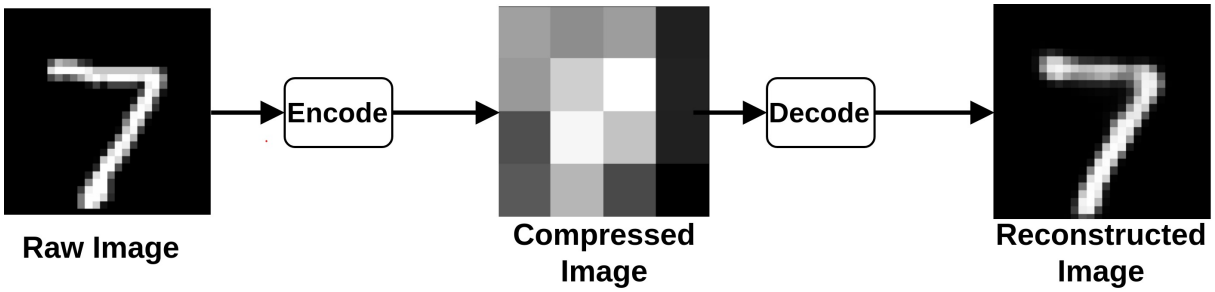


Figure 4.1 Auto Encoder result

Our input images have a size of 28×28. After passing through the encoder, the image size reduces to 128, which is more than 6 times smaller than the original image. Upon going through the decoder, the original 28×28 image is reconstructed with high quality.

After validating the model using Python, the model parameters, including weights and biases, are exported for loading into Vitis HLS. Subsequently, the model is tested using C++ to verify the results.

4.2 Verification

After completing the coding and simulation process, we proceeded to synthesize the source code to evaluate the performance and resource utilization of the circuit. We divided the process into two main parts: Encoder and Decoder, making it convenient to track and determine the specific performance of each part.

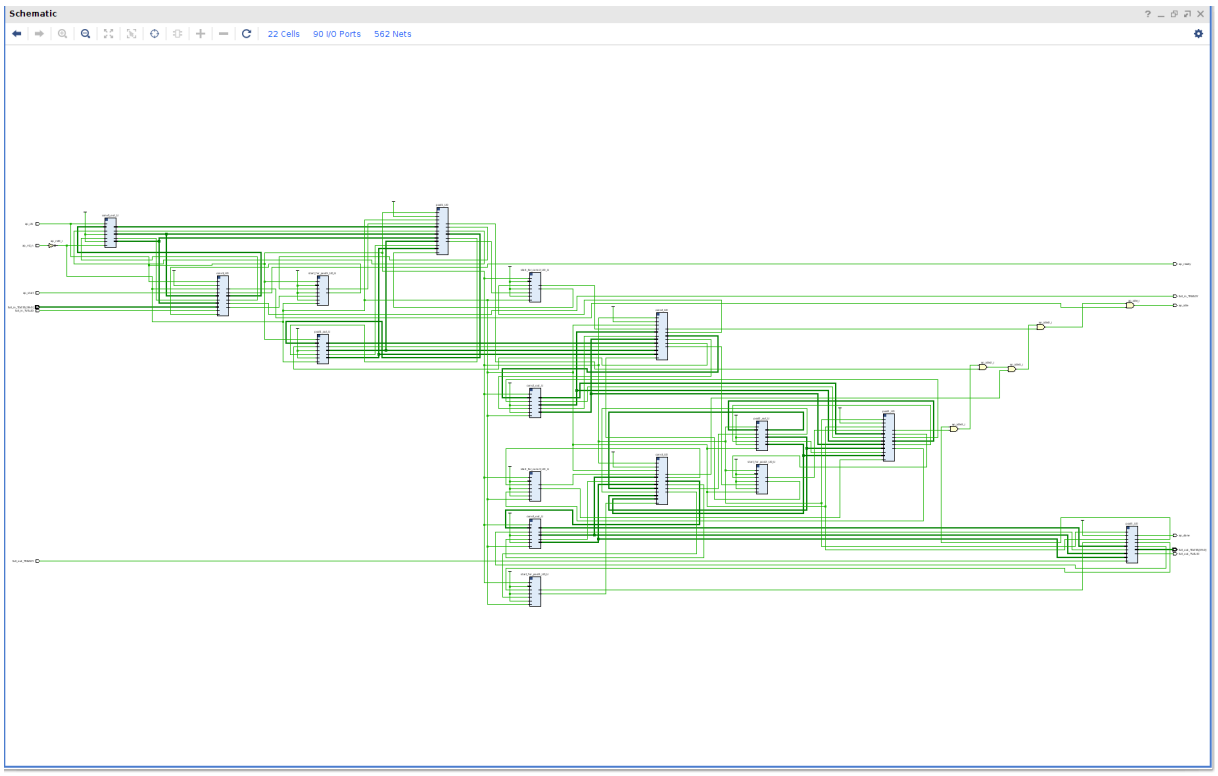


Figure 4.2 Auto Encoder Block

4.2.1 Encoder

4.2.1.1 Results after Synthesis

After synthesizing the Encoder circuit in Vivado, we obtained the following results: Encoder Part Latency

+ Latency:
    * Summary:

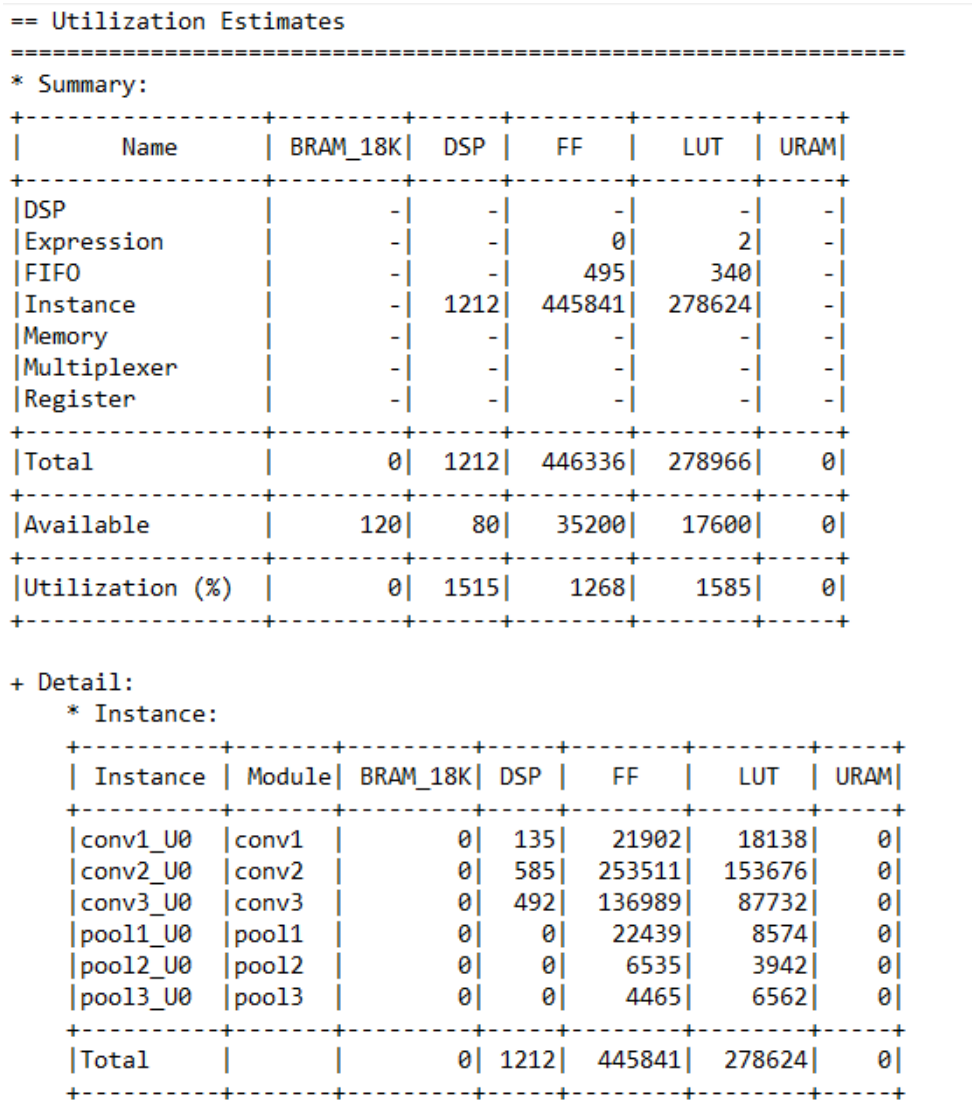
Latency (cycles)		Latency (absolute)		Interval		Pipeline
min	max	min	max	min	max	Type
14685	14685	0.147 ms	0.147 ms	14413	14413	dataflow

    + Detail:
        * Instance:

Instance	Module	Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
conv1_U0	conv1	14412	14412	0.144 ms	0.144 ms	14412	14412	no
pool1_U0	pool1	12548	12548	0.125 ms	0.125 ms	12548	12548	no
conv2_U0	conv2	4235	4235	42.350 us	42.350 us	4235	4235	no
pool2_U0	pool2	1572	1572	15.720 us	15.720 us	1572	1572	no
conv3_U0	conv3	723	723	7.230 us	7.230 us	723	723	no
pool3_U0	pool3	517	517	5.170 us	5.170 us	517	517	no

Figure 4.3 Encoder Part Latency

We measured and verified the processing time parameters of the Encoder circuit, including both overall time and time for each instance block. DSP, FF, LUT, and DRAM Utilization Parameters in the Encoder Block



```
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+------+--------+--------+-----+
|       Name      | BRAM_18K|  DSP |   FF   |   LUT  | URAM|
+-----------------+---------+------+--------+--------+-----+
|DSP              |        -|     -|       -|       -|    -|
|Expression       |        -|     -|       0|       2|    -|
|FIFO             |        -|     -|     495|     340|    -|
|Instance         |        -|  1212|  445841|  278624|    -|
|Memory           |        -|     -|       -|       -|    -|
|Multiplexer      |        -|     -|       -|       -|    -|
|Register         |        -|     -|       -|       -|    -|
+-----------------+---------+------+--------+--------+-----+
|Total            |        0|  1212|  446336|  278966|    0|
+-----------------+---------+------+--------+--------+-----+
|Available        |      120|    80|   35200|   17600|    0|
+-----------------+---------+------+--------+--------+-----+
|Utilization (%)  |        0|  1515|    1268|    1585|    0|
+-----------------+---------+------+--------+--------+-----+

+ Detail:
    * Instance:
    +----------+-------+---------+-----+--------+--------+-----+
    | Instance | Module| BRAM_18K| DSP |   FF   |   LUT  | URAM|
    +----------+-------+---------+-----+--------+--------+-----+
    |conv1_U0  |conv1  |        0|  135|   21902|   18138|    0|
    |conv2_U0  |conv2  |        0|  585|  253511|  153676|    0|
    |conv3_U0  |conv3  |        0|  492|  136989|   87732|    0|
    |pool1_U0  |pool1  |        0|    0|   22439|    8574|    0|
    |pool2_U0  |pool2  |        0|    0|    6535|    3942|    0|
    |pool3_U0  |pool3  |        0|    0|    4465|    6562|    0|
    +----------+-------+---------+-----+--------+--------+-----+
    |Total     |       |        0| 1212|  445841|  278624|    0|
    +----------+-------+---------+-----+--------+--------+-----+
```

Figure 4.4 Number of DSP, FF, LUT and DRAM in the Encoder Block

We assessed and recorded the usage of resources, including DSP (Digital Signal Processor), Flip-Flops (FF), Look-Up Tables (LUT), and DRAM memory. This ensures optimal resource utilization.

4.2.1.2 Generated Circuit and Main Blocks

After synthesis, we integrated Verilog files into Vivado to create the circuit. Below is the generated circuit and a description of the main blocks:

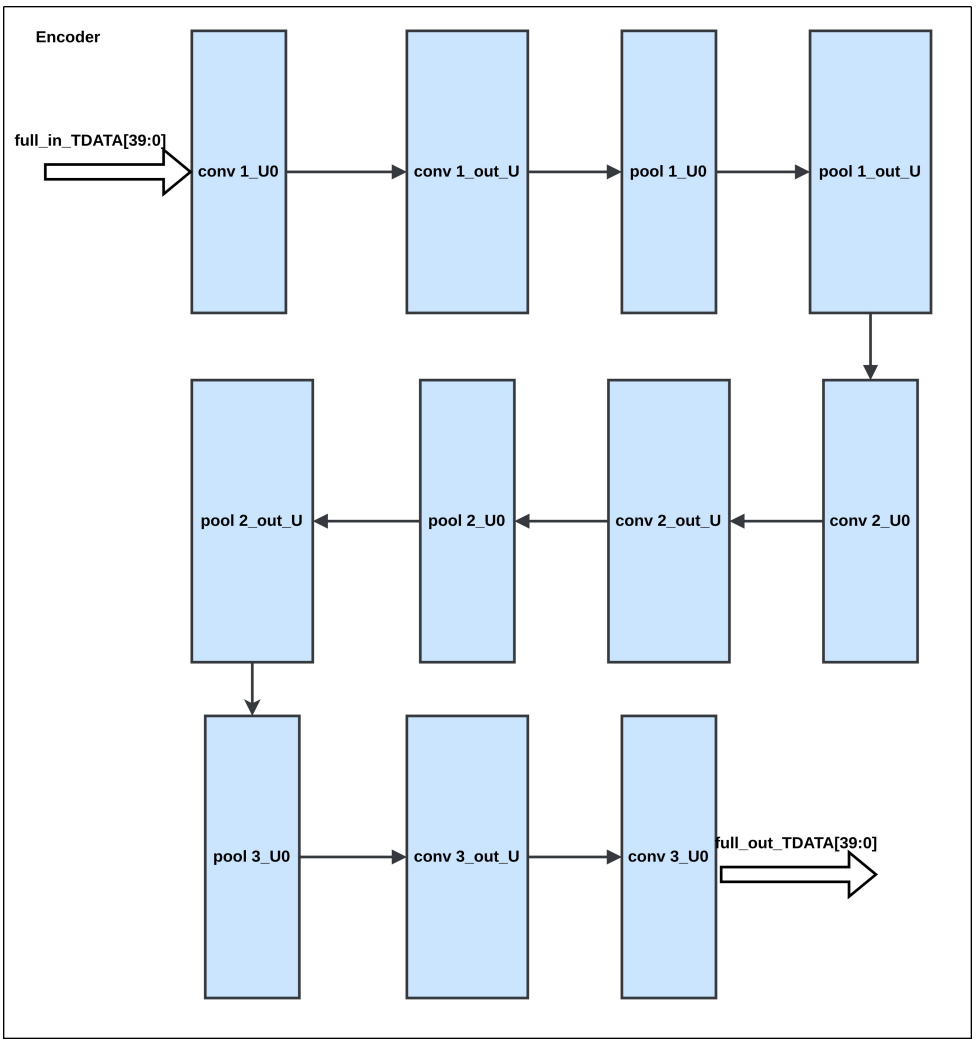


Figure 4.5 Main encoder block

Generated Circuit: Overview of the overall structure of the circuit after synthesis. Design Blocks: We listed and described the main blocks of the Encoder circuit, including logic components and signal processing units. Specifically, key blocks such as Multiplication Block, Pipeline Block, and... are presented, explaining their interactions within the circuit.

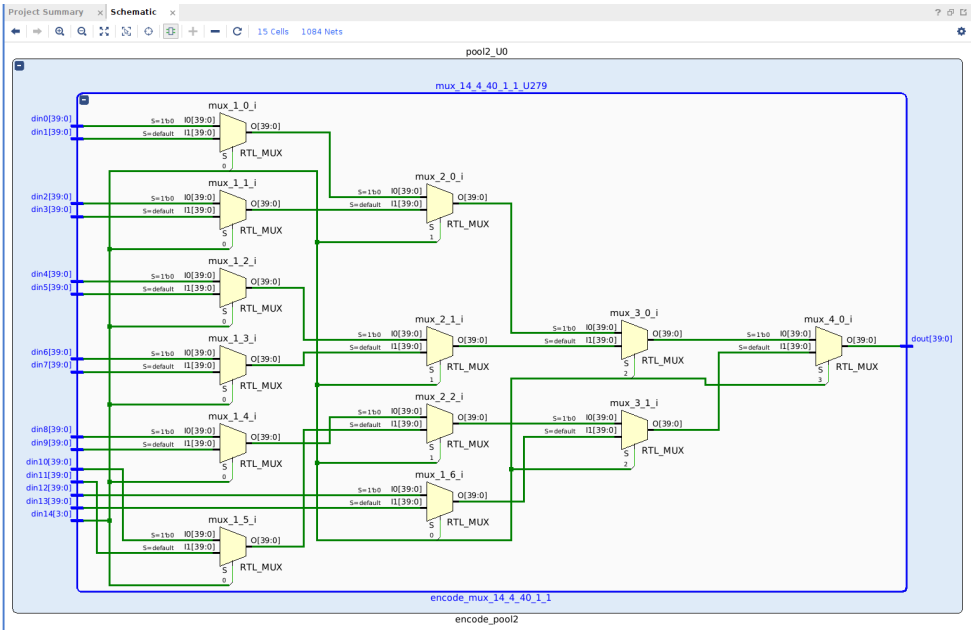


Figure 4.6 Multiplexer Block

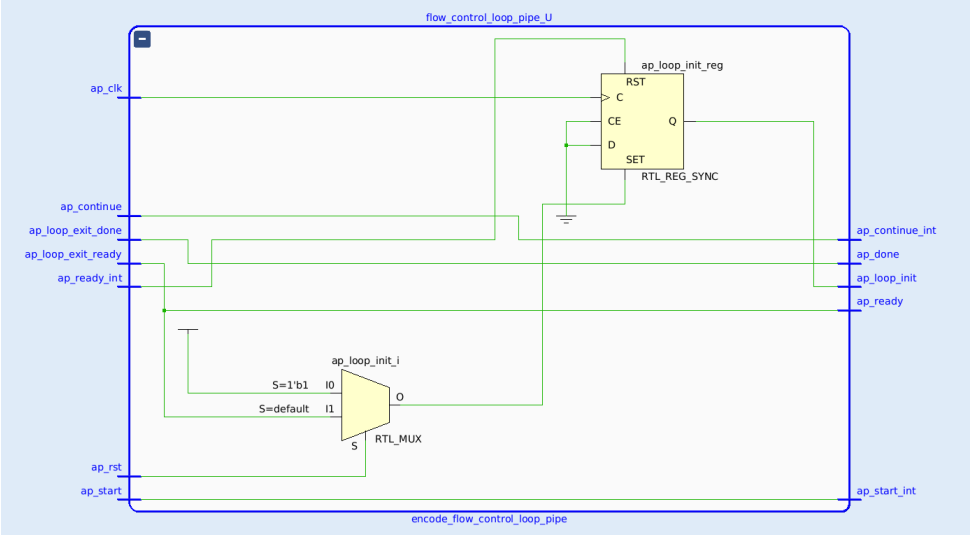


Figure 4.7 Loop pipe Control

4.2.1.3 Encoding simulation results

Similar to the decoding process, we conducted a thorough analysis of the encoding results using

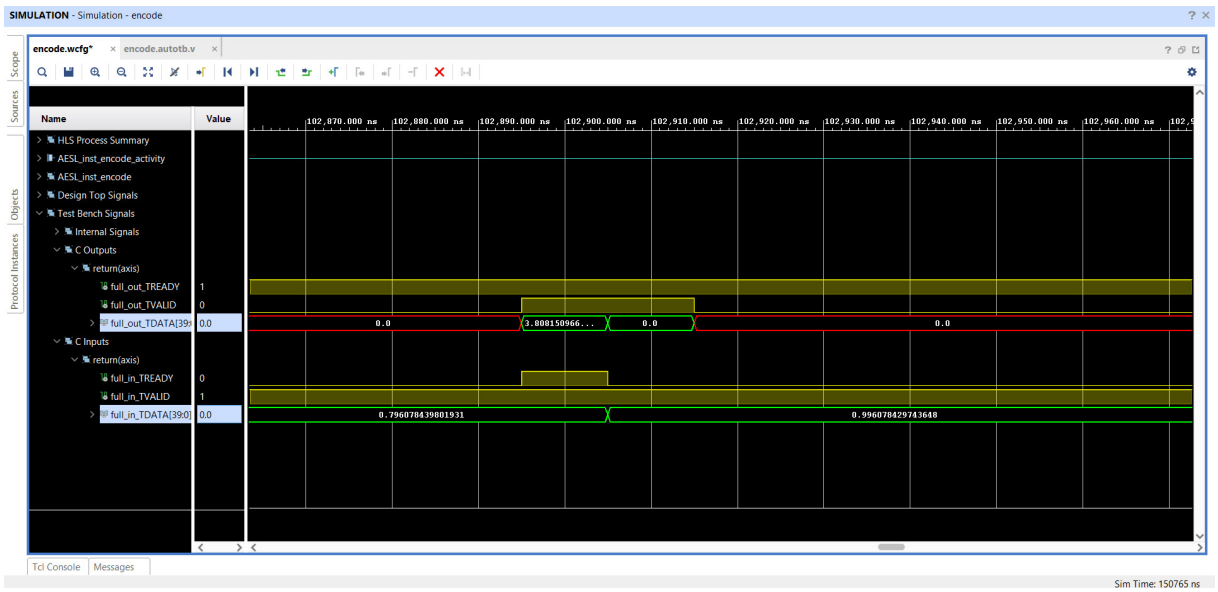


Figure 4.8 Encoder waveform

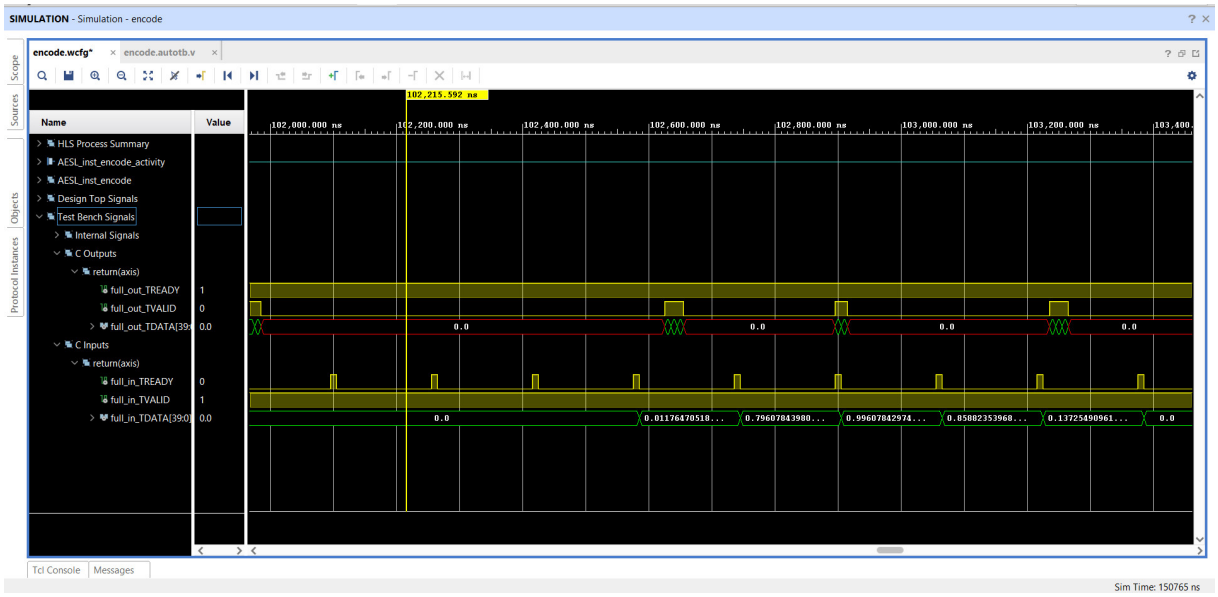


Figure 4.9 Encoder waveform

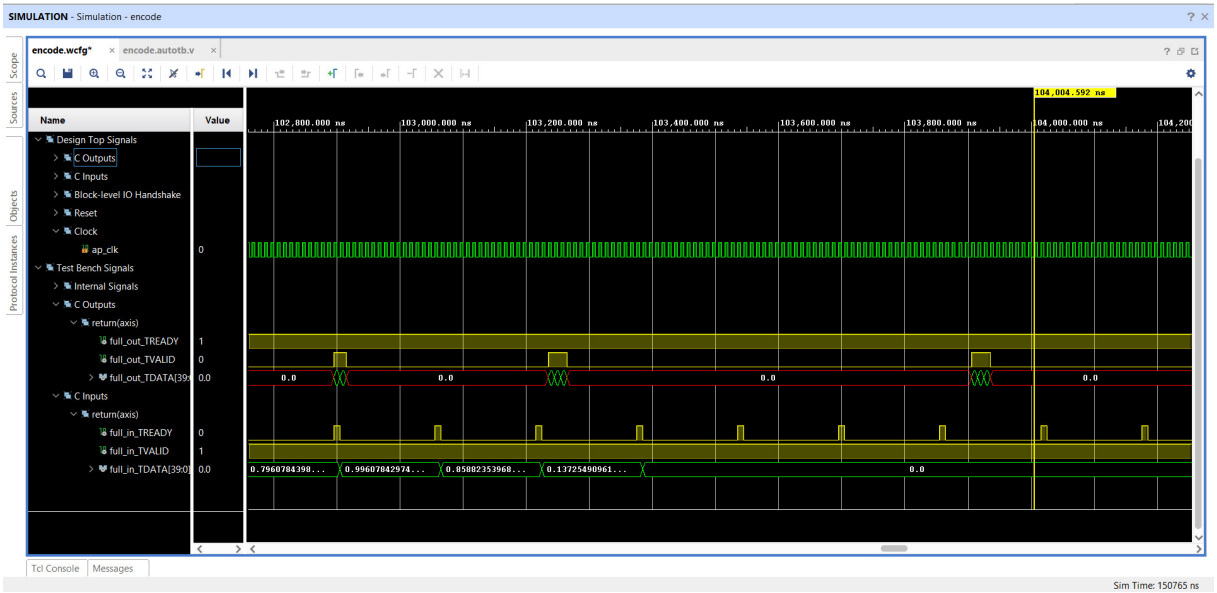


Figure 4.10 Encoder waveform

4.2.2 Decoder

4.2.2.1 Results after Synthesis

Similar to the Encoder, we obtained results after synthesizing the Decoder circuit, including: DSP, FF, LUT, and DRAM Utilization Parameters in the Decoder Block:

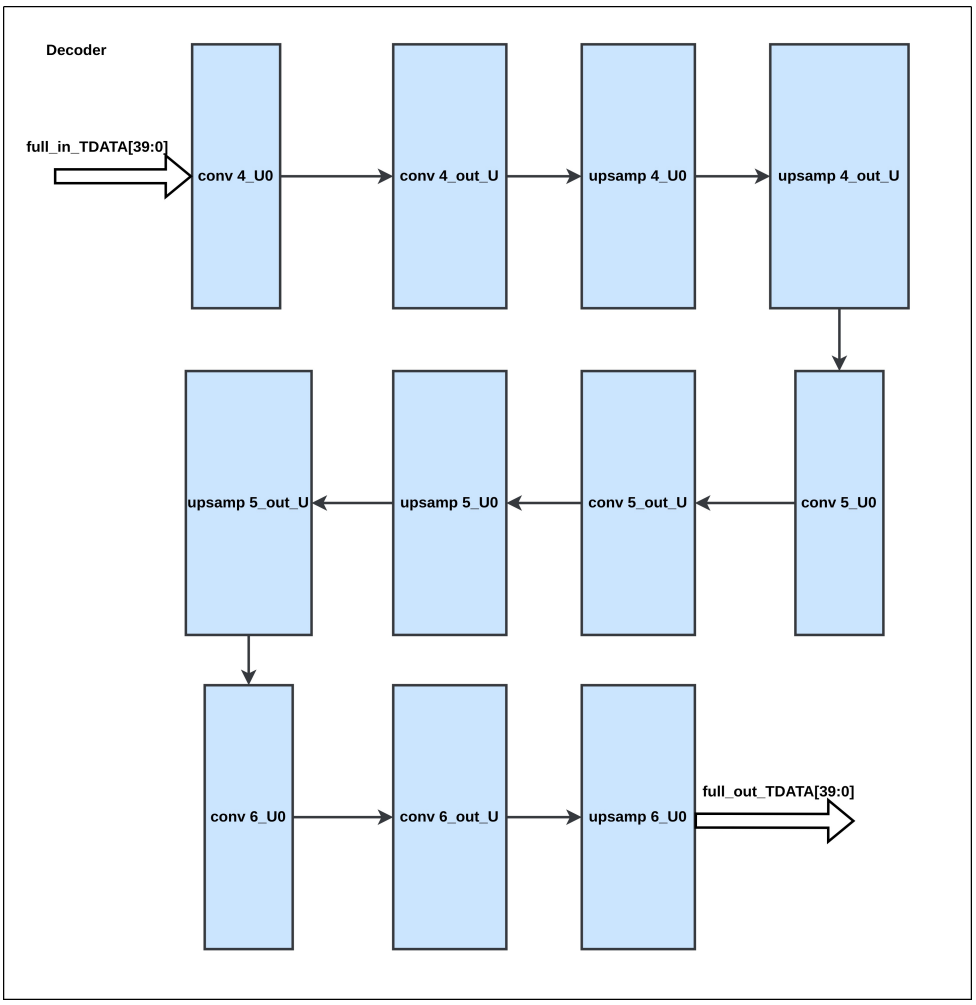


Figure 4.11 Main decoder block

== Utilization Estimates

* Summary:

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	2	-
FIFO	-	-	594	408	-
Instance	9	1695	574162	342711	0
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	9	1695	574756	343121	0
Available	120	80	35200	17600	0
Utilization (%)	7	2118	1632	1949	0

+ Detail:

    * Instance:

Instance	Module	BRAM_18K	DSP	FF	LUT	URAM
conv4_U0	conv4	0	467	132160	85354	0
conv5_U0	conv5	0	498	138623	88939	0
conv6_U0	conv6	0	586	221464	133764	0
conv7_U0	conv7	0	144	77193	32759	0
upsamp4_U0	upsamp4	3	0	1102	496	0
upsamp5_U0	upsamp5	3	0	1130	497	0
upsamp6_U0	upsamp6	3	0	2490	902	0

Figure 4.12 DSP, FF, LUT, and DRAM in Decoder Block

We continued evaluating and documenting the resource usage in the Decoder block, ensuring optimal resource utilization. We measured and verified the processing time parameters of the Decoder circuit.

+ Latency:

    * Summary:

Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
14928	14928	0.149 ms	0.149 ms	14619	14619	dataflow

    + Detail:

        * Instance:

Instance	Module	Latency (cycles) min	Latency (cycles) max	Latency (absolute) min	Latency (absolute) max	Interval min	Interval max	Pipeline Type
conv4_U0	conv4	362	362	3.620 us	3.620 us	362	362	no
upsamp4_U0	upsamp4	521	521	5.210 us	5.210 us	521	521	no
conv5_U0	conv5	874	874	8.740 us	8.740 us	874	874	no
upsamp5_U0	upsamp5	2057	2057	20.570 us	20.570 us	2057	2057	no
conv6_U0	conv6	4171	4171	41.710 us	41.710 us	4171	4171	no
upsamp6_U0	upsamp6	12561	12561	0.126 ms	0.126 ms	12561	12561	no
conv7_U0	conv7	14618	14618	0.146 ms	0.146 ms	14618	14618	no

Figure 4.13 Decoder Part Latency

4.2.2.2 *Main Blocks of the Decoder Circuit*

We listed and described the main blocks in the Decoder circuit, providing a clear understanding of the structure and interactions between components.

This information offers a detailed and comprehensive view of the performance, resource utilization, and structure of both the Encoder and Decoder after the synthesis and simulation process. This data serves as a foundation for further optimization and adjustments to the circuit if necessary.

4.2.2.3 *Decoding Results*

We created waveforms to visualize the correspondence between the input and output data during the decoding process. The following aspects were observed and analyzed. We compared the input data with the corresponding output data, ensuring that the decoding process produced the expected results.

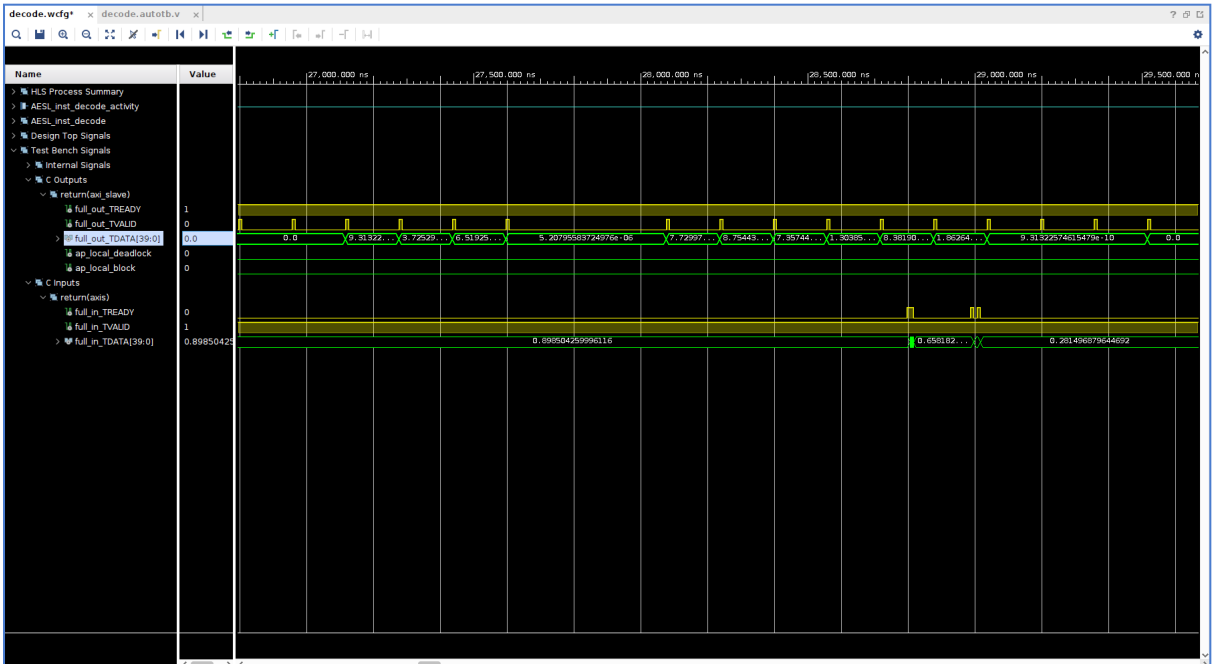


Figure 4.14 Input Data vs. Output Data

Examining the data in relation to clock signals, we assessed the timing and transitions of the data during the decoding operation.

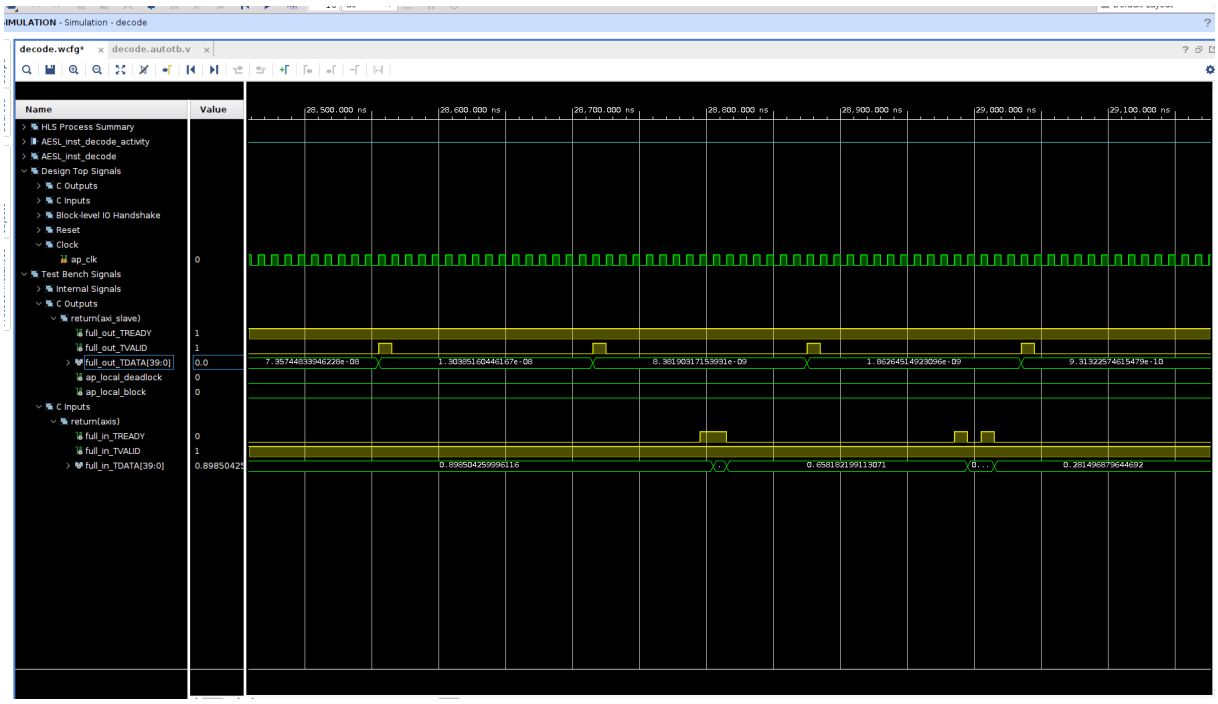


Figure 4.15 Clock Signal and Data Transition

Examining the data in relation to clock signals, we assessed the timing and transitions of the data during the decoding operation.

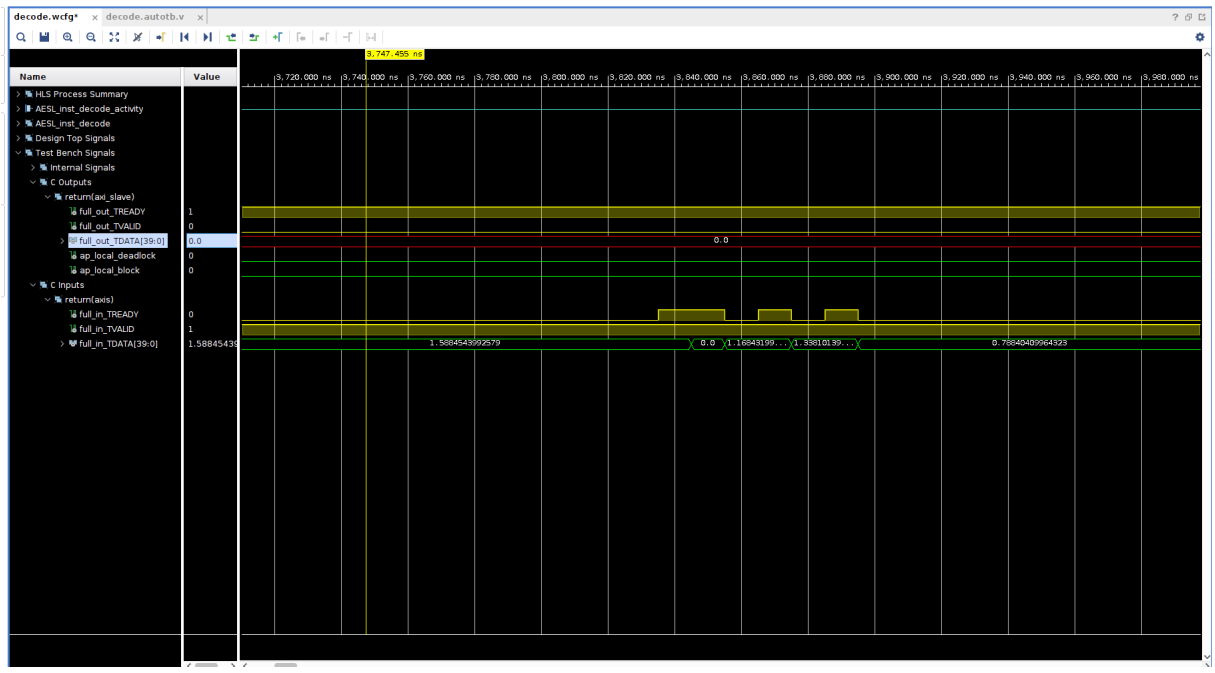


Figure 4.16 Stored data

And there are input values that are only stored in the buffer and window without any output.

4.2.3 AutoEncoder

4.2.3.1 Results after synthesis

Similar to encoding and decoding results, I also have synthesis table 4.17

```
================================================================
== Utilization Estimates
================================================================
* Summary:
+-----------------+---------+------+--------+--------+-----+
|       Name      | BRAM_18K|  DSP |   FF   |   LUT  | URAM|
+-----------------+---------+------+--------+--------+-----+
|DSP              |        -|     -|       -|       -|    -|
|Expression       |        -|     -|       0|       2|    -|
|FIFO             |        -|     -|    1386|     938|    -|
|Instance         |        4|  4328|  169156|  110468|    0|
|Memory           |        -|     -|       -|       -|    -|
|Multiplexer      |        -|     -|       -|       -|    -|
|Register         |        -|     -|       -|       -|    -|
+-----------------+---------+------+--------+--------+-----+
|Total            |        4|  1611|  170542|  111408|    0|
+-----------------+---------+------+--------+--------+-----+
|Available        |      624|  1728|  460800|  230400|   96|
+-----------------+---------+------+--------+--------+-----+
|Utilization (%)  |       ~0|    93|      37|      48|    0|
+-----------------+---------+------+--------+--------+-----+
```

Figure 4.17 Utilization Estimates

In the summary figure 4.17, I change bit length representation to 24 bits to fit with the capabilities of ZCU104. So that, you can see the number of DPS, FF, LUT doesn't not surpass these resources available in ZCU104.

```
============================================================
* Summary:
+--------------------+-----+-----+------------+----------------------+--------------+
|      RTL Ports     | Dir | Bits|  Protocol  |     Source Object    |    C Type    |
+--------------------+-----+-----+------------+----------------------+--------------+
|full_in_AXI_TDATA   |   in|   24|        axis|   full_in_AXI_V_data_V|       pointer|
|full_in_AXI_TKEEP   |   in|    3|        axis|   full_in_AXI_V_keep_V|       pointer|
|full_in_AXI_TSTRB   |   in|    3|        axis|   full_in_AXI_V_strb_V|       pointer|
|full_in_AXI_TUSER   |   in|    2|        axis|   full_in_AXI_V_user_V|       pointer|
|full_in_AXI_TLAST   |   in|    1|        axis|   full_in_AXI_V_last_V|       pointer|
|full_in_AXI_TID     |   in|    5|        axis|     full_in_AXI_V_id_V|       pointer|
|full_in_AXI_TDEST   |   in|    6|        axis|   full_in_AXI_V_dest_V|       pointer|
|full_in_AXI_TVALID  |   in|    1|        axis|   full_in_AXI_V_dest_V|       pointer|
|full_in_AXI_TREADY  |  out|    1|        axis|   full_in_AXI_V_dest_V|       pointer|
|full_out_AXI_TDATA  |  out|   24|        axis|  full_out_AXI_V_data_V|       pointer|
|full_out_AXI_TKEEP  |  out|    3|        axis|  full_out_AXI_V_keep_V|       pointer|
|full_out_AXI_TSTRB  |  out|    3|        axis|  full_out_AXI_V_strb_V|       pointer|
|full_out_AXI_TUSER  |  out|    2|        axis|  full_out_AXI_V_user_V|       pointer|
|full_out_AXI_TLAST  |  out|    1|        axis|  full_out_AXI_V_last_V|       pointer|
|full_out_AXI_TID    |  out|    5|        axis|    full_out_AXI_V_id_V|       pointer|
|full_out_AXI_TDEST  |  out|    6|        axis|  full_out_AXI_V_dest_V|       pointer|
|full_out_AXI_TVALID |  out|    1|        axis|  full_out_AXI_V_dest_V|       pointer|
|full_out_AXI_TREADY |   in|    1|        axis|  full_out_AXI_V_dest_V|       pointer|
|ap_clk              |   in|    1|  ap_ctrl_hs|           AutoEncoder|  return value|
|ap_rst_n            |   in|    1|  ap_ctrl_hs|           AutoEncoder|  return value|
|ap_start            |   in|    1|  ap_ctrl_hs|           AutoEncoder|  return value|
|ap_done             |  out|    1|  ap_ctrl_hs|           AutoEncoder|  return value|
|ap_ready            |  out|    1|  ap_ctrl_hs|           AutoEncoder|  return value|
|ap_idle             |  out|    1|  ap_ctrl_hs|           AutoEncoder|  return value|
+--------------------+-----+-----+------------+----------------------+--------------+
```

Figure 4.18 Interfaces in CNN Block

Figure 4.18 summary the signal port in our system. There are AXIS protocol used for input, output data and some control port. So to connect with other block, we need to setup axis protocol for transfer data.

4.2.3.2 Simulation Results

We also export waveform of test data results for AutoEncoder network. And the results exactly the same as the data I have trained in Python.

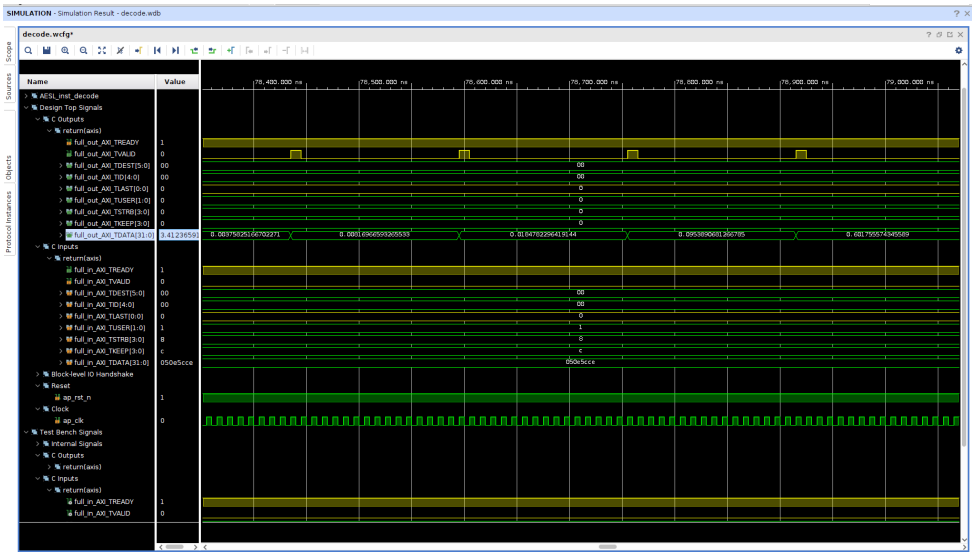


Figure 4.19 Wave form for AutoEncoder

4.2.3.3 Register Transfer Level generate

Figure 4.20 illustrate AutoEncoder IP Block we generate after co-simulation.

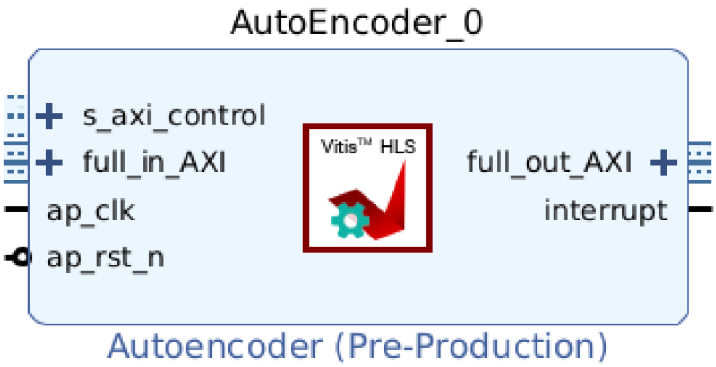


Figure 4.20 AutoEncoder IP Block

4.3 Compare with python

Next, we will compare the simulation results using Vitis HLS and Python based on two key metrics, MSE and PSNR.

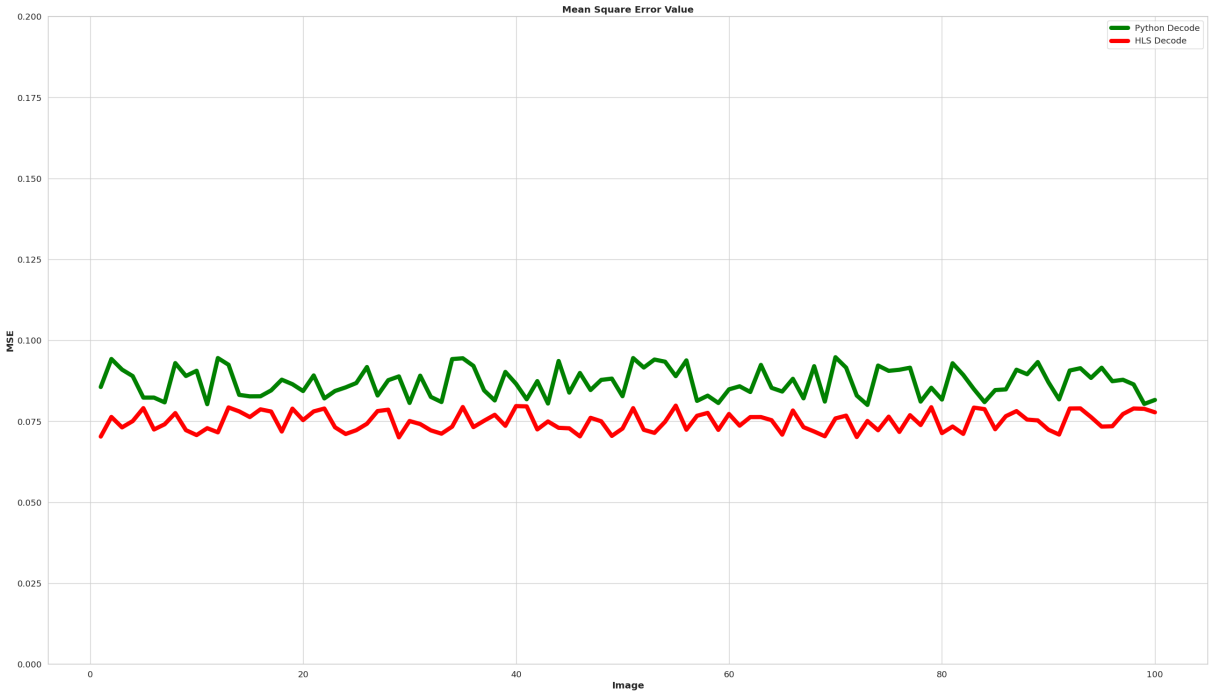


Figure 4.21 Stored data

Figure 4.21 illustrate the simulation results of applying the autoencoder to 100 test images using Python and HLS. The results indicate that the MSE between Python and HLS is significantly negligible.

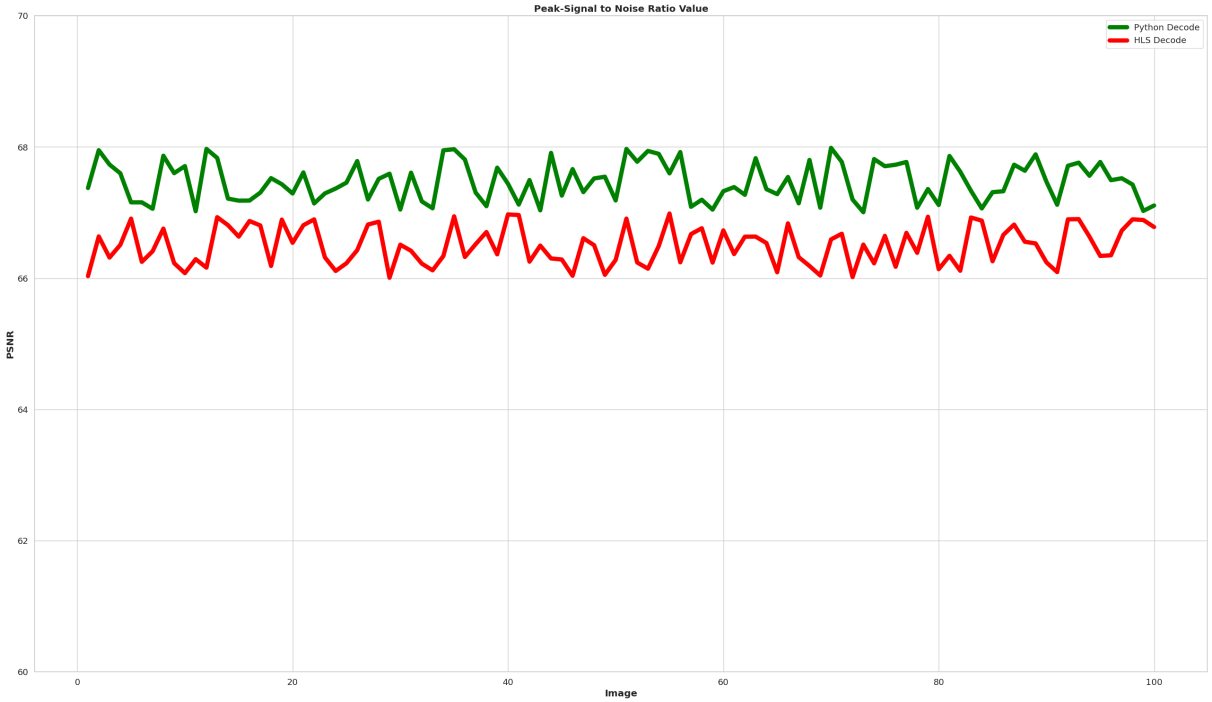


Figure 4.22 Stored data

Figure 4.22 depicts the PSNR results of 100 images using Python and 100 images

using HLS. The results also demonstrate the similarity between the Python model and the HLS simulation. Importantly, our model utilizes fixed-point representation and has been tested on ZCU104.

4.4 Implementaion on ZCU104

After create SoC Block of the all system, we synthesis generate bitstream and implement on ZCU104 through Vivado.

We have already installed Petalinux in host PC to control all the process. However I still get a trouble in resize data, because my data have the size 28×28 while the input originate data get from the camera have the size 1920×1080. Therefore, they do not share the same ratio, leading to our inability to directly input data into the network. Currently, we can only transmit image data into the IP MP SoC block. Then, we save the data onto the SD card and display it on the screen. This process is illustrated in Figure 4.23.

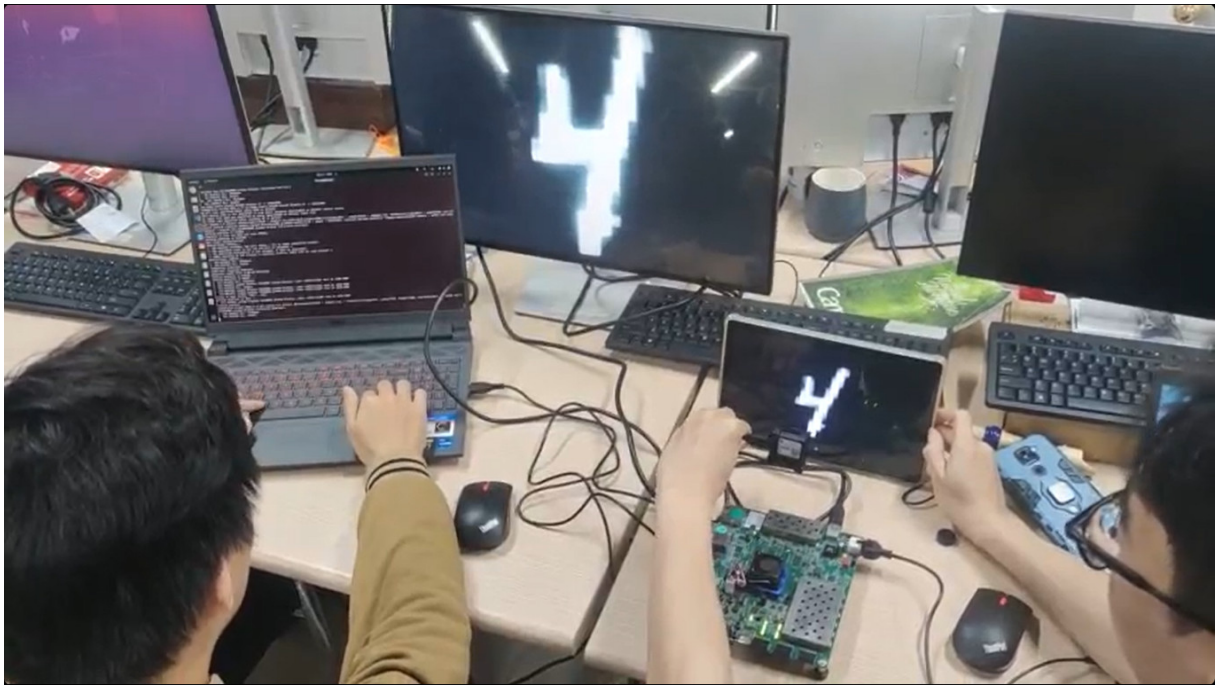


Figure 4.23 Implementation on FPGA

CHAPTER 5. CONCLUSIONS AND FUTURE DEVELOPMENT

The Autoencoder project for image compression and decompression has yielded positive results, showcasing the power of CNN technology in optimizing the representation and restoration of visual information. The Autoencoder model not only effectively compresses image data but also retains crucial features, reducing storage space and enhancing the efficiency of image transmission and processing.

Furthermore, the decoding capability of the model is highly impressive, accurately and flexibly reconstructing original images from compressed data. This highlights the adaptability and self-learning capacity of the model across diverse image types.

In advancing the project, our focus is on optimizing the model, expanding its applications into various domains, integrating with other models such as GANs, incorporating it into real-world applications, handling large datasets, and enhancing security measures. These progressive steps aim to transform the Autoencoder model into a versatile and powerful tool applicable across a wide range of real-world scenarios

REFERENCES

[1] Aydin, Seda Guzel and Bilge, Hasan Şakir, "FPGA Implementation of Image Registration Using Accelerated CNN", journal: Sensor

Reconfigureable Autoencoder based on Reduced Instruction Architecture

Nicholas Teffandi
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
13221084@mahasiswa.itb.ac.id

Fauzan Ibrahim
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
13221030@mahasiswa.itb.ac.id

Achmad Novel
School of Electrical Engineering and Informatics
Institut Teknologi Bandung
Bandung, Indonesia
13221057@mahasiswa.itb.ac.id

***Abstract*—This paper will discuss a proposed design of reconfigurable reduced instruction architecture for an Autoencoder. The proposed design consists of several main components starting from the Control Unit, Arithmetic Logic Unit, Memory block, Instruction memory, and LUT for activation function. The proposed design contains a prewritten special instructions set that are customized to run autoencoder algorithm, where both instruction and data are represented in 16-bit format. The instruction contains 4 fields with 4 bit width each, which are operation code, source address 1, source address 2, and destination address. On the other hand, the data are represented in two's complement fixed point representation format, where both exponent and fraction took 8 bits. The proposed design is verified based on 3 different cases, a 3x3 circle, cross, and greyscale circle, all of which show that the proposed design can learn input information and can reconstruct it with minimal error. The verification is done with the assumption the data is normalized/denormalized outside of the architecture, and all of the weight and bias is initialized with the value of 0.5 with a learning rate of 0.5. The design then implemented on PYNQ-Z1 Board where the board utilization is very small compare to the available resources. The design has maximum frequency of 150 MHz while only consuming 0.115 W of power. Those figure of merits shows that the design is able to be used in parallel to process larger images.**

***Index Terms*—Autoencoder, Instruction, Architecture, Design**

I. Introduction

In recent years, the widespread adoption of artificial intelligence (AI) and machine learning (ML) techniques has led to a growing demand for high-performance and energy-efficient hardware implementations of neural networks [1]. Autoencoders, a class of unsupervised learning models, have proven to be powerful tools for feature learning, data compression, and anomaly detection [2]. To fully unleash the potential of Autoencoders in real-world applications, it is imperative to explore hardware acceleration solutions that can meet the increasing computational demands while maintaining energy efficiency.

Traditional software implementations of Autoencoders on general-purpose processors often struggle to deliver the required throughput and low latency, especially when dealing with large-scale datasets or real-time applications [3]. As a result, there is a growing interest in the development of dedicated hardware architectures tailored for efficient execution of Autoencoders algorithms.

This paper presents a comprehensive exploration of the hardware implementation of Autoencoders using Verilog, a hardware description language widely used for digital circuit design. By translating the Autoencoders architecture into hardware, we aim to leverage parallelism and optimize the computational efficiency of the model. This hardware-centric approach is motivated by the need to accelerate Autoencoders inference tasks, making them suitable for deployment in resource-constrained environments, such as edge devices and IoT applications.

II. Proposed Design

The Autoencoder algorithm is built upon a neural network system designed to reconstruct the input observation/image with the lowest error possible. Autoencoders' main component consists of an encoder, a latent feature representation, and a decoder. The encoder and decoder can be written as a function g and f respectively that depends on some parameters, in this case, weight and bias. [4]

$$x_o = f(g(x_i))$$

By nature, that function needs to be run in sequential order between one iteration and the other. In digital circuits, there are generally two ways to approach it, one with a state machine and the other is based on instruction sets which form the foundation of modern computers.

The proposed design is built with similar characteristic of an architecture that supports a reduced ISA instruction. However, this architecture is specialized and optimized to calculate the Autoencoder Algorithm for machine learning purposes. The architecture consists of several critical component such as Instruction Memory block, program counter, Memory block, Arithmetic and Logical Unit (ALU), Control Unit (CU), Mux and Demux selector, Sector selector block, and Look Up Table (LUT) table for Sigmoid and first derivative of Sigmoid function (refer to figure 1). The architecture dataflow is handled by the Control Unit, where this module is responsible for controlling the ALU, Memory Block, and Mux and Demux to ensure correct execution of the instruction.

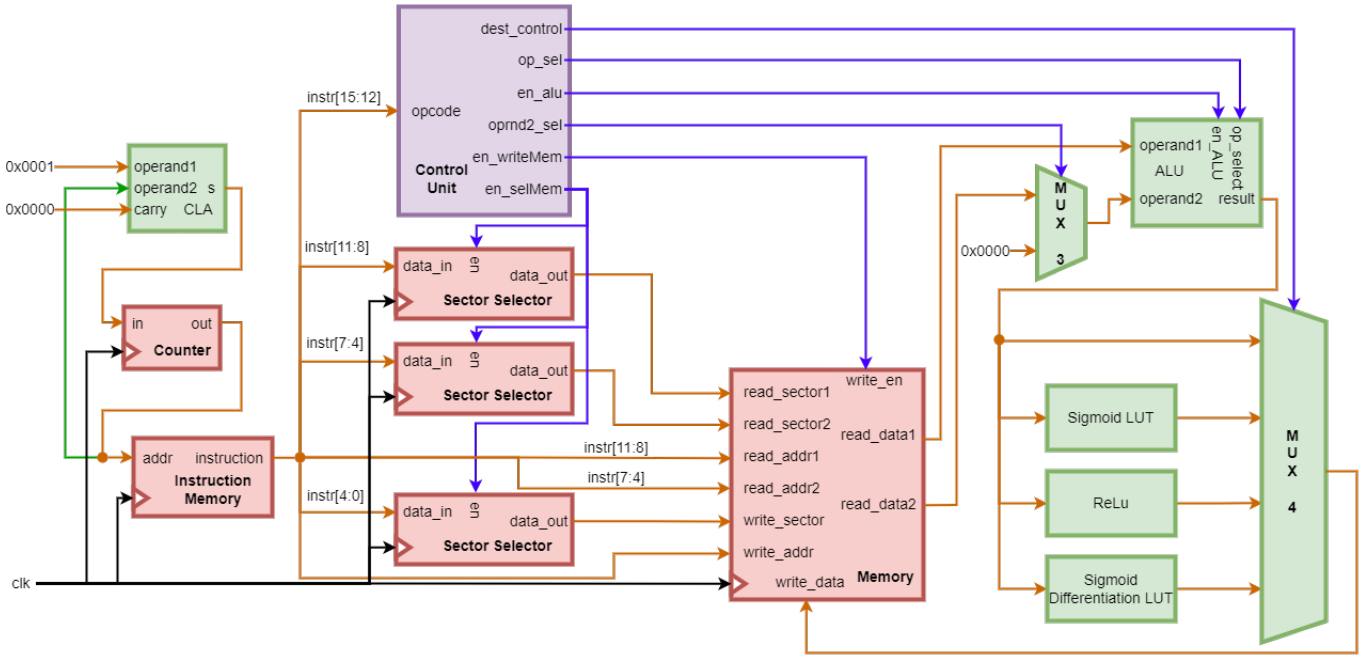


Fig. 1. Top Level Architecture

TABLE I
INSTRUCTION FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
opcode				addr1				addr2				addr3			

TABLE II
NUMBER FORMAT

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
s	exp							frac							

A. Arithmetic and Logical Unit

The ALU used in this architecture are capable of three mathematical operations, addition, subtraction and multiplication, which all are sufficient to fully run Autoencoder algorithm, the ALU block consists of a CLA (Carry Look Adder) [5] and Multiplier, the decision of using CLA as an adder is to optimize its operation speed with the cost of area. However, to calculate a logistic function such as Sigmoid and Sigmoid first derivative, a Look Up Table (LUT) is utilized for the computation to reduce execution time and complexity of the calculation.

B. ReLU (Rectified Linear Unit

ReLU is an activation function where it passes any input value to its output,only if the input is greater than zero, else it will produce zero output for negative input. This activation function is implemented with combinational gates where the MSB of the data which is the sign bit is inverted and connected with AND gate for every remaining bit of the data.

C. Lookup Table

The LUT is one of the ways to decrease computing time by saving the value of functions in a certain range. The LUT is produced by calculating the output of the function then convert the value into the binary representation used in the design, then each output is paired up with the respective input. From its inherent characteristics, LUT occupies a big space in the chip which corresponds to large resources in hardware. However, it will be very beneficial for large structures that use the same functions multiple times which is the case here in the Autoencoder Algorithm. The Sigmoid function consists of an exponential function which itself is a constant. Aside from the Sigmoid function, the algorithm also uses its differential that still has the exponential part. Therefore, using LUT for both functions will save a great deal of execution time compared with calculating it manually.

D. Memory Block

Memory block is constructed to store and load every calculation for every instruction (refer to figure 2), the memory contains a 16 address where every address can store and

load 16 bits of information, to further expand the number of addresses without compromising the format of the instruction, another block is required which is the sector selector. The idea is that every memory block with 16 addresses will generate 16 times, with each of these memory block referred as sectors, a sector selector block is used to select those sectors separately (split sector reading and writing), to change sector selection, an additional instruction is required.

E. Instruction Memory Block

The Instruction Memory block contains a prewritten instruction for the architecture to execute, the instruction is written such that it will perform mathematical operation necessary for Autoencoder algorithm. The program counter will increment the reading address of the instruction memory until one cycle of forward propagation and backward propagation is completed, then it will loop until the specified training number. Secondly, the fetched instruction contains 4 critical information, which is the operation code, source address 1, source address 2 and destination address, the details of the instruction structure will be discussed later, this information will be fed into the memory block and CU to be executed at positive edge and the resulting value will be written back into the memory block at negative edge of the clock, hence every instruction is executed in single cycle. Finally, if the instruction requires special function such as ReLu (Rectified Linear Unit), Sigmoid and Sigmoid first derivative, the mux will change the memory writeback data source to those special function block output.

F. Control Unit

Control Unit is a module responsible for controlling and managing other supporting module via control signals, it functions as a block for interpreting operational codes to signals to control the flow of data and activating functions on other blocks.

G. Instruction Format

The system uses 16-bit data width i.e 16-bit instructions and also 16-bit data. The instruction format can be seen in table I. The opcode is used to determine which signals to activate in the control unit. Then, the other 12-bit data is parsed into three sections to determine which memory location to be used in the current calculation. For data, it is only used to represent numbers with the MSB as sign, the next 7 bits (bit 8-15) as the exponential part and the 8-bit LSB as the fractional part as seen in II. The number represented by the data uses the conventional base of 2 calculation with the negative number is the two's complement of the positive representation.

The approach of the design consisted of several functions separated into 3 types of function including math function, memory access function, and LUT accessing function. These functions will be called with the opcode on the instruction, here are the operation codes for accessing these functions that are available within the design refer to table III,

TABLE III
OPERATION CODES

Opcode	Function	Explanation
0000	Addition	Adding first address with second address
0001	Subtraction	Subtracting first address with second address
0010	Multiplication	Multiplying first address with second address
0011	Memory Write	Accessing memory to write current address to memory
0100	Sector Select	Accessing MUX for selecting memory sector
0101	Sigmoid LUT	Converting operation result for correlated sigmoid Value within the LUT
0110	ReLU	Applying ReLU Function if address1 has the appropriate condition for RELU
0111	Sigmoid Differential LUT	Converting operation result for correlated sigmoid differential Value within the LUT
1111	NOP	Do nothing

With these opcodes the design can be easily manipulated into other functionality by altering the operations within instruction memory.

The design of this system provides a step-by-step process for implementing math functions in neural network design. To begin, the instruction will select the appropriate memory sector for the first sector input, addr1, followed by addr2 as the second sector input, and then record the result sector in addr3, refer to table I for the instruction format. The instruction will then guide the Control Unit (CU) to enable the memory sector selector while disabling the writing function to ensure that the selected memory sector is not altered. Once the sector is selected, the instruction can perform the desired function with addr1 and addr2 as inputs by identifying the address of the variable within the selected sector and saving the result into addr3 within the sector that has been picked already.

H. Design novelty and advantages

There are several advantages to the proposed design, firstly, the design is done with consideration of minimal hardware cost while maintaining its speed. For every cycle instruction, there are no idling architecture component, which implies that there are no computing resources wasted. Secondly, the architecture is expandable, meaning that the number of pixel increase can be easily accommodated by generating more of the same architecture as the basic architecture are able to handle 3x3 pixel images, albeit it needs to be modified to optimize it, i.e. since all architecture run by the same control unit and instruction, the top level can utilize a data bus while the remaining architecture shared a common control unit and memory instruction. Furthermore, the design proposed employed a LUT for a logistic function dan its first derivatives, which are faster and easily reconfigurable and opens the possibility of sharing the same LUT for multiple basic architecture, hence optimizing its cost effectiveness for the total architecture. Finally, due to nature of the architecture, changes in instruction can be easily done, therefore offering flexibility in the mathematical operation of the Autoencoder.

III. DESIGN VERIFICATION

The verification is done with setup and flow illustrated in Figure 5, where the autoencoder are compared based on 3

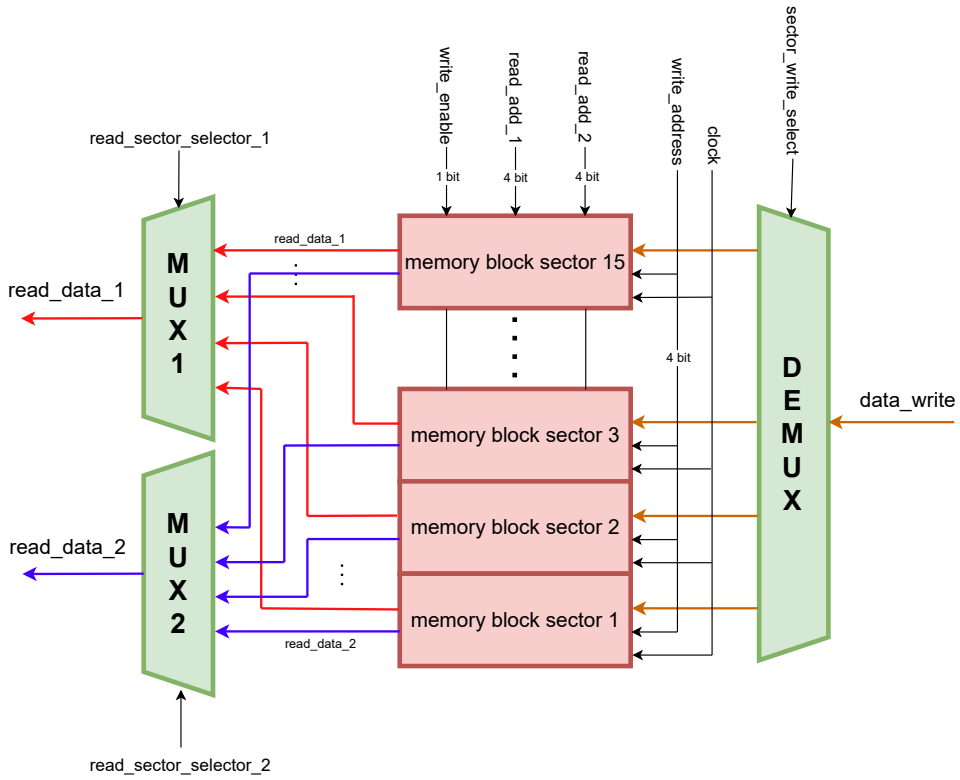


Fig. 2. Memory block architecture

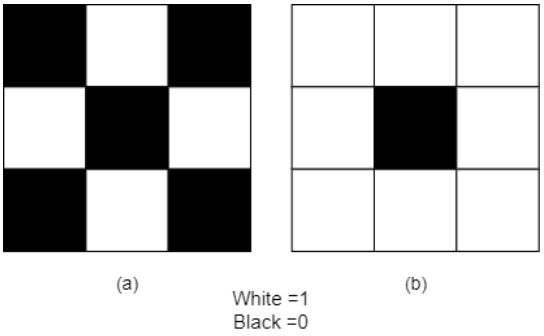


Fig. 3. Image Verification, (a) Cross (b) Circle

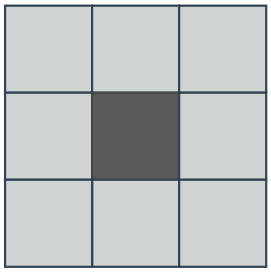


Fig. 4. Image Verification, (c) grayscale circle

cases, circle, cross and greyscale version of the circle. Note that all the weight and bias are initialized with value of 0.5, and it is assumed that the data input pixel are normalized, and the output of the system have not been de-normalized. The process of normalization and denormalization are assumed to be located outside of the proposed design.

Refer to Table V and IV, where the second column multi rows denote pixel output (ascending from top down), the verification is done with python to observe the difference between the proposed design with the python version of the algorithm. The verification is done with training number of 10000 training, where 3 random iteration is chosen and compared. For the cross cases, the observation is done at 1, 20, and 10000 iteration, where if it is directly compared with the python cases, it is evident that there are small difference between python generated value with its counterpart, this is due to the limited precision with the proposed design where it is limited to 8 bits of fraction, hence resulting in slight deviation from python result which is ran on 64 bit machine. On the other side, the circle case shows similar error with the observation conducted at first, second and last iteration. In summary, the proposed design has been successfully implemented into HDL code where the design is able to adapt to its input pixels, for more details on the simulation waveform, please refer to Appendix A for both cases.

To further benchmark its performance another test is conducted where the pixel input is modified into a grayscale. For the test, a greyscale circle is selected as input, where the ring outside of the circle is set as 0.75 (normalized) and the centre is 0.25 (normalized). Notice that on table VI, where

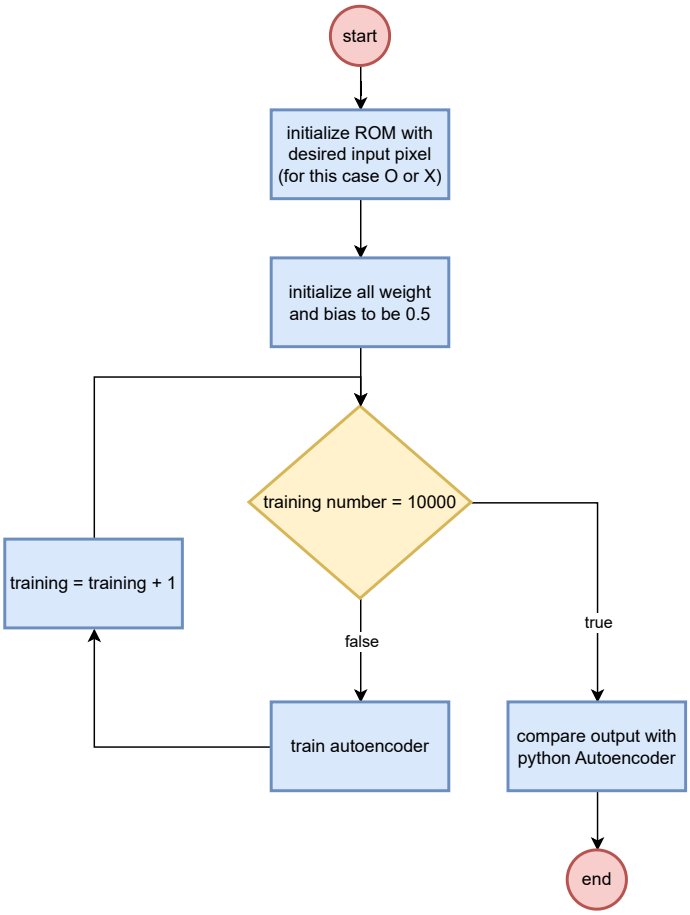


Fig. 5. Verification Scheme

at the third iteration there are significant deviation between python model and the proposed design, however as the training number increased, the gap between the model has shrunk significantly. Hence proving that the autoencoder can adapt to grayscale input. For more detail, please refer to Appendix for the simulation wave.

IV. Implementation Result

The proposed design is implemented into PYNQ-Z1 Evaluation Board via Vivado Design Suites, where the metrics such as circuit domain which is categorized into LUT, LUTRAM, FF, DSP the performance metrics which composed of several crucial parameters (maximum clock frequency allowed, the power consumption, the latency and the critical path speed) can be found at table VII. The performance metrics are obtained by trial and error of the Design Rules Constraint of the clock period input period, which need to comply with its total propagation delay from the proposed design, which is 6.50 ns.

In Table VII, it is evident that the presented design harnesses a fraction of the board's maximum capacity. This observation highlights the design's modularity, allowing for seamless integration with identical instances to collectively manage significantly larger pixel loads. This highlights a key strength of the proposed design, showcasing its scalability through parallel processing capabilities. The ability to combine multiple instances of the proposed design enables the concurrent processing of larger-sized images, exemplifying the efficiency and adaptability of the proposed system for handling substantial computational workloads. This modular approach not only optimizes resource utilization but also positions the design as a versatile solution for image processing tasks of varying scales within the scope of parallel computing paradigms.

Based on the performance metrics acquired, using this design in parallel with the same instances to handle a much larger image would still yield the same latency since its instantiation process has its own 3 by 3 image. However, it is important to note that the increase number of this design used would definitely increase the power draw and also the board utilization. Therefore, when applicating this design in parallel, the user has to keep in mind those limits or they can also process data in groups instead of processing it all at the same time.

V. Conclusion

In conclusion, the successful hardware implementation of Autoencoder through the utilization of an instruction-based

TABLE IV
CROSS PIXEL OUTPUT

Iteration	Python	Proposed Design
1	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
	0.9526	0.9492
20	0.0068	0.0
	0.9941	0.9961
	0.0068	0.0
	0.9941	0.9961
	0.0068	0.0
	0.9941	0.9961
	0.0068	0.0
	0.9941	0.9961
	0.0068	0.0
10000	0.0	0.0
	0.9999	1.0
	0.0	0.0
	0.9999	1.0
	0.0	0.0
	0.9999	1.0
	0.0	0.0
	0.9999	1.0
	0.0	0.0

TABLE V
CIRCLE PIXEL OUTPUT

Iteration	Python	Proposed Design
1	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
	0.9933	0.9922
2	0.9941	1.0
	0.9941	1.0
	0.9941	1.0
	0.9941	1.0
	0.0000	0.0
	0.9941	1.0
	0.9941	1.0
	0.9941	1.0
	0.9941	1.0
10000	0.9999	1.0
	0.9999	1.0
	0.9999	1.0
	0.9999	1.0
	0.0000	0.0
	0.9999	1.0
	0.9999	1.0
	0.9999	1.0
	0.9999	1.0

architecture demonstrates the efficacy of our approach. The employed architecture not only facilitates the execution of instructions but also affords modularity, enabling seamless scalability for processing larger images. The ability to employ multiple instances of the same architecture without compromising computational efficiency signifies a crucial advantage in handling larger image datasets. This not only enhances the system's adaptability to diverse image processing requirements but also underscores its potential for accommodating increased computational demands. The testing affirm the viability and versatility of the proposed approach, positioning it as a promising solution for efficient and scalable Autoencoder implementations in hardware, with broader implications for diverse applications in the realm of image processing and beyond.

TABLE VI
CIRCLE PIXEL OUTPUT (GREYSCALE)

Iteration	Python	Proposed Design
3	0.6729	0.3438
	0.6729	0.3438
	0.6729	0.3438
	0.6729	0.3438
	0.4601	0.0980
	0.6729	0.3438
	0.6729	0.3438
	0.6729	0.3438
	0.6729	0.3438
59	0.7465	0.6797
	0.7465	0.6797
	0.7465	0.6797
	0.7465	0.6797
	0.2549	0.2188
	0.7465	0.6797
	0.7465	0.6797
	0.7465	0.6797
	0.7465	0.6797
175	0.7499	0.7500
	0.7499	0.7500
	0.7499	0.7500
	0.7499	0.7500
	0.2500	0.2500
	0.7499	0.7500
	0.7499	0.7500
	0.7499	0.7500
	0.7499	0.7500

TABLE VII
FIGURE OF MERITS

FPGA Board Utilization			
Resource	Used	Available	Utilization(%)
LUT	1048	53200	1.97
LUTRAM	376	17400	2.09
FF	61	106400	0.04
DSP	1	220	0.45
Performance Metrics			
Maximum Frequency	Power		Critical Path Speed
150 MHz	0.115 W		4.407 ns

REFERENCES

[1] S. Haykin, "Neural networrks a comprehensive foundation." Prentice Hall, 2002.

[2] Y. B. Ian Goodfellow and A. Courville, in *Deep Learning*. MIT Press, 11 2016.

[3] H. Tong, B. Liu, and S. Wang, "Software defect prediction using stacked denoising autoencoders and two-stage ensemble learning," *Information and Software Technology*, vol. 96, pp. 94–111, 2018.

[4] U. Michelucci, "An introduction to autoencoders," 2022.

[5] S. Brown and Z. Vranesic, "Fundamentals of digital logic and vhdl design," 2009.

Design and Implementation of a Learning Circuit for Kaomoji Generator with Autoencoder

Masato Shotoku, Takuya Hirahara, Kaito Tsuchiya
Graduate school of Engineering, Chiba University, Japan
Email: m.shotoku@chiba-u.jp

Abstract: We designed and implemented a learning circuit for *Kaomoji* generator using autoencoder. A simplified version of MLP-Mixer was used for the autoencoder's neural network. We designed the system with software/hardware co-design and implemented on a System on Chip (SoC). Learning on the designed system was 7.33 times faster than on the PC.

Keywords—Kaomoji, Autoencoder, MLP-Mixer, SoC

I. INTRODUCTION

We focused on "*Kaomoji*," which are composed of characters, to apply autoencoder to characters, and designed hardware to generate *Kaomoji*. The hardware was built using software/hardware co-design and implemented on a SoC.

An overview of our system is shown in Figure. 1.1. In learning, Processing System (PS) sends input data, which is also teacher data, to Programmable Logic (PL), and PL updates the parameters. In generating, PL generates *Kaomoji* using random numbers generated internally. This *Kaomoji* is sent to PS and displayed on a monitor.

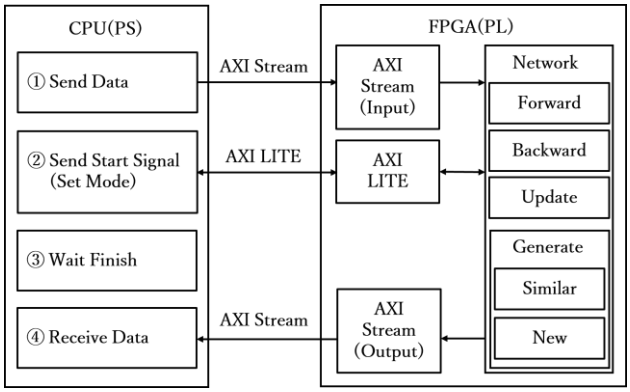


Figure. 1.1. System overview.

II. *KAOMOJI*

Kaomoji are a combination of letters and symbols that represent a person's facial expression and emotion. We focused on *Kaomoji* used in Japan, not on "emoticon" used in other countries.

(´ω`)ノ　ヾ(* ∀´ *)ノ　(* ¯∇ ¯)ノ
(≧Д≦;)　ヽ(^o^)ノ　(ТДТ)　m(-_-)m

Figure. 2.1. Examples of *Kaomoji*.

The preprocessing performed by our system is to convert each of the characters in *Kaomoji* into a one-hot vector. The preprocessing is performed as follows.

1. Disassemble the *Kaomoji* into individual characters.
2. Align to 10 characters by adding [PAD] to missing parts and convert unknown characters to [UNK].
3. Number each character using the list of characters (Table. 2.1.).
4. Convert each number to a one-hot vector.

Table. 2.1. List of characters and numbers.

0	PAD	12	"	24	●	36	¯	48	ヽ	60	]
1	UNK	13	ヾ	25	>	37	□	49	♪	61	[
2	(	14	*	26	д	38	!	50	Д	62	Σ
3	⌒	15	★	27	<	39	°	51	ゝ	63	゜
4	)	16	。	28	ノ	40	ε	52	;	64	|
5		17	_	29	-	41	O	53	≧	65	'
6	ω	18	/	30	☆	42	◇	54	≦	66	Т
7	σ	19	¥	31	@	43	m	55	〃	67	∇
8	・	20	―	32	o	44	=	56	▼	68	っ
9	、	21	´	33	▽	45	°	57	~	69	0
10	?	22	v	34	.	46	`	58	:	70	○
11	^	23	∀	35	·	47	ゞ	59	◎	71	#

III. GENERATION OF *KAOMOJI*

We tried two methods of controlling the latent vector, the input to the decoder, to generate new *Kaomoji*.

A. Generation of Similar *Kaomoji*

The latent vector obtained by inputting *Kaomoji* to the encoder is added to a random numbers before being input to the decoder to generate *Kaomoji* that are similar to the input.

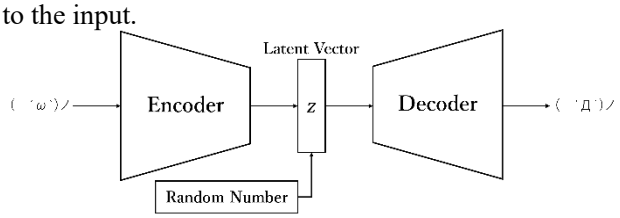


Figure. 3.1. Network for similarity generation.

B. Generation of New *Kaomoji*

The random numbers are set directly to the latent vector and input to the decoder to generate new *Kaomoji* that are not in the learning data.

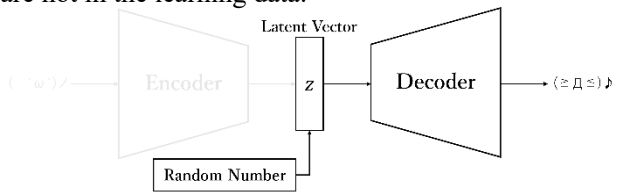


Figure. 3.2. Network for new generation.

IV. NEURAL NETWORK

A. Network Model

Figure. 4.1 shows the network model and Table. 4.1 shows the definitions of the constants in this model.

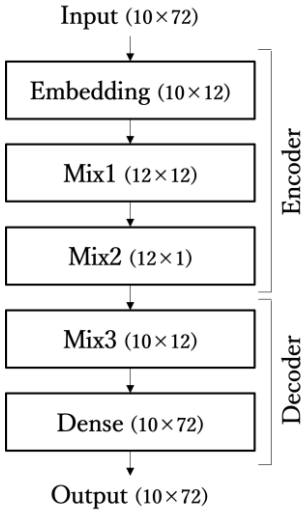


Figure. 4.1. Network model.

Table. 4.1. Constants used in the network.

Name	Value	Detail
N	10	Max charcters of *Kaomoji*
cnum	72	Number of available characters
emb	12	Dimensions of character vector
hid	12	Dimensions of latent vector
batch	32	Batch size

B. Forward Propagation

(a). Embedding Layer in Forward

In Embedding layer, each one-hot vector obtained in preprocessing is transformed into a 12-dimensional vector. The role of this layer is to acquire relationships between characters. For example, that " ・ " and " ∗ " are similar, and that " (" and ") " are symmetrical.

The process in this layer is the matrix product of matrix x_{e} and weights w_{e}. x_{e} is the matrix obtained by combining to one-hot vectors in the row direction.

$$y_{\mathrm{e}} = x_{\mathrm{e}} \cdot w_{\mathrm{e}} \tag{4.1}$$

$$y_{\mathrm{e}} \in \mathbb{R}^{\mathrm{N}\times\mathrm{emb}},\ x_{\mathrm{e}} \in \mathbb{R}^{\mathrm{N}\times\mathrm{cnum}},\ w_{\mathrm{e}} \in \mathbb{R}^{\mathrm{cnum}\times\mathrm{emb}}$$

(b). Mix Layer in Forward

In Mix layer, the matrix product is performed after transposition to prevent row-by-row independence of the computation. After transposition, different weights are used for each row. The method is based on MLP-Mixer[1], a network used in the image processing field that does not use convolution or attention.

Only Mix1 layer is described since each Mix layer is calculated in the same way, just the matrix shape is different.

The process in Mix1 layer is to transpose input x_{m1}, computed row by row, and combined to obtain y_{m1}.

$$a_{\mathrm{m1},i} = x_{\mathrm{m1},i} \cdot w_{\mathrm{m1},i} + b_{\mathrm{m1},i} \tag{4.2}$$

$$y_{\mathrm{m1},i} = \tanh\left(a_{\mathrm{m1},i}\right) \tag{4.3}$$

$$x_{\mathrm{m1}}^{\mathrm{T}} = \begin{bmatrix} x_{\mathrm{m1},1} \\ \vdots \\ x_{\mathrm{m1},i} \\ \vdots \\ x_{\mathrm{m1,emb}} \end{bmatrix} \quad y_{\mathrm{m1}} = \begin{bmatrix} y_{\mathrm{m1},1} \\ \vdots \\ y_{\mathrm{m1},i} \\ \vdots \\ y_{\mathrm{m1,emb}} \end{bmatrix}$$

$$x_{\mathrm{m1}} \in \mathbb{R}^{\mathrm{N}\times\mathrm{emb}},\ x_{\mathrm{m1},i} \in \mathbb{R}^{1\times\mathrm{N}},\ y_{\mathrm{m1}} \in \mathbb{R}^{\mathrm{emb}\times\mathrm{hid}}$$

$$a_{\mathrm{m1},i}, b_{\mathrm{m1},i}, y_{\mathrm{m1},i} \in \mathbb{R}^{1\times\mathrm{hid}},\ w_{\mathrm{m1},i} \in \mathbb{R}^{\mathrm{N}\times\mathrm{hid}}$$

Since the output of Mix2 layer is latent vector and not matrix, the vectors are transposed, replicated by 10, combined in the row direction, and then input to Mix3 layer.

(c). Dense Layer in Forward

The process in Dense layer is the matrix product of the input x_{d} and the weights w_{d}, and using SoftMax function to obtain y_{d}.

$$a_{\mathrm{d}} = x_{\mathrm{d}} \cdot w_{\mathrm{d}} \tag{4.4}$$

$$y_{\mathrm{d},i} = \mathrm{SoftMax}\left(a_{\mathrm{d},i}\right) \tag{4.5}$$

$$\mathrm{SoftMax}\left(x_j\right) = \frac{e^{x_j}}{\sum_{k=0}^{\mathrm{cnum}-1} e^{x_k}} \tag{4.6}$$

$$a_{\mathrm{d}} = \begin{bmatrix} a_{\mathrm{d},1} \\ \vdots \\ a_{\mathrm{d},i} \\ \vdots \\ a_{\mathrm{d,N}} \end{bmatrix} \quad y_{\mathrm{d}} = \begin{bmatrix} y_{\mathrm{d},1} \\ \vdots \\ y_{\mathrm{d},i} \\ \vdots \\ y_{\mathrm{d,N}} \end{bmatrix}$$

$$x_{\mathrm{d}} \in \mathbb{R}^{\mathrm{N}\times\mathrm{hid}},\ w_{\mathrm{d}} \in \mathbb{R}^{\mathrm{hid}\times\mathrm{cnum}}$$

$$a_{\mathrm{d}}, y_{\mathrm{d}} \in \mathbb{R}^{\mathrm{N}\times\mathrm{cnum}},\ a_{\mathrm{d},i}, y_{\mathrm{d},i} \in \mathbb{R}^{1\times\mathrm{cnum}}$$

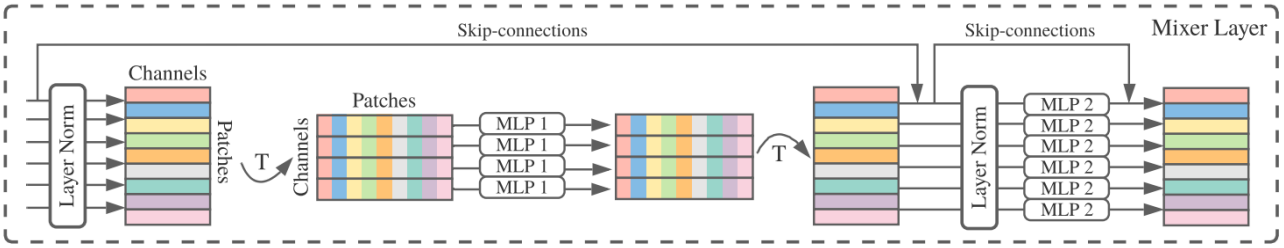


Figure. 4.2. MLP-Mixer[1]

V. LEARNING

A. Creating *Kaomoji* Dataset

For this model, a dataset of *Kaomoji* was created before learning. The *Kaomoji* were collected from the Web site[1][2] by Web scraping using the Python library BeautifulSoup.

Source code. 5.1. Example of Web scraping.

```
def collect_kaomoji_copy(url):
    res = requests.get(url)
    soup = BeautifulSoup(res.content, 'html.parser')
    el = soup.find('table',
          attrs={ 'class': 'kaomoji' }).find_all('input')
    return el
```

This function reads data from the argument URL and then retrieves *Kaomoji* by specifying HTML tags.

We augmented the data by inverting the left and right sides of the *Kaomoji*. As a result, the dataset contains 10,300 *Kaomoji*.

Table. 5.1. Example of data augmentation.

Before	After	
(≧∀≦*)	(*≧∀≦)	Asymmetry
(o ゜∇ ゜) σ	σ (゜ ∇゜ o)	Asymmetry
(≧ω≦)	None	Symmetry
♪(>д<)♪	None	Symmetry

B. Back Propagation

In autoencoder learning, the output values y obtained by forward propagation and the input values x, which is also teacher data, compute the loss. In our system, cross-entropy-loss is used as the loss function, which is calculated by the following equation.

$$E = \sum_{i}^{\text{batch}} \sum_{j}^{\text{N}} \sum_{k}^{\text{cnum}} \left(\frac{x_{i,j,k} \log y_{i,j,k}}{\text{batch}} \right) \tag{5.1}$$

$$x, y \in \mathbb{R}^{\text{N}\times\text{cnum}}$$

(a). Dense Layer in Backward

The values obtained by back propagation in Dense layer are the gradient G_d of E for the weights w_d and the gradient D_d of E for the input x_d. The equations for obtaining these are given below.

$$\text{G}_\text{d} = \frac{\partial E}{\partial w_\text{d}} = x_\text{d}^\text{T} \cdot \text{d}_\text{s} \tag{5.2}$$

$$\text{D}_\text{d} = \frac{\partial E}{\partial x_\text{d}} = \text{d}_\text{s} \cdot w_\text{d}^\text{T} \tag{5.3}$$

$$\text{d}_\text{s} = \frac{y - x}{\text{batch}} \tag{5.4}$$

$$\text{G}_\text{d}, w_\text{d} \in \mathbb{R}^{\text{hid}\times\text{cnum}},\ x, y, \text{d}_\text{s} \in \mathbb{R}^{\text{N}\times\text{cnum}}$$

$$\text{D}_\text{d}, x_\text{d} \in \mathbb{R}^{\text{N}\times\text{hid}}$$

d_s are the back propagation values from SoftMax function.

(b). Mix Layer in Backward

The back propagation of the three Mix layers is similar, so only Mix1 layer is explained.

The values obtained in the back propagation of Mix1 layer are $\text{Gw}_{\text{m1},i}$, $\text{Gb}_{\text{m1},i}$, $\text{D}_{\text{m1},i}$ $(i = 1,2 \cdots \text{emb})$.

$\text{Gw}_{\text{m1},i}$ is the gradient of E for the weight $w_{\text{m1},i}$ and $\text{Gb}_{\text{m1},i}$ is the gradient of E for the bias $b_{\text{m1},i}$. $\text{D}_{\text{m1},i}$ is the gradient of E for the input $x_{\text{m1},i}$. The equations for obtaining these are shown below.

$$\text{d}'_{\text{m1},i} = \frac{\partial E}{\partial a_{\text{m1},i}} = \text{d}_{\text{m1},i} \cdot f'\left(x_{\text{m1},i}\right) \tag{5.5}$$

$$f'\left(x_{\text{m1},i}\right) = \left(1 - \tanh^2 x_{\text{m1},i}\right) \tag{5.6}$$

$$\text{Gw}_{\text{m1},i} = \frac{\partial E}{\partial w_{\text{m1},i}} = x_{\text{m1},i}^\text{T} \cdot \text{d}'_{\text{m1},i} \tag{5.7}$$

$$\text{Gb}_{\text{m1},i} = \frac{\partial E}{\partial b_{\text{m1},i}} = \text{d}'_{\text{m1},i} \tag{5.8}$$

$$\text{D}_{\text{m1},i} = \frac{\partial E}{\partial x_{\text{m1},i}} = \text{d}'_{\text{m1},i} \cdot w_{\text{m1},i}^\text{T} \tag{5.9}$$

$$x_{\text{m1}}^\text{T} = \begin{bmatrix} x_{\text{m1},1} \\ \vdots \\ x_{\text{m1},i} \\ \vdots \\ x_{\text{m1,emb}} \end{bmatrix} \quad \text{D}_{\text{m1}} = \begin{bmatrix} \text{D}_{\text{m1},1} \\ \vdots \\ \text{D}_{\text{m1},i} \\ \vdots \\ \text{D}_{\text{m1,emb}} \end{bmatrix}^\text{T}$$

$$x_{\text{m1}}, \text{D}_{\text{m1}} \in \mathbb{R}^{\text{N}\times\text{emb}},\ x_{\text{m1},i}, \text{D}_{\text{m1},i} \in \mathbb{R}^{1\times\text{N}}$$

$$a_{\text{m1},i}, \text{Gb}_{\text{m1},i}, b_{\text{m1},i}, \text{d}_{\text{m1},i}, \text{d}'_{\text{m1},i}, y_{\text{m1},i} \in \mathbb{R}^{1\times\text{hid}}$$

$$\text{Gw}_{\text{m1},i}, w_{\text{m1},i} \in \mathbb{R}^{\text{N}\times\text{hid}}$$

$\text{d}_{\text{m1},i}$ are the back propagation values from Mix2 layer.

Since the back propagation values for Mix3 layer is vector and not matrix, the back propagation values D_{m3} is converted to the vector by adding row by row before propagation.

(c). Embedding Layer in Backward

The values obtained by back propagation in Embedding layer is the gradient G_e of E for the weight w_e. The equation for obtaining G_e is shown below.

$$\text{G}_\text{e} = \frac{\partial E}{\partial w_\text{e}} = x_\text{e}^\text{T} \cdot \text{d}_\text{e} \tag{5.10}$$

$$\text{G}_\text{e}, w_\text{e} \in \mathbb{R}^{\text{cnum}\times\text{emb}},\ x_\text{e} \in \mathbb{R}^{\text{N}\times\text{cnum}},\ \text{d}_\text{e} \in \mathbb{R}^{\text{N}\times\text{emb}}$$

x_e are the input values and d_e are the back propagation values from Mix1 layer.

C. Updating Parameter

The updated formula for the optimization algorithm Momentum used is given below.

$$v_{t+1} = \beta v_t - \alpha \text{G} \tag{5.11}$$

$$w_{t+1} = w_t + v_{t+1} \tag{5.12}$$

G is the gradient and w is the parameter of the layer. The hyperparameters are α and β. The values in this model are given below.

$$\alpha = 0.001,\ \beta = 0.9$$

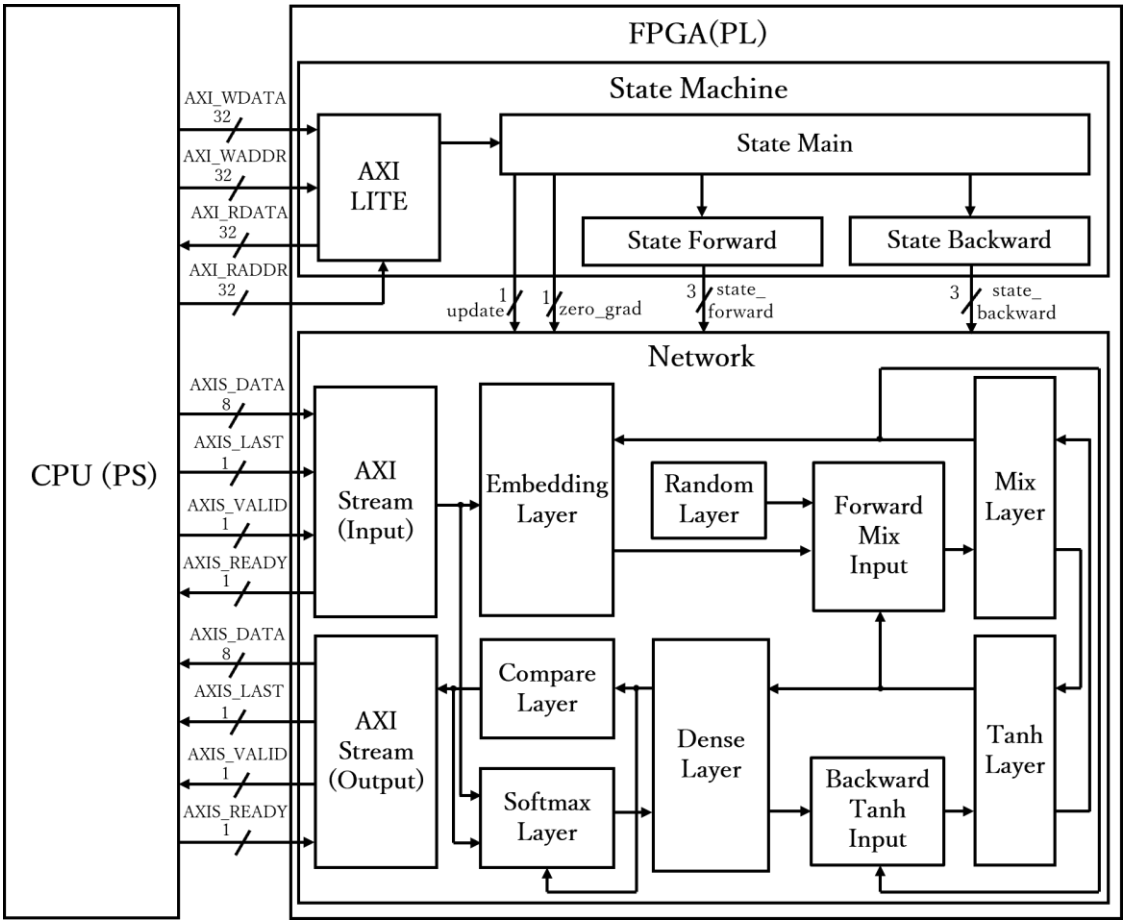


Figure. 6.1. General view of the circuit.

VI. CIRCUIT STRUCTURE AND OPERATION

A. Operation Mode on Circuit

In our system, PS controls PL in AXI4-LITE format. Control signals and their addresses in AXI4-LITE format are shown in Table 6.1.

Table. 6.1. Control signals and addresses.

Address (32bit)	Data (32bit)
0	next, run, rst_n
1	mode (2bit)
2	fin_backward, fin_forward, fin_main

The lower 3 bits of address 0 are assigned to the reset signal "rst_n", the start signal "run", and the signal "next", which is used when the circuit is operated continuously. the lower 2 bits of address 1 are assigned to "mode", which indicates the operation mode of the circuit (Table. 6.2), and the lower 3 bits of address 2 are assigned to the end signals.

Table. 6.2. Circuit operating modes.

Mode	Value	Detail
TRAIN	0	Calculates gradient and Updates parameter
FORWARD	1	Forward propagation and Outputs *Kaomoji*
SIMILAR	2	Generates similar *Kaomoji*
NEW	3	Generates new *Kaomoji*

The following is a description of process in each mode. First, in FORWARD mode, the following process is performed.

1. Send *Kaomoji* data from PS to PL using AXI4-Stream (Input).
2. Send "run" signal from PS to PL in AIX4-LITE format for forward propagation.
3. Obtain the number of each character in Compare layer and stores it in the FIFO.
4. Send end signal from PL to PS in AXI4-LITE format.
5. Send output data from the FIFO to the PS using AXI4-Stream (Output).
6. To continue forward propagation, send "next" signal and perform steps 1. through 5. again.

In SIMILAR mode, random numbers are added to the Mix2 layer output. In NEW mode, all Mix3 layer inputs are set to random numbers. The random numbers are generated by Random layer using Xorshift[4] algorithm.

Finally, TRAIN mode is described below.

1. The processing is performed up to Compare Layer in the same way as in FORWARD mode.
2. Back propagation is performed from SoftMax Layer to obtain the gradient of each parameter, which is stored in RAM.
3. After calculating the gradient for one batch, the parameters are updated.

4. When the parameter update is completed, PL sends end signal.
5. When training multiple batches, send "next" signal from PS and perform steps 1. through 4. again.

B. Learning on Circuit

In learning, parameter updates are performed in mini-batch learning, with a batch size of 32. In our system, mini-batch learning is performed by computing the gradient multiplied by 1/32 and adding them together 32 times for one batch of data. Since the gradient values need to be initialized for each batch, the state machine`s zero_grad is used to initialize the gradient during the first forward propagation.

C. Layer Module

The schematic of Dense Layer is shown in Figure. 6.2. The basic structure of Embedding Layer, Mix Layer and Dense Layer are all similar. They are composed of "Forward" for forward propagation, "Backward" for back propagation, "Optimizer" for updating parameters, and RAM for storing parameters.

There are two RAMs for storing weights, RAM wt and RAM w. The two RAMs allow for independent forward and back propagation, thus can be executed in a pipeline.

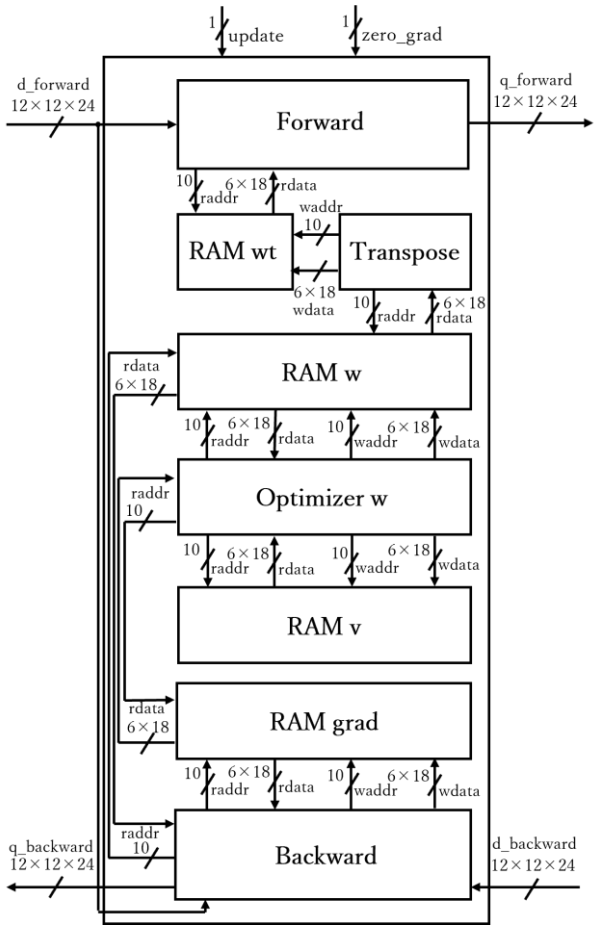


Figure. 6.2. Dense Layer.

VII. DESIGN INNOVATION

A. Reduction of Bit Width

Before designing the model in hardware, an emulator was built to verify the learning and generation operations. The emulator was designed with fixed-point precision, and the number of bits for each parameter and the forward and back propagation values were reduced to maintain at least 60% accuracy.

B. Pipeline processing and Parallelization

In this design, pipelining was used for each of the following three tasks: sending and receiving learning data, forward and back propagation in one batch, and data readout and computation using the values. For forward and back propagation, pipelining was made possible by using different Block RAMs for each. For calculations where the input values are matrices, parallelization was performed row by row. With these innovations, the number of clocks required to learn one epoch was reduced by 1/177.

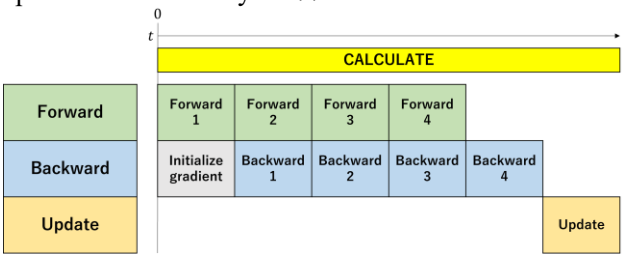


Figure. 7.1. Pipeline of forward and backward

C. Reduction of Resource Usage in Mix Layer

The three Mix layers are computed using the same module multiple times to reduce the resource usage. This innovation reduced the amount of LUTs, FFs, and DSP slices used in the three Mix layers by 59.9%, 62.3%, and 62.5% respectively.

VIII. EVALUATION

A. Resources Usage in System

ZCU104[5] in AMD Xilinx was used for the actual operation. For device control, PYNQ[6], an OS image targeting SoC, was used. Table. 9.2 shows the resources and the maximum operating frequency of our system. Xilinx Vivado 2023.2 was used as the logic synthesis and place-and-route tool.

Table. 8.1. Resources and Maximum Frequency.

Resources	Utilization	Utilization [%]
LUT	99774	43.30
LUTRAM	935	0.92
FF	160785	34.89
BRAM	123.5	39.58
DSP	694	40.16
IO	4	1.11
BUFG	2	0.37
Maximum Frequency [MHz]	126.18	–

B. Evaluation of Speed and Accuracy

Figure. 8.1 shows the learning time of the emulator on PC (Table. 8.2) and our system on SoC for the epoch. The clock frequency was set to 125 MHz.

Table. 8.2. Details of the PC for the emulator.

CPU	Intel Core i7-11700 2.50GHz
RAM	64 GB
OS	Ubuntu 18.04 LTS

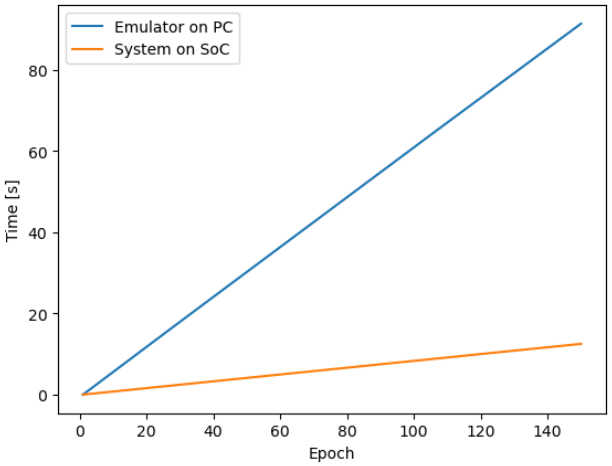


Figure. 8.1. Learning time for epochs.

Figure. 8.1 shows that the learning time at 100 epochs was 61.5 seconds for the PC and 8.39 seconds for the SoC. Thus, the learning with the designed hardware was about 7.33 times faster than the PC learning.

Next, Figure. 8.2 shows the accuracy of the emulator on the PC and our system on the SoC for the epoch.

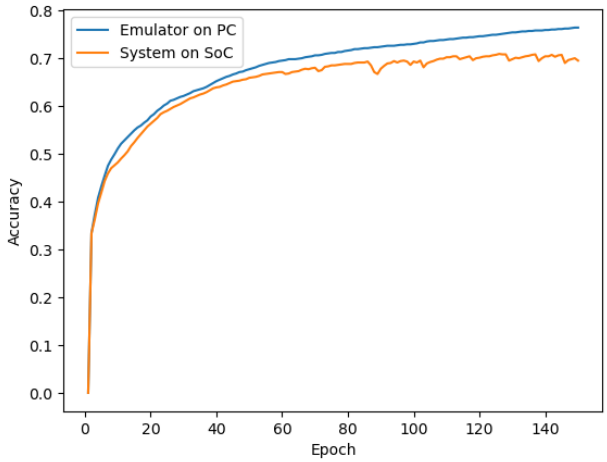


Figure. 8.2. Accuracy for epochs.

These accuracies were calculated for each character of *Kaomoji*. Figure. 8.2 shows that the maximum accuracy for the SoC is 70.9%.

C. Result of *Kaomoji* Generation

The results of similarity generation and new generation on ZCU104 are shown below. The probability of new generation was about 20% in a subjective evaluation.

Table. 8.3. Examples of Similar Generation.

Base	Examples of Similar *Kaomoji*	
ヾ(＊　∀´　＊)ノ	ヾ(＊　∀´　＊)	ヾ(￣　∀゜＊)ノ
	ヾ(＊　∀￣　＊)	ヾ(＊　∀￣　))ノ
	ヾ(￣　ω　　＊)ノ	(＊　ώ　＊)
	ヾ(*;∀´　＊)ノ	(＊　ω-　))♪

Table. 8.4. Examples of New Generation.

Examples of Success		Examples of Failure
ヾ(・`∀・　)	(　∀`)・!	(　^`　。)
)!(^-^　)))	(*-^・^)-)!	(^)^*^-!
ヾ(≧^　^)	♪(￣　∀´　))	!・　;))・・!
(`￣^∀　)ノ	(・∀-)ノ	ヾ(・))・_-)
(^　^)・!	(￣-´*)ノ!	)((∀^)ノ

IX. CONCLUSION

We focused on "*Kaomoji*" to apply autoencoder to characters, and designed and implemented a learning circuit for *Kaomoji* generator. The *Kaomoji* generator was constructed using a five-layer neural network inspired by MLP-Mixer. Two methods were used to generate *Kaomoji*: similarity generation and new generation. The model achieved a 70.9% accuracy, which meets the target of "more than 60% accuracy," considering the fixed-point precision and the generation rate of *Kaomoji*. The learning using our system on the SoC was 7.33 times faster than that using the emulator on the PC.

As a future prospect, the accuracy can be improved by using different models such as VAE (Variational Autoencoder), which introduces probability distribution in the latent vector.

Although our system mainly targets *Kaomoji* used in Japan, it could conceivably target emoticon used mainly in other countries (Figure 9.1).

:-)　8-)　:-(　: D

Figure. 9.1. Examples of emoticon.

REFERENCES AND LINKS

[1] Ilya Tolstikhin. et al. “MLP-Mixer: An all-MLP Architecture for Vision.” arXiv preprint arXiv: 2105.01601, 2021.

[2] https://kaomoji-copy.com, Accessed on 2024/02/28.

[3] https://kmoji.com, Accessed on 2024/02/28.

[4] Marsaglia George. “Xorshift RNGs.” Journal of Statistical Software 8, no. 14 2003: 1-6.

[5] https://japan.xilinx.com/products/boards-and-kits/zcu104.html, Accessed on 2024/02/28.

[6] http://www.pynq.io, Accessed on 2024/02/28.

Development of a Hazard Map Prediction Circuit Using an Autoencoder

Tomoki Kanno, Kosuke Mori, Takaho Ueyama
Graduate school of Enginering, Chiba University
Chiba, Japan
Email:as_tomo_as@chiba-u.jp

***Abstract*—We designed a circuit for an autoencoder that generates hazard maps from geographic information and built a system on an FPGA that outputs the generated hazard maps as 3D models. In addition, it was designed to achieve a circuit size that could fit on an FPGA evaluation board and to speed up the computation processing time.**

Keywords— Autoencoder, FPGA, Pipelining, 3D model, Hazard map, disaster preventation

I. Introduction

In recent years, natural disasters have become more severe and more frequent. In order to protect human lives from such disasters, it is important to predict the risk areas in advance and take all possible countermeasures. A hazard map is a map that shows the expected disaster areas, the extent of damage, evacuation sites, and evacuation routes, based on predictions of the damage that would occur in the event of such a natural disaster. Examples of hazard maps are shown in Figure 1. While the use of hazard maps in natural disasters is important to minimize the damage caused by disasters, hazard maps do not cover all areas at risk of disasters. Therefore, there is a problem that the area is erroneously perceived as a safe area even though it is at risk of disaster. For this reason, we aimed to design a hardware that can predict a hazard map displaying various disaster risks in an arbitrary region by using an autoencoder, which was the design theme of this LSI design contest. Using this hardware, we thought it would be possible to visually confirm the possibility of various disasters occurring in any given area's topography, and thus solve the problem, and took on the challenge of designing the circuit.

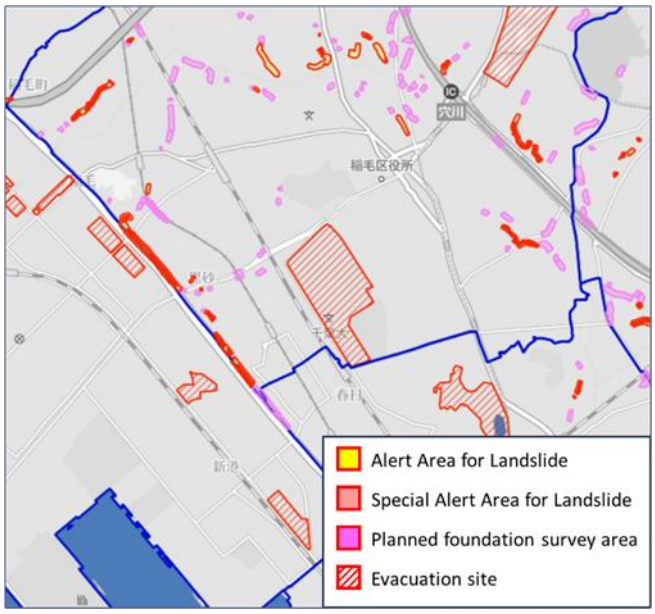


Fig. 1. Hazard map around Chiba University
(reference: 千葉市地震・風水害ハザードマップ[1])

The specific operation of the system is as follows. By inputting 32 × 32 elevation data into the autoencoder, which consists of three layers, the system outputs data indicating which areas of the land are at risk from a certain disaster. Outputting these data as a 3D model, the system can generate a hazard map for any given area.

II. Learning Method

Initially, the details of the autoencoder are described. Figure 2 shows an outline of the autoencoder created by this system. An autoencoder is a network in which the number of units in the input and output layers are the same and the input is restored. It consists of two parts: an encoder that extracts features from the input data to create a compact representation, and a decoder that reconstructs the original input from the compact representation. Suppose the number of units in the input and output layers is n and the number of units in the hidden layer is m. Define the input matrix $\boldsymbol{x}$ and the output matrix $\boldsymbol{o}$ in the autoencoder as in equations (1) and (2), respectively.

$$\boldsymbol{x} = (x_1, x_2, \dots x_n)^T \quad (1)$$
$$\boldsymbol{o} = (o_1, o_2, \dots o_n)^T \quad (2)$$

where x_n is the nth input value and o_n is the nth output value. In the model shown in Figure 2, the computation is performed according to equation (3)

$$\boldsymbol{o} = f\left(\tilde{\boldsymbol{w}} f(\boldsymbol{w}x + \boldsymbol{b}) + \tilde{\boldsymbol{b}}\right) \quad (3.3)$$

where $\boldsymbol{w}, \tilde{\boldsymbol{w}}$ are weight matrices, $\boldsymbol{b}, \tilde{\boldsymbol{b}}$ are bias matrices, $\boldsymbol{w} \in \mathbb{R}^{\boldsymbol{m}\times\boldsymbol{n}}$, $\boldsymbol{b} \in \mathbb{R}^{\boldsymbol{m}\times\boldsymbol{1}}$, $\tilde{\boldsymbol{w}} \in \mathbb{R}^{\boldsymbol{m}\times\boldsymbol{n}}$, and $\tilde{\boldsymbol{b}} \in \mathbb{R}^{\boldsymbol{n}\times\boldsymbol{1}}$. In addition, $f(\cdot)$ is a function called the activation function, which is used to improve the accuracy of learning.

This system is designed to output hazard maps for an area when arbitrary geographic information is input via an autoencoder, and it is necessary to train the system to handle various land shapes. Therefore, we created a dataset consisting

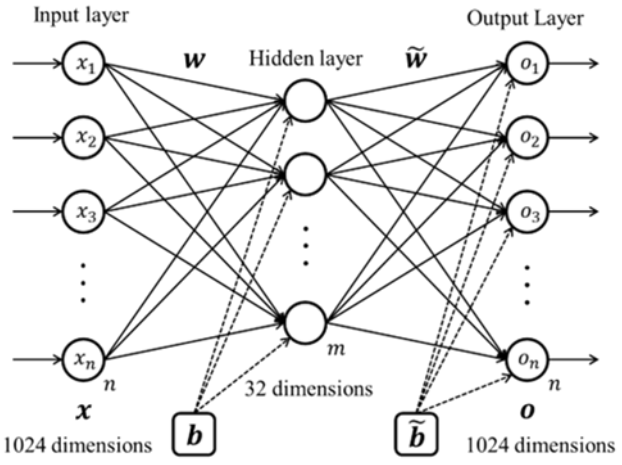


Fig. 2. Autoencoder used in this system

Fig. 3. Part of the datasets

of geographic information data and hazard map data necessary for training. The elevation distribution of any given land in Japan was obtained by referring to the "Hazard Map Portal Site [2]" provided by Geospatial Information Authority of Japan. The data was then normalized to 10 m intervals of elevation in the range of 0 to 2550 m to generate a 256-level grayscale image. In addition, an image of the disaster hazard areas in the same area was created by painting them white, and these data sets were used as training data (Fig. 3). We selected three types of disasters for which this system outputs hazard maps: landslide disasters, tsunamis, and floods. The number of datasets used in this training was 2128 for landslide disasters, 952 for tsunamis, and 672 for floods, for a total of 3752 training datasets. In addition, 33 landslide disaster sets, 4 tsunami sets, and 4 flood sets were created as test data. Using these data, training was conducted on an autoencoder that outputs hazard maps for landslide disasters, tsunamis, and floods when geographic information is input.

The training of the autoencoder was done entirely in Python. Table 1 shows the each parameters of autoencoder designed for this system.

Table. 1. Each parameter in learning

Epoch	0
Batch size	64
Activation Function	ReLU Function
Error function	Mean Square Error
Optimization Method	Adam
Learning start weights	All random
Learning start bias	All 0

Figure 4 shows an example of the results of learning under these conditions. After the autoencoder training is completed, the final vector of weights and biases is processed appropriately and output as text data in binary notation. Using this data, the autoencoder process can be implemented in an FPGA

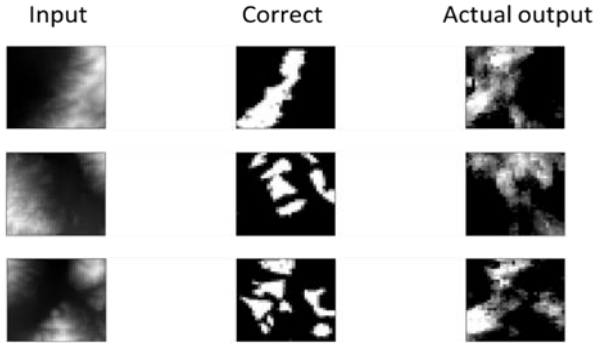


Fig. 4. Part of the learning results

III. Circuit Design

The FPGA evaluation board used in the implementation was the Zynq FPGA made by AMD-XilinX. Table 2 shows the evaluation board specifications and development environment. In addition to the FPGA, this evaluation board is equipped with a SoC. Table 3 shows the specifications of the PS part of the SoC.

Table. 2. Evaluation Board Specifications

FPGA	Zynq Ultra Scale+ MPSoC ZCU104
Logic cell	504,000
DSP Slice	1,728
Block RAM	11[Mb]
Communication Protocol	AXI4-Stream AXI4-Lite
Development environment	Vivado 2023.1

Table. 3. Evaluation Board Specifications

CPU	Cortex-A53 1.5GHz 670 CPU
Memory	2.0[Gb]
OS	PYNQ Linux based on Ubuntu 22.04
Compiler	g++ 11.2.0
Development environment	Jupyter Notebook version 6.3.0

In this system, the inference part was implemented on an FPGA, using parameters learned on an emulator. Figure.5 shows an overall diagram of the configured system. The circuits designed for this system can be divided into two parts: the AXI communication circuit and the Main circuit, which performs the calculation and signal control of the autoencoder. For communication between the PS and PL sections, the Slave / Master AXI-4 Stream Module using the AXI-4 (Advanced eXtensible Interface 4) [3] standard and an AXI-4 Lite Module were used. AXI-4 Stream supports data burst transfers and can transfer a sequence of data at a time instead of having no address specification. AXI-4 Lite supports transactions with simple control, and instead of burst transfers, it can be controlled differently by addressing.

The Main circuit can be divided into three parts: the Input/Output Data Control circuit, which adjusts the bit width of input/output data and connects until ready to send/receive data; PS Operation Status, which controls the autoencoder circuit from the PS side; and Autoencoder Calculation Module, which performs the autoencoder operations. The Autoencoder Calculation Modul is further divided into an Encoder Module which extracts features from the input data, a Decoder Module which composes the output data from the features, and a State Machine which controls the state of these modules. Figure 6 is a schematic of the Autoencoder Calculation Modul.

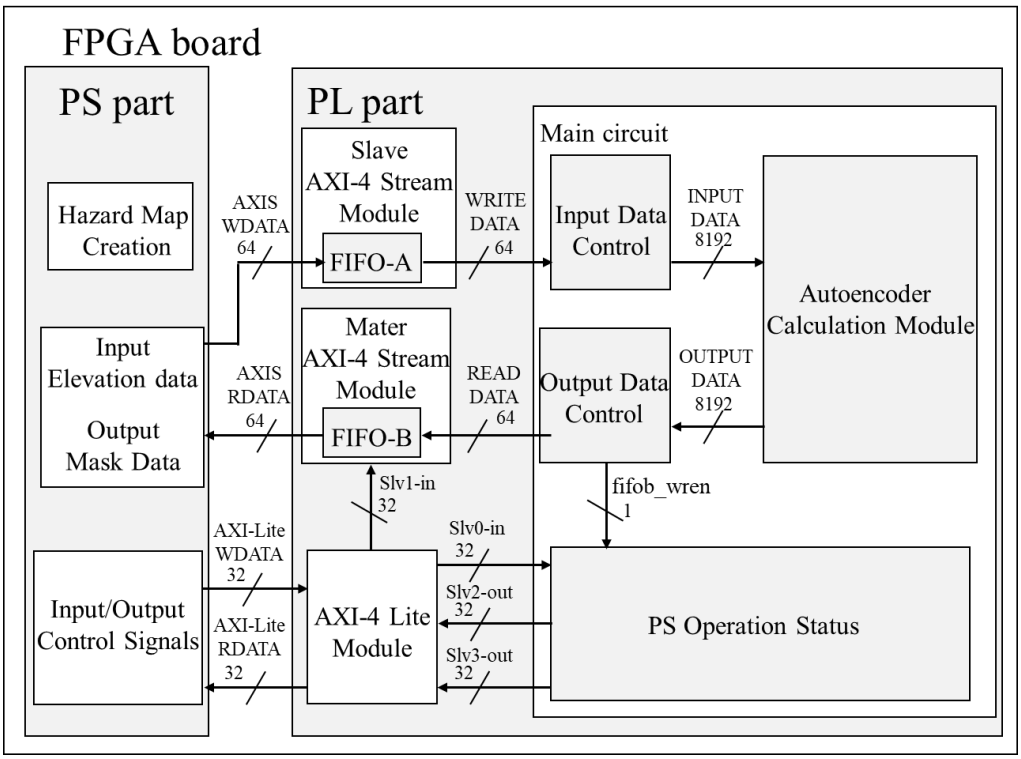


Fig. 5. System configuration

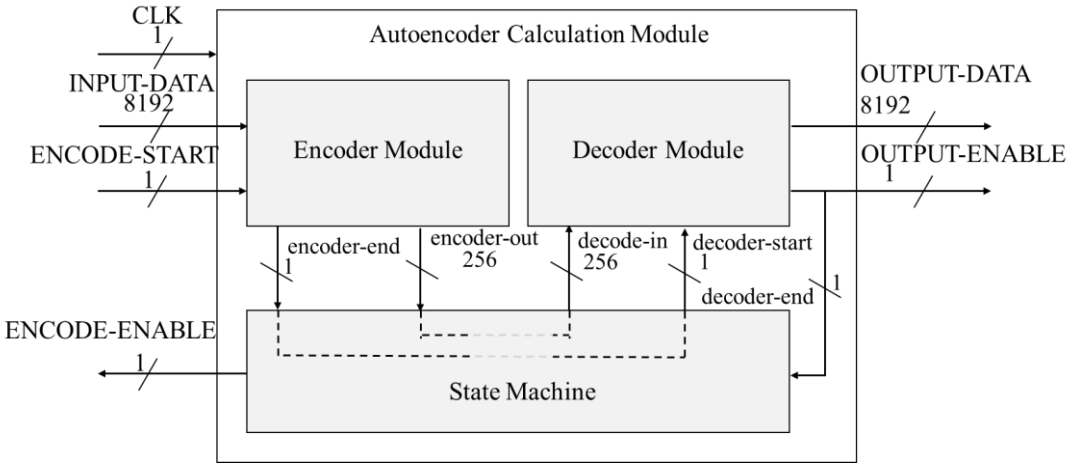


Fig. 6. Autoencoder Calculation Modul

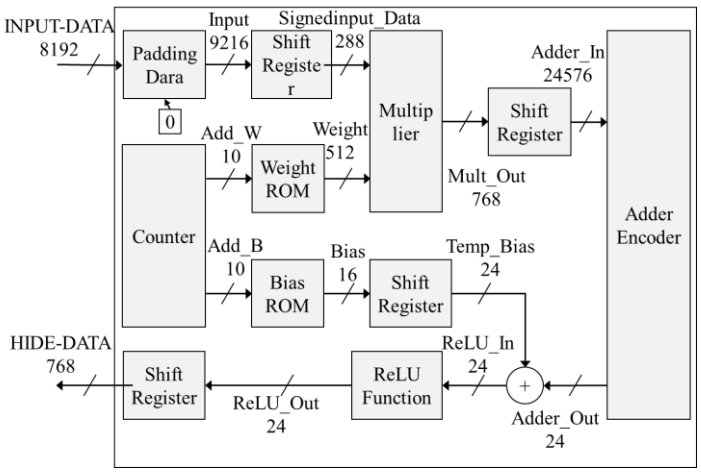


Fig. 7. Encoder Module

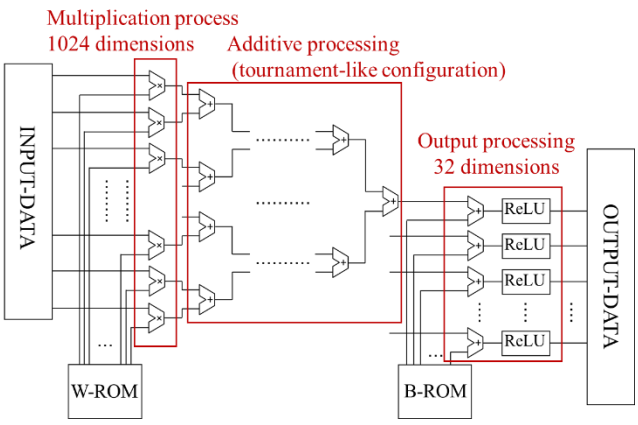


Fig. 8. Calculation Process in Encoder Module

Figure 7 shows a schematic of the Encoder Module and Figure 8 shows a diagram of the flow of processing within the module. The Encoder Module compresses the input 1024-dimensional elevation data into a 32-dimensional hidden layer. One-dimensional elevation data of 1024 x 8 = 8192 bits is input to the Encoder Module, and a positive sign "0" is added to the beginning of each data in the Padding Data section before sending it to the Multiplier circuit. The Multiplier circuit multiplies the input data and weights. Figure 9 shows a schematic of the multiplication process in Encoder Module. Since the input data is 9 bits and the weights are 16 bits, the result of each multiplication is 25 bits, of which the lower 24 bits are used as the multiplication result. To account for carry-over in subsequent addition processes, 19 bits are extracted from the 24-bit data, and depending on the sign, 5 bits of "00000" or "11111" are added to the beginning. This modified data is then sent to the Adder Encoder circuit. The Adder Encoder circuit adds up all the results of the previous multiplication and sends them to the ReLU function circuit. By arranging the adders in a tournament-like configuration, the structure is such that the addition process can be completed in 10 clocks. The ReLU function circuit outputs the value as it is if the sign bit of the input data is 0, and outputs "0" for 24 bits if it is 1. This process is repeated, and "HIDE-DATA" is output as 1-dimensional data when all 32 dimensions of the hidden layer data are available.

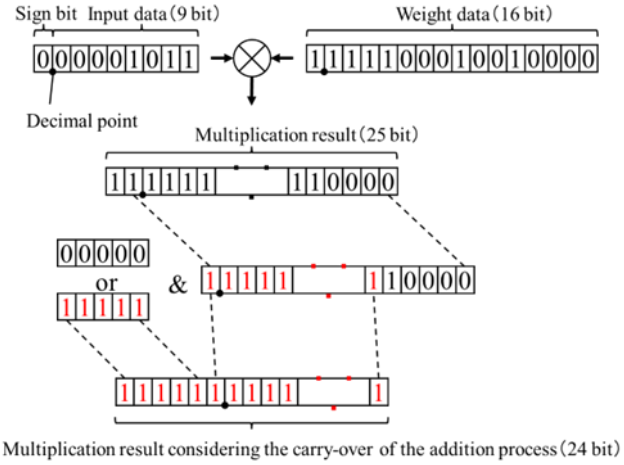


Fig. 9. Multiplication process in Encoder Module

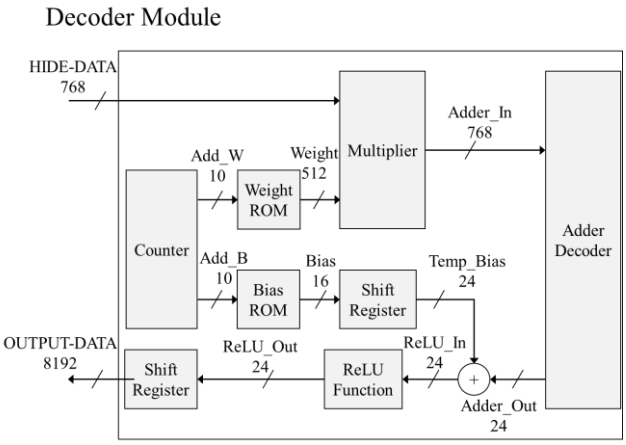


Fig. 10. Decoder Module

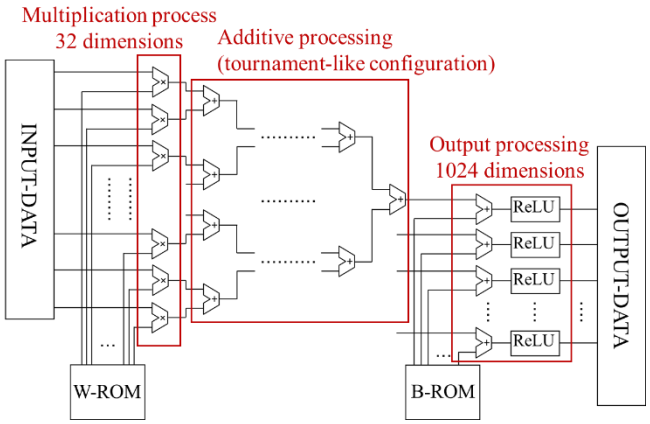


Fig. 11. Calculation Process in Decoder Module

Figure 10 shows a schematic of the Decoder Module and Figure 11 shows a diagram of the flow of processing within the module. The Decoder Module processes the circuit to reconstruct the input 32-dimensional hidden layer into a 1024-dimensional output layer. HIDE-DATA" input to the Decoder Module is sent directly to the Multiplier circuit. Figure 12 shows a schematic of the multiplication process in Decoder Module. The input data is 24 bits and the weights are 16 bits, so the result of each multiplication is 40 bits, of which the appropriate 24 bits are sent to the Adder Decoder circuit as the result of multiplication. Similar to the Adder Encoder circuit, the Adder Decoder circuit adds the multiplication results of the input and weight data, and inputs the results to the ReLU function circuit. The adder decoder circuit is configured to complete the addition process in 5 clocks because the adders are arranged in a tournament-like configuration. The ReLU function circuit performs the same processing as the Encoder Module, extracts the decimal 8 bits from each of the 1024-dimensional data, and outputs them as 1-dimensional OUTPUT-DATA.

This system further reduces calculation time by pipelining in three locations. Figure 13 shows a schematic of pipelining. First, (a) is the multiplier circuit. By performing the multiplication process 32 times in parallel, the conventional process of 3 clocks x 32 times = 96 clocks can be reduced to 3 + 32 - 1 = 34 clocks. This is an approximately 2.8-fold reduction in processing time. Second, (b), the Encoder Module is pipelined 32 times, and the Decoder Module is pipelined 1024 times. As a result, the Encoder Module can reduce the number of processing clocks to 1043 clocks, while the Decoder Module can reduce the number of processing clocks to 1050 clocks, from 3552 clocks for the Encoder Module and 10240 clocks for the Decoder Module. These changes will reduce processing time by a factor of approximately 3.4 in the encoder module and by a factor of

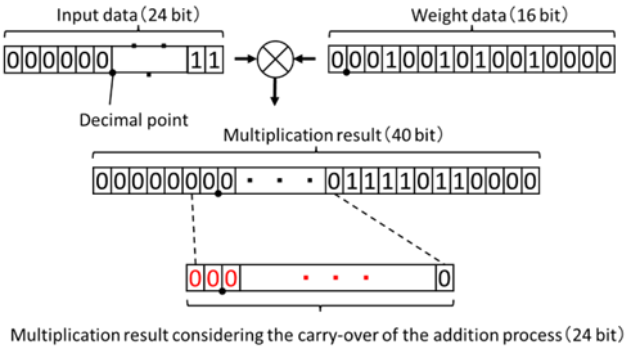


Fig. 12. Multiplication process in Decoder Module

approximately 9.8 in the decoder module. Third, (c) is pipelining in the Autoencoder Calculation Module. Since the clocks required for the Encoder and Decoder are different, the timing of processing is controlled by the State Machine. This reduces processing time by a factor of two when processing multiple sets of elevation data.

Finally, the Hazard Map Prediction circuit that outputs the results obtained from the autoencoder calculation as a 3D hazard map is described. An overview of the system of the Hazard Map Prediction circuit is described. Figure 14 shows a schematic diagram of data transmission and processing in the Hazard Map Prediction circuit. When the elevation data and control signals are output from the PS section to the PL section, the PL section performs the auto encoder calculation using those data. Upon completion of the auto encoder calculation, the data indicating the hazard areas and the control signal are sent to the PS section. At last, the received data is processed by the application to obtain a 3D hazard map. PYNQ (PythonProductivity for Zynq) [4], which is used in the OS of the PS section, is an open source project provided by AMD-XilinX, and allows data processing and data transfer to the PL section using Python. The advantages of using PYNQ are twofold: applications can be developed in Python and PYNQ can be easily controlled remotely. This will enable the development of Python-based applications in the PS section, and will also make it easier for users to use the system remotely, which is expected to lead to practical application as an IoT system. This system is designed to incorporate an application that displays a 3D model of the hazard map and allows remote control and file transfer.

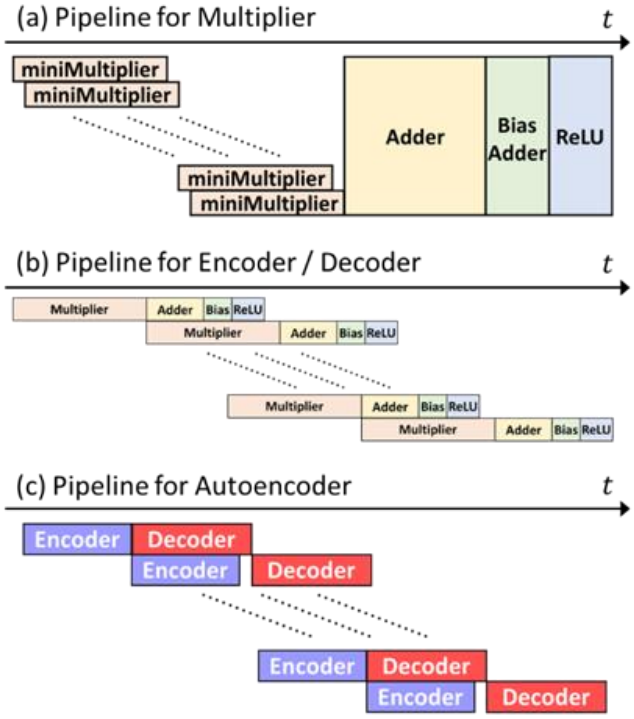


Fig. 13. Pipelining

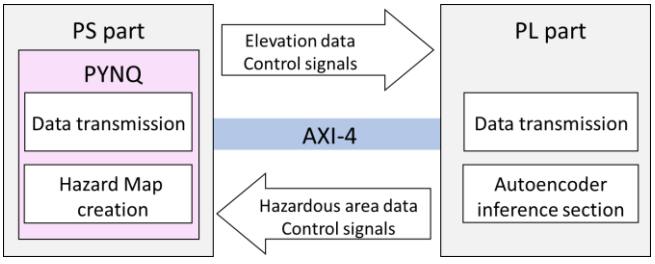


Fig. 14. Data exchange in Application section

IV. System Evaluation

Table 4 shows the circuit size and maximum operating frequency of the designed circuit. The FPGA operating frequency is set at 100 MHz. Table 5 shows a comparison of processing speeds on an actual device. All processes were measured five times each, and their average values are shown.

Table. 4. Circuit size and maximum operating frequency

Resources	Used resources	Utilization[%]
LUT	86021	37.34
LUTRAM	789	0.78
FF	150843	32.74
BRAM	63	20.19
DSP	64	3.7
BUFG	24	4.41
Maximum Frequency	161.06[MHz]	

Table. 5. compute processing speed

Environment	1 set	64 sets
PC	0.0430[s]	2.5782[s]
FPGA PS part	0.3102[s]	19.663[s]
Development method	0.0022[s]	0.0042[s]

PC is an AMD Ryzen7 2700 PC on which the autoencoder was calculated using Python. FPGA PS part is the autoencoder calculation using Python on the PYNQ project of the ARM CPU mounted on the PS part of the Zynq UltraScale + MPSoC ZCU104. Development method is the calculation process of this system. In the results of the development methodology, the processing time for 64 sets is only about twice as long as for one set alone. This is thought to be due to the processing time required for data input/output and the pipelined processing of the encoder and decoder. Another factor contributing to the higher speed of the development method compared to other methods is the processing in Python. The interpreter language converts the source code into machine language sequentially during execution, and the processing time is also included in the measurement, resulting in slower processing speed. However, it has the advantage of being very easy for users to handle in terms of data processing. For this reason, this development method, in which the main calculation part is performed in the PL part of the FPGA and the application part is processed in Python in the PS part, is considered to be effective. In addition, FPGA processing speeds are generally faster than those of compiler languages, even when compared to compiler languages, because FPGAs are arranged and wired specifically for the target computation process.

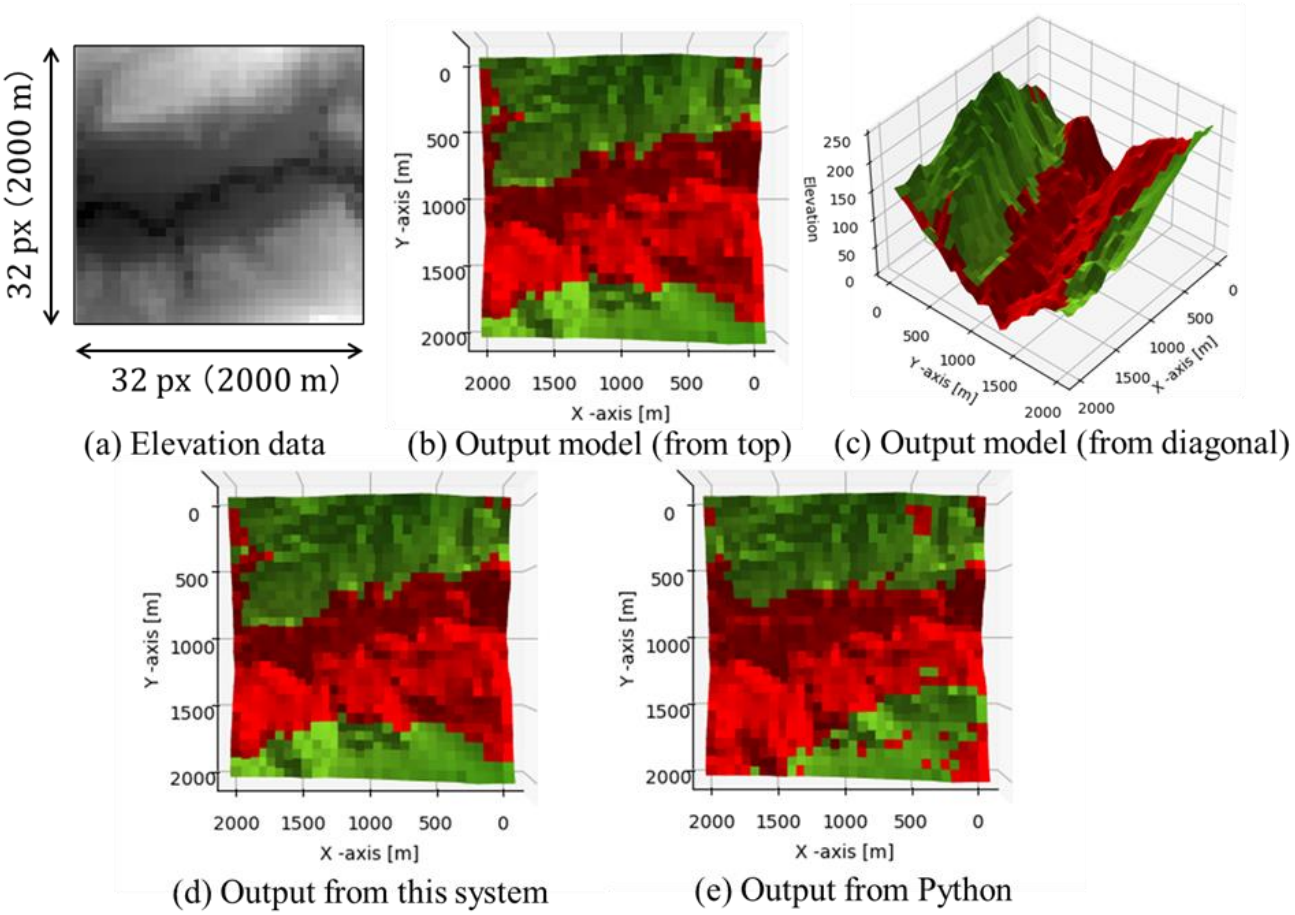


Fig. 15. Output results of hazard map for landslides

V. Output Results

This section discusses some of the output results obtained when this system is run and their contents. Figure 15 shows the results of learning a hazard map for a landslide disaster, running the circuit implemented in FPGA, and outputting the map as a 3D model. The figure shows the elevation data input to the circuit, the output results from the circuit created, and a comparison between them and the simulation results in Python. The areas painted red on the hazard map represent danger zones. These areas are where output greater than 0 was obtained after the autoencoder calculation and after passing the ReLU function. Observation of the output model shows that the hazardous area extends laterally around the slope of the cliff. In such geographic features, landslides may occur on the slopes. When the output model is viewed as a 3D model from an angle, the hazardous areas can be identified more clearly. Comparing the results of arithmetic processing in FPGA with the results of simulation in Python, it can be seen that the output results are generally equivalent. However, when observing the detailed range of the danger zones, errors arise from the simulations in Python. This is because the weights and biases used in the creation circuit are not precise enough, and improvement can be expected by using more precise data.

VI. Conclusion

The design theme of this year's LSI Design Contest was an auto encoder, and we decided to develop a hazard map system by pursuing practicality and originality in this theme. By pipelining, parallelizing, and adjusting the circuit size, we have developed an autoencoder system that infers a hazard map from geographic features and outputs a 3D model of the hazard. We are planning to further enhance and select datasets and to conduct learning that includes information other than geographic information in order to generate hazard maps with even higher accuracy. On the other hand, the design of the circuitry has been made with sufficient resource usage through various innovations, and we intend to further speed up the system and make it stand-alone by implementing a learning circuit and further parallelizing the multiplications.

References

[1] 千葉市役所：千葉市地震・風水害ハザードマップ (https://www.city.chiba.jp/other/jf_hazardmap/map.html?lay=saigai_12)

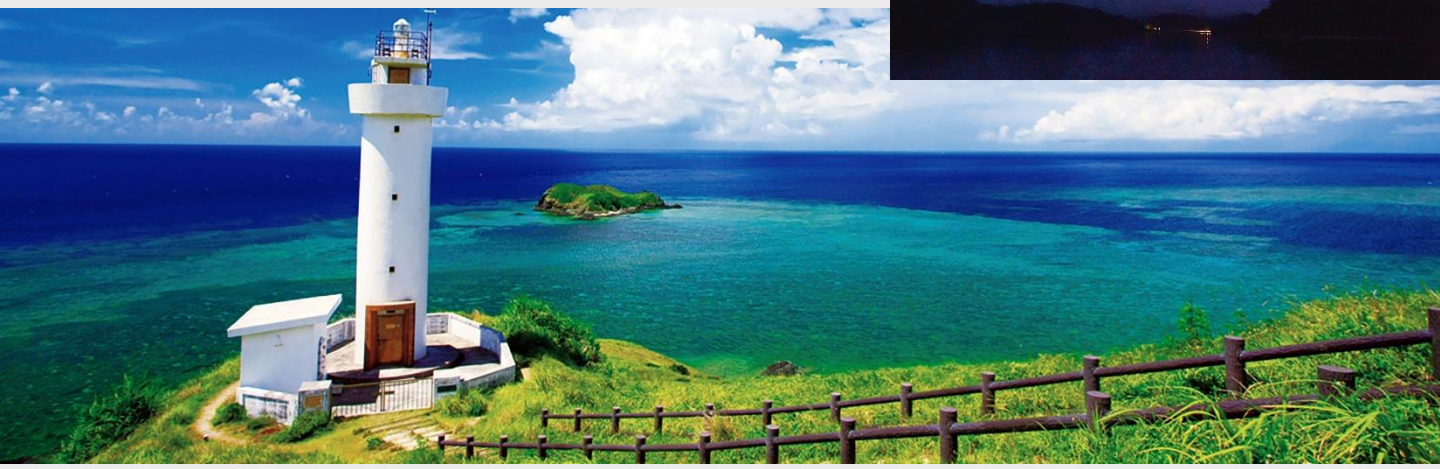
[2] 国土交通省：重ねるハザードマップ (https://disaportal.gsi.go.jp/)

[3] Vivado Design Suite: AXI リファレンス ガイド (UG1037) (https://docs.xilinx.com/v/u/ja-JP/ug1037-vivado-axi-reference-guide)

[4] PYNQ: PYTHON PRODUCTIVITY (http://www.pynq.io/)

また来年沖縄で、お会いしましょう。

OKINAWA



コンテストに関してのお問合せ：
九州工業大学情報工学部情報・通信工学科尾知研究室内
LSIデザインコンテスト実行委員会事務局

TEL：0948-29-7667
http://www.lsi-contest.com