

Politechnika Warszawska

WYDZIAŁ ELEKTRONIKI
I TECHNIK INFORMACYJNYCH



Instytut Systemów Elektronicznych

Praca dyplomowa magisterska

na kierunku Elektronika
w specjalności Mikrosystemy i Systemy Elektroniczne

Implementacja sztucznych sieci neuronowych w systemach
SoC i MPSoC z wykorzystaniem akceleratorów, realizowanych
w technice HLS

Krzysztof Wasilewski
Numer albumu 265956

promotor
dr inż. Wojciech Zabołotny

WARSZAWA 2020

Implementacja sztucznych sieci neuronowych w systemach SoC i MPSoC z wykorzystaniem akceleratorów, realizowanych w technice HLS

Streszczenie

Sztuczne Sieci Neuronowe są w dzisiejszych czasach wykorzystywane w wielu zastosowaniach. Duża złożoność algorytmów Sztucznej Inteligencji wymaga dużej mocy obliczeniowej oraz odpowiednich metod optymalizacji i właściwego wyboru sprzętu. W przypadku obliczeń, które można zrównoleglić, rozsądny wyborem są układy FPGA (ang. *Field Programmable Gate Array*).

Głównym celem pracy było zbadanie wydajności implementacji Sztucznych Sieci Neuronowych z wykorzystaniem akceleratorów, realizowanych w technice HLS (ang. *High Level Synthesis*). Dokonano porównania szybkości wykonywania obliczeń w systemie, zaimplementowanym z użyciem układu FPGA z rozwiązaniami programowymi, uruchamianymi na komputerze PC. Założeniem projektu było stworzenie akceleratora, umożliwiającego osiągnięcie wzrostu wydajności algorytmu detekcji obiektów znajdujących się na obrazie w czasie rzeczywistym.

Wykonano projekt implementacji Sztucznej Sieci Neuronowej przy wykorzystaniu systemu SoC (ang. *System on Chip*) z układem FPGA i techniki HLS. Użycie metody HLS pozwala na projektowanie przy wykorzystaniu języka C, C++ lub System C i umożliwia korzystanie z wielu bibliotek zaimplementowanych w języku C i C++, co znacznie przyspiesza pracę nad projektem. W ostatnich latach można zaobserwować wielki postęp w dziedzinie komputerowego rozpoznawania obrazów (ang. *Computer Vision*), jednak większość dostępnych implementacji jest przeznaczona do uruchomienia na komputerze PC.

Słowa kluczowe: Sztuczne Sieci Neuronowe, HLS, komputerowe rozpoznawanie obrazów, FPGA

Artificial Neural Networks implementation in SoC and MPSoC systems using accelerators synthesized by HLS method

Abstract

Nowadays, Artificial Neural Networks (ANN) are used in many applications. High complexity of Artificial Intelligence algorithms requires high computing power, appropriate optimization methods and efficient hardware. In the case of computation that is easy to parallelize it is reasonable to use FPGA systems.

The main aim of the thesis was to investigate the performance of Artificial Neural Networks implementation using accelerators synthesized by HLS (High-Level Synthesis) method. A performance comparison was made between implementation using FPGA and software solutions, tested on PC. Assumption of the project was to create an accelerator, that enables to achieve efficiency improvement in the real-time object detection algorithm.

The Artificial Neural Network implementation was made using SoC (System on Chip) with FPGA system and HLS method. The HLS enables to design a project using C, C++ or System C language and to use lots of C libraries, which makes working on the project faster. It has been seen recently, that huge improvement was made in the field of Computer Vision, but most of the available implementations are made to run on PC.

Keywords: Artificial Neural Networks, HLS, Computer Vision, FPGA



.....
miejscowość i data

.....
imię i nazwisko studenta

.....
numer albumu

.....
kierunek studiów

OŚWIADCZENIE

Świadomy/-a odpowiedzialności karnej za składanie fałszywych zeznań oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie, pod opieką kierującego pracą dyplomową.

Jednocześnie oświadczam, że:

- niniejsza praca dyplomowa nie narusza praw autorskich w rozumieniu ustawy z dnia 4 lutego 1994 roku o prawie autorskim i prawach pokrewnych (Dz.U. z 2006 r. Nr 90, poz. 631 z późn. zm.) oraz dóbr osobistych chronionych prawem cywilnym,
- niniejsza praca dyplomowa nie zawiera danych i informacji, które uzyskałem/-am w sposób niedozwolony,
- niniejsza praca dyplomowa nie była wcześniej podstawą żadnej innej urzędowej procedury związanego z nadawaniem dyplomów lub tytułów zawodowych,
- wszystkie informacje umieszczone w niniejszej pracy, uzyskane ze źródeł pisanych i elektronicznych, zostały udokumentowane w wykazie literatury odpowiednimi odnośnikami,
- znam regulacje prawne Politechniki Warszawskiej w sprawie zarządzania prawami autorskimi i prawami pokrewnymi, prawami własności przemysłowej oraz zasadami komercjalizacji.

Oświadczam, że treść pracy dyplomowej w wersji drukowanej, treść pracy dyplomowej zawartej na nośniku elektronicznym (płycie kompaktowej) oraz treść pracy dyplomowej w module APD systemu USOS są identyczne.

.....
czytelny podpis studenta

Spis treści

1. Wstęp	11
1.1. Wprowadzenie	11
1.2. Wstęp teoretyczny	11
1.2.1. Model neuronu	12
1.2.2. Funkcja aktywacji	13
1.2.3. Perceptron wielowarstwowy	15
1.2.4. Sieci Głębokie	15
1.2.5. Proces uczenia sieci	16
1.2.6. Wsteczna propagacja błędu	17
1.2.7. Hiperparametry	17
1.2.8. Przeuczenie sieci	17
1.2.9. Splotowe Sieci Neuronowe	17
1.3. Zastosowania Sztucznych Sieci Neuronowych	18
2. Cel i zakres pracy	21
2.1. Motywacja	21
2.2. Układy GPU i FPGA w zastosowaniach <i>Machine Learning</i>	21
2.3. Implementacja Sztucznych Sieci Neuronowych w układach FPGA	23
2.3.1. Uczenie Sztucznej Sieci Neuronowej w układach FPGA	23
3. Wybór sprzętu	25
3.1. Xilinx Zynq-7000	25
3.2. Z-turn Board	26
3.2.1. Interfejsy komunikacyjne	26
3.3. Kamera	27
3.3.1. Moduł MY-CAM011B	28
3.3.2. Moduł MY-CAM002B	29
4. Implementacja	33
4.1. Budowa systemu	33
4.1.1. Schemat blokowy systemu	33
4.2. Wybór narzędzi	33
4.3. Projekt systemu w środowisku Vivado	34
4.3.1. Pamięć Block RAM	34
4.4. Wykorzystanie metody HLS	35
4.5. Optymalizacja w Vivado HLS	37
4.5.1. Rozwijanie pętli	38
4.5.2. Pipeline	40
4.5.3. Dyrektywa <i>Dependence</i>	40
4.5.4. Partycjonowanie pamięci	41

4.5.5. Arytmetyka zmiennoprzecinkowa	41
4.6. Zbiór danych wejściowych	41
4.7. Opracowanie modelu ANN	41
4.7.1. Uczenie Sztucznej Sieci Neuronowej	42
4.7.2. Implementacja modelu przy użyciu narzędzia Vivado HLS	42
4.8. Synteza projektu w narzędziu Vivado	43
4.9. Implementacja przy użyciu narzędzia Vitis	44
4.10. Test z wykorzystaniem systemu operacyjnego Petalinux	45
4.10.1. Wstępna konfiguracja systemu Petalinux	46
4.10.2. Konfiguracja jądra systemu Petalinux	47
4.10.3. Konfiguracja systemu plików	49
4.10.4. Przygotowanie obrazu systemu	50
4.10.5. Uruchomienie systemu Petalinux	50
5. Wyniki i wnioski	53
5.1. Test modelu sieci z jedną warstwą ukrytą	53
5.1.1. Test klasyfikacji cyfr z użyciem kamery	53
5.1.2. Test na płytce Z-turn z użyciem kamery	54
5.2. Test modelu posiadającego dwie warstwy ukryte	56
5.3. Implementacja akceleratora obliczeń ANN w Vivado HLS	57
5.4. Dostosowanie parametrów w implementacji HLS	58
5.4.1. Optymalizacja pętli	58
5.4.2. Dobranie parametru H	58
5.5. Arytmetyka stałoprzecinkowa	58
5.6.	60
5.6.1. Wyniki po wykonaniu optymalizacji HLS	60
6. Podsumowanie	65
6.1. Wnioski dotyczące techniki HLS	65
6.2. Możliwości rozwoju projektu	65
Bibliografia	67
Wykaz symboli i skrótów	69
Spis rysunków	70
Spis tabel	71

1. Wstęp

1.1. Wprowadzenie

W ostatnich latach można zaobserwować gwałtowny rozwój w dziedzinie Uczenia Maszynowego (ang. *Machine Learning*) i Sztucznej Inteligencji (ang. *Artificial Intelligence*). Coraz więcej urządzeń i systemów projektowanych jest w sposób zapewniający działanie autonomicznie, bez potrzeby ingerencji człowieka. Jednym z algorytmów, który powstał już dość dawno, są Sztuczne Sieci Neuronowe (ang. *Artificial Neural Networks*). Temat Sieci Neuronowych ma długą historię rozwoju, sięgającą początku lat 40. XX wieku, jednak w ostatnich latach można zaobserwować znaczny postęp w tej dziedzinie [1]. Rozwój technologii umożliwił zastosowanie algorytmów AI w wielu aplikacjach. Działalność naukowa w kierunku Sztucznych Sieci Neuronowych spowodowała powstanie nowych modeli i architektur.

Większość algorytmów wykorzystujących Sztuczną Inteligencję wymaga dużej mocy obliczeniowej i wyboru odpowiedniego sprzętu. Często powtarzaną operacją matematyczną w przypadku algorytmu Sztucznej Sieci Neuronowej jest mnożenie macierzy. Działanie to można w łatwy sposób zrównoleglić, implementując sieć w układzie FPGA i tym samym zwiększyć efektywność algorytmu.

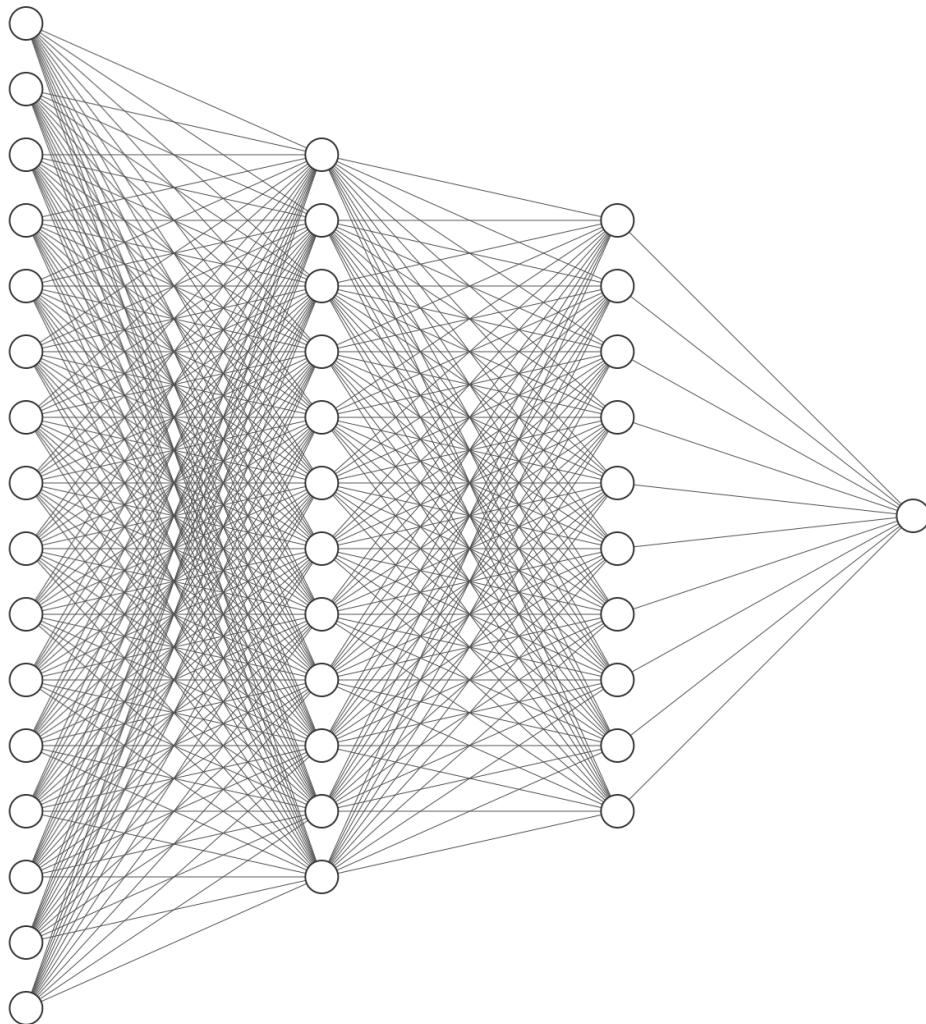
Praca nad implementacją algorytmów Sztucznej Inteligencji w większości przypadków zaczyna się od stworzenia i uruchomienia modelu. Do tego zadania często wykorzystywane są wysokopoziomowe biblioteki takie jak *keras* lub *Theano*, które w znacznym stopniu przyspieszają proces tworzenia oprogramowania oraz ułatwiają wprowadzanie zmian w modelu sieci. Rozwój i dopracowywanie algorytmu Sztucznej Inteligencji wymaga wielu iteracji uruchamiania kodu z różnymi parametrami i właściwościami sieci. Aby w pełni wykorzystać potencjał Sztucznych Sieci Neuronowych, należy dobrać odpowiednio metody projektowania i testowania modeli.

1.2. Wstęp teoretyczny

Sieć neuronowa jest algorytmem przetwarzającym dane, inspirowanym działaniem mózgu. ANN składa się z wielu elementów przetwarzających informacje – neuronów. Model przetwarzania informacji jest inspirowany ludzkim mózgiem jednak w stosunku do komórki nerwowej, neuron w Sztucznej Sieci Neuronowej jest bardzo uproszczony [2]. Pomimo to ANN umożliwiają rozwiązywanie złożonych problemów w wielu dziedzinach z dużą dokładnością.

Pomiędzy neuronami prowadzone są połączenia, z których każde ma swoją wagę, która zmienia wartość w trakcie uczenia sieci. Warstwę sieci, w której wszystkie wyjścia każdego z neuronów są podłączone do wszystkich neuronów w warstwie następnej, nazywa się warstwą w pełni połączoną (ang. FC — *Fully Connected*). Sieć posiadająca wszystkie warstwy w pełni połączone jest zwana siecią w pełni połączoną.

W znacznej większości sieci neuronowe budowane są w ten sposób, że dane są propagowane w kierunku od warstwy wejściowej do wyjściowej. Takie sieci nazywamy sieciami jednokierunkowymi (ang. *feedforward*). Przykładową sieć neuronową w pełni połączoną typu *feedforward* przedstawiono na Rys. 1.1.



Rysunek 1.1. Schemat w pełni połączonej jednokierunkowej Sieci Neuronowej (wygenerowany na stronie: <http://alexlenail.me/NN-SVG/>)

1.2.1. Model neuronu

Sztuczne Sieci Neuronowe to algorytm inspirowany działaniem ludzkiego mózgu i znajdujących się w nim neuronów. Model matematyczny (Rys. 1.2) pojedynczego neuronu składa się z następujących elementów [3]:

- wektora wejściowego:

$$x = (x_1, x_2, \dots, x_j)^T$$

- wektora wag przypisanych do każdego z wejść

$$\mathbf{w} = (w_{k1}, w_{k2}, \dots, w_{km})^T$$

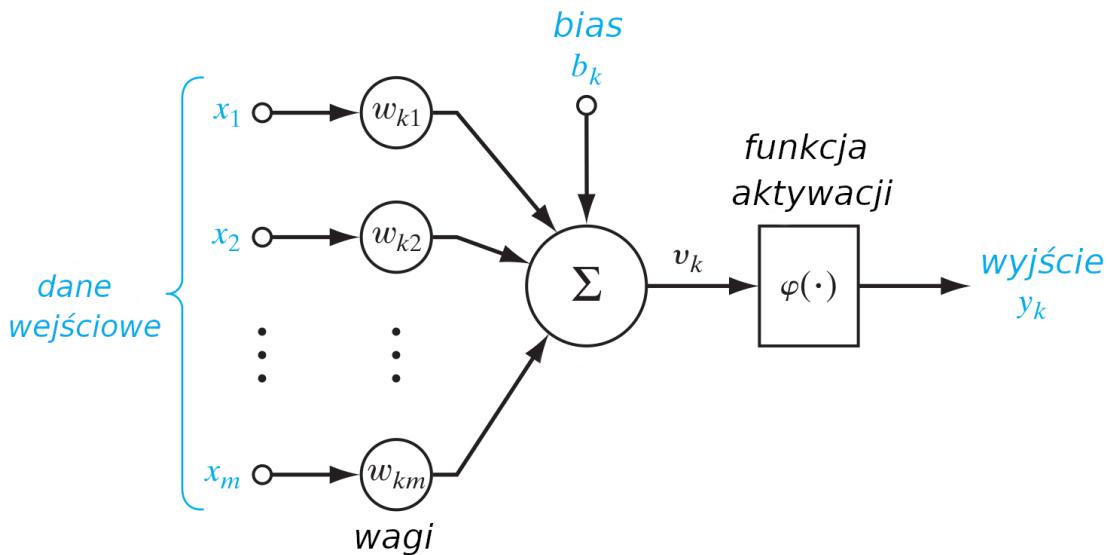
- wyrazów wolnych (bias) b
- funkcji aktywacji $\phi(u)$
- wyjścia neuronu y .

Wyrazy wolne (bias) umożliwiają kontrolowanie progu aktywacji neuronu, niezależnie od wartości wejściowych. W praktyce oznacza to przesuwanie wykresu funkcji aktywacji w zależności od znaku w prawą lub w lewą stronę. Wyjście neuronu można policzyć, stosując następujące wzory:

$$u_k = \sum_{j=1}^N w_{kj} x_j$$

$$y_k = \phi(u_k + b_k)$$

Najprostsza Sieć Neuronowa składająca się z jednego neuronu, nazywana jest *Perceptronem*. Algorytm ten można wykorzystać do zadań klasyfikacji binarnej.



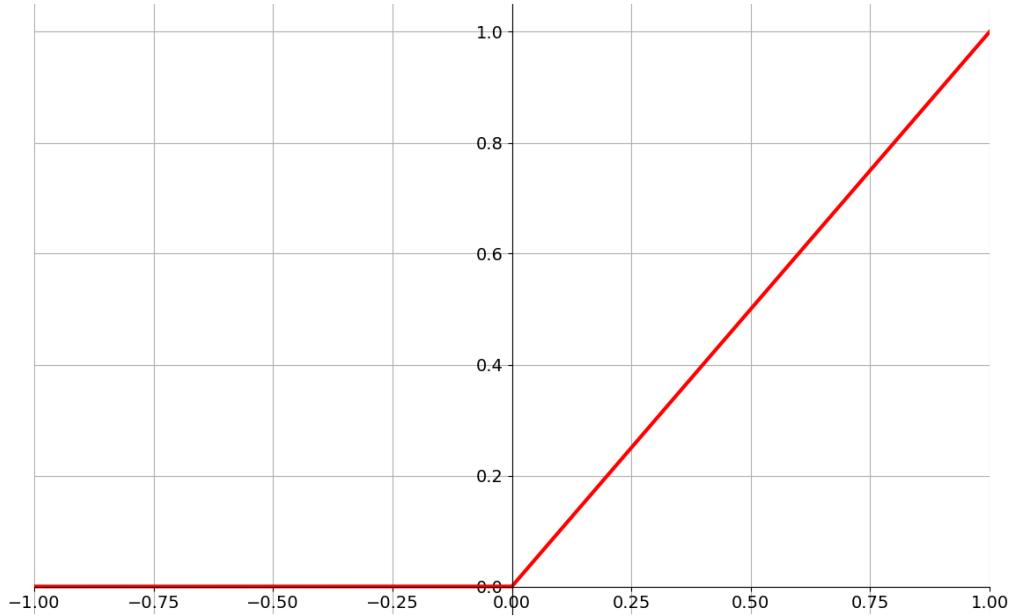
Rysunek 1.2. Model neuronu [4]

1.2.2. Funkcja aktywacji

Na działanie algorytmu znaczny wpływ może mieć dobór odpowiedniej funkcji aktywacji. Wśród najczęściej stosowanych funkcji aktywacji wyróżnia się następujące funkcje:

- funkcja ReLu (ang. *Rectified Linear Units*):

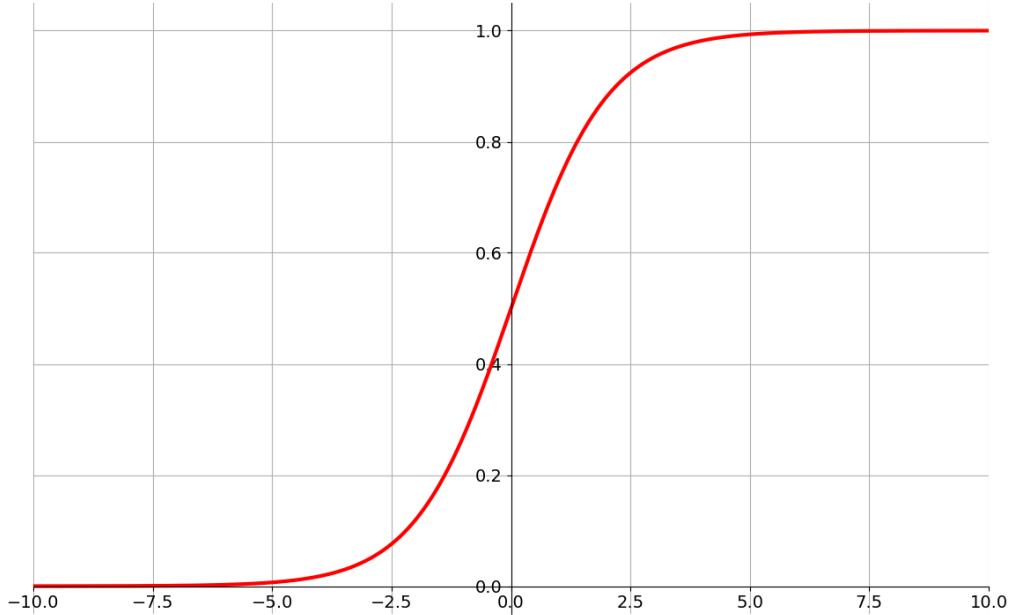
$$\phi(x) = \max(0, x)$$



Rysunek 1.3. Wykres funkcji ReLU

- funkcja sigmoid:

$$\phi(x) = \frac{a}{a + e^{-bx}}$$



Rysunek 1.4. Wykres funkcji sigmoid

- funkcja softmax (liczona dla każdego neuronu warstwy wyjściowej, $j = 1, \dots, N$):

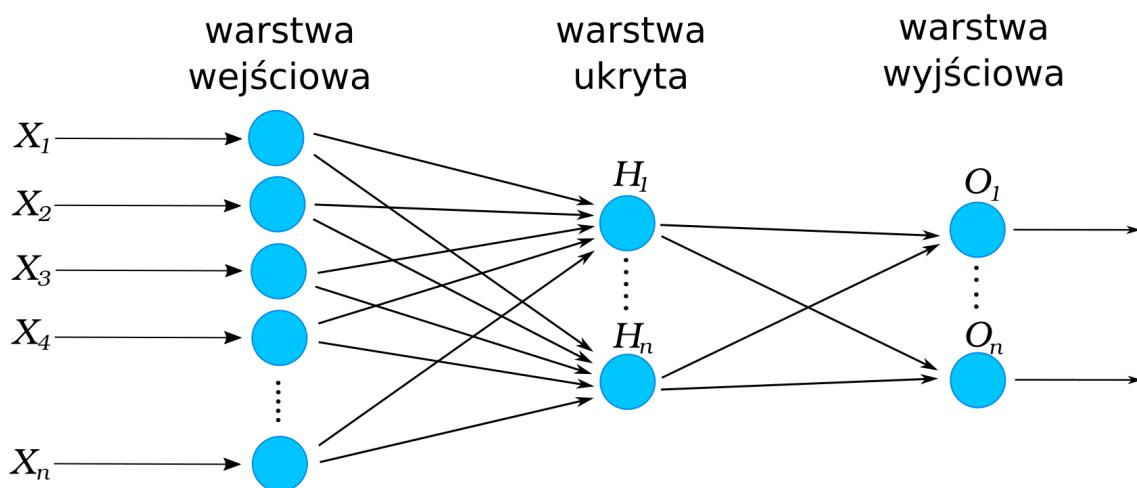
$$\phi(x_j) = \frac{e^{x_j}}{\sum_{k=1}^N e^{x_k}}$$

W warstwach ukrytych często używaną funkcją aktywacji jest funkcja ReLU oraz sigmoid. Funkcję sigmoid stosuje się również w warstwach wyjściowych przy zadaniach klasyfikacji binarnej. Funkcja softmax jest używana w problemach klasyfikacji wieloklasowej (np. rozpoznawanie cyfr). Stosując funkcję softmax w warstwie wyjściowej, można traktować wektor wartości wyjść sieci jako rozkład prawdopodobieństwa wyboru danego rozwiązania.

1.2.3. Perceptron wielowarstwowy

Jednym z pierwszych modeli Sztucznych Sieci Neuronowych był Perceptron Wielowarstwowy (ang. MLP — *Multi-Layer Perceptron*), składający się z warstwy neuronów wejściowej, ukrytych i wyjściowej (Rys.1.5). Wyjście neuronów w danej warstwie staje się wejściem neuronów warstwy następnej.

Sieć Neuronowa typu MLP najczęściej posiada warstwy w pełni połączone (ang. FC – *Fully Connected*) lub inaczej Gęste (ang. *Dense*). W warstwie FC każde z wyjść jest podłączone do wszystkich wejść neuronów w warstwie następnej.



Rysunek 1.5. Model Perceptronu Wielowarstwowego

1.2.4. Sieci Głębokie

Rozwój algorytmów AI doprowadził do powstania Głębokich Sieci Neuronowych (ang. DNN — *Deep Neural Networks*), czyli takich, które posiadają wiele warstw ukrytych [5]. Algorytm Ucznia Głębokiego (ang. *Deep Learning*) umożliwia rozwiązywanie skompliko-

wanych problemów takich jak rozpoznawanie i klasyfikację obiektów na obrazie (Rys.1.6). Seria warstw ukrytych (ang. *hidden layers*) umożliwia ekstrakcję cech obiektów. Kolejne warstwy umożliwiają wykrywanie krawędzi, potem konturów, a na końcu całych kształtów i obiektów.



Rysunek 1.6. Model Głębokiego Uczenia sieci [5]

1.2.5. Proces uczenia sieci

Uczenie Sztucznej Sieci Neuronowej można podzielić na nadzorowane (ang. *supervised learning*) i nienadzorowane (ang. *unsupervised learning*) oraz wersję pośrednią – uczenie częściowo nadzorowane (ang. *semi-supervised learning*). W praktyce najczęściej wykorzystywanym rodzajem uczenia jest *supervised learning*, jednak uczenie nienadzorowane, które zakłada tworzenie modelu sieci bez ingerencji człowieka, jest ciekawym tematem badań i rozwoju algorytmów Sztucznych Sieci Neuronowych [6]. Proces uczenia nadzorowanego sieci polega na zwiększaniu dokładności ANN w rozwiązywaniu danego problemu. Odbywa się to poprzez iteracyjne korygowanie wartości wag i wyrazów wolnych (bias) tak, aby zminimalizować błąd pomiędzy rozwiązaniem wzorcowym i otrzymanym. Jeden z algorytmów zmiany wag podczas uczenia jest reguła delty (ang. *delta rule*):

$$\Delta w_{kj} = \eta e_k(n)x_j(n)$$

$$e_k(n) = d_k(n) - y_k(n)$$

W powyższym wzorze $d_k(n)$ jest zakładanym rozwiązaniem, a $y_k(n)$ rozwiązaniem otrzymanym. Wartość η oznacza współczynnik szybkości uczenia (ang. *learning rate*). Gdy jest wystarczająco mały, algorytm uczenia osiąga rozwiązanie optymalne, jednak większy współczynnik η może przyspieszyć ten proces. Wpływ na szybkość uczenia ma również wielkość miniserii (ang. *mini-batch*) uczących. Jest to liczba elementów zbioru uczącego, podawanych na wejście algorytmu, przed zaktualizowaniem wartości wag sieci. Z powodu ograniczonych rozmiarów zboru uczącego, jest on wykorzystywany wielokrotnie. Jedno przejście przez wszystkie próbki ze zbioru uczącego nazywane jest epoką (ang. *epoch*) [7].

1.2.6. Wsteczna propagacja błędu

Architekturę sieci MLP często stosuje się wraz z algorytmem wstecznej propagacji błędu (ang. *Backpropagation*), która umożliwia proces uczenia sieci. Poprzez obliczenie błędu w neuronach warstwy wyjściowej i propagacji wstecz błędu przez całą sieć, algorytm pozwala dostosować wartość wagi każdego z połączeń w taki sposób, aby zminimalizować wartość funkcji kosztu. Sieć jest uczona do momentu, gdy błąd stanie się akceptowalnie mały.

1.2.7. Hiperparametry

W procesie uczenia Sztucznej Sieci Neuronowej bardzo istotne jest odpowiednie ustalenie parametrów sieci, zwanych hiperparametrami (ang. *hyperparameters*). Są to parametry sieci, ustawiane przez użytkownika w celu otrzymania modelu, dającego najlepsze rozwiązania. Na wynik uczenia wpływ mają m.in. następujące Hiperparametry:

- liczba i rodzaj warstw sieci
- liczba neuronów w każdej z warstw
- współczynnik szybkości uczenia η
- wielkość miniserii
- liczba epok.

1.2.8. Przeuczenie sieci

Przy używaniu Sztucznych Sieci Neuronowych w zadaniach klasyfikacji często spotykanym problemem jest przeuczenie sieci (ang. *overfitting*). Przeuczona sieć jest zbyt mocno dopasowana do danych uczących i przy walidacji poprawność klasyfikacji jest znacznie mniejsza. Jednym ze sposobów na polepszenie zdolności do generalizacji trenowanego modelu sieci jest zastosowanie *Dropoutu*, czyli losowego usuwania pewnej ustalonej liczby neuronów. Technika ta, pomimo swojej prostoty, daje bardzo dobre wyniki w przeciwdziałaniu przeuczeniu sieci.

1.2.9. Splotowe Sieci Neuronowe

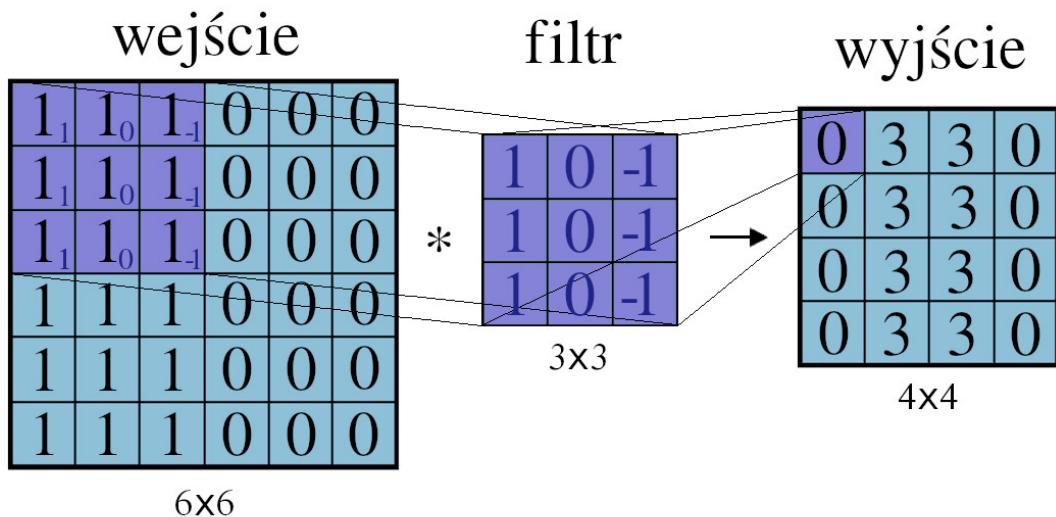
Splotowe Sieci Neuronowe (ang. *CNN – Convolutional Neural Networks*) są typem sieci głębokich, które znakomicie nadają się do rozpoznawania kształtów i z powodzeniem sto-

sowane w wielu zagadnieniach klasyfikacji obrazów. Charakteryzują się dużą odpornością na rotację, skalowanie, zniekształcanie i inne zakłócenia [4]. Sieci te są uczone w trybie nadzorowanym.

Za ekstrakcję cech obiektów w dwuwymiarowym obrazie odpowiedzialna jest warstwa splotowa sieci. Wejście M tej warstwy jest przekształcane przy użyciu odpowiedniego filtra K . Operację splotu w sieciach CNN można opisać wzorem:

$$S(i, j) = (M * K)(i, j) = \sum_k \sum_j M(i - k, j - l)K(k, l)$$

Wynik $S(i, j)$ nazywany jest mapą atrybutów (ang. *feature map*). Warstwa splotowa umożliwia zmniejszenie rozmiaru przetwarzanych danych podczas uczenia sieci oraz zwiększa jej zdolność do generalizacji. Operację splotu przedstawiono na Rys. 1.7. Zaprezentowany przypadek to splot macierzy 6x6 przy użyciu filtra o rozmiarze 3x3.

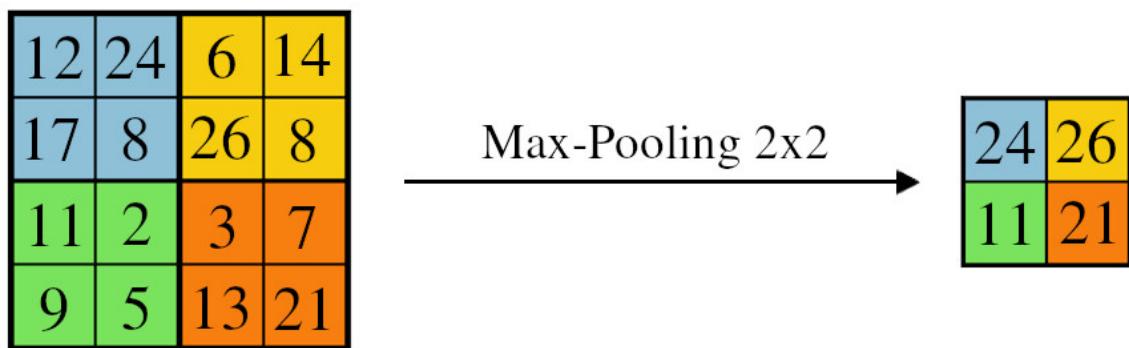


Rysunek 1.7. Splot macierzy 6x6 przy użyciu filtru o rozmiarze 3x3

Razem z warstwą splotową często stosowana jest warstwa grupowania (ang. *pooling*), występującą najczęściej w jednym z dwóch wariantów (ang. *max-pooling*) lub (ang. *average pooling*). W przypadku przetwarzania obrazu metoda (ang. *max-pooling*) polega na zmniejszeniu jego rozmiaru, poprzez wybranie wartości maksymalnej lub średniej, stosując okno o wybranym rozmiarze. Najczęściej wybieranym krokiem (ang. *stride*) przesuwania okna jest wartość, równa jego rozmiarowi. Na Rys. 1.8 przedstawiono przykład *max-poolingu* o rozmiarze 2x2 ze skokiem równym rozmiarowi okna.

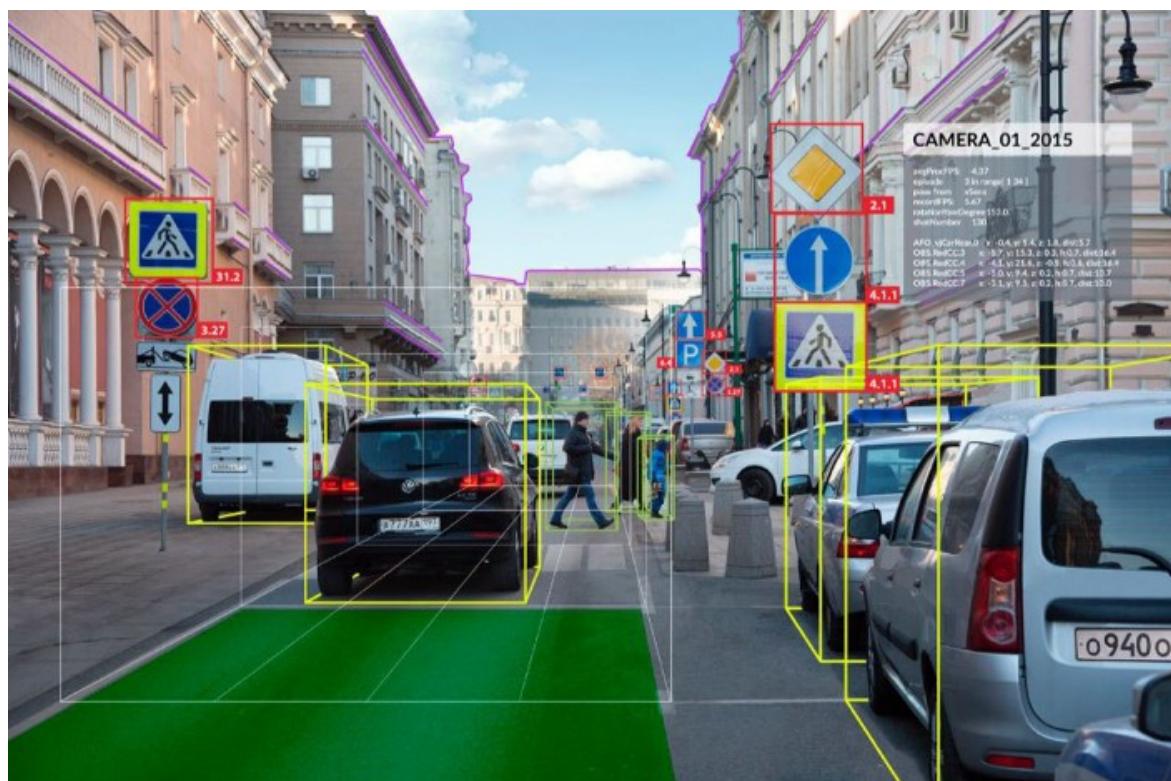
1.3. Zastosowania Sztucznych Sieci Neuronowych

W dzisiejszych czasach Sztuczne Sieci Neuronowe są wykorzystywane w wielu zastosowaniach. Dziedziny, w których stosuje się algorytmy ANN to m.in.:



Rysunek 1.8. Max-pooling o rozmiarze 2x2

- medycyna – wspomaganie wystawiania diagnozy
- przemysł – automatyczna kontrola jakości produktów
- zastosowania militarne – wspomaganie radarów i termowizji
- ekonomia – przewidywanie notowań giełdowych
- robotyka – autonomiczne samochody (Rys. 1.9)
- rozpoznawanie i synteza mowy – sterowanie różnymi systemami (smartfon, komputer, urządzenia IoT) przy użyciu asystenta głosowego



Rysunek 1.9. Rozpoznawanie obiektów na obrazie z kamery samochodu autonomicznego [8]

2. Cel i zakres pracy

Celem pracy było zaprojektowanie i implementacja sztucznej sieci neuronowej przy wykorzystaniu systemu SoC (ang. System on Chip) i techniki HLS. Użycie metody HLS umożliwia projektowanie przy wykorzystaniu języka C, C++ lub System C oraz co zazwyczaj skraca czas wykonania projektu. Dodatkowo HLS umożliwia korzystanie z wielu bibliotek, zawierających funkcje, wykorzystywane w implementacji Sztucznych Sieci Neuronowych. Oczekiwany wynikiem pracy było stworzenie akceleratora, umożliwiającego osiągnięcie wzrostu wydajności w stosunku do rozwiązań software'owych uruchamianych na komputerze PC.

2.1. Motywacja

Sztuczne Sieci Neuronowe są związane z dużą ilością obliczeń, które mogą być wykonywane równolegle. Pozwala to osiągnąć krótszy czas wykonania programu, co ma duże znaczenie dla zastosowań w systemach działających w czasie rzeczywistym np. w branży *Automotive*. Aby osiągnąć przyspieszenie obliczeń, stosuje się różne metody. Jednym z najpopularniejszych obecnie sposobów na zwiększenie wydajności algorytmów AI jest wykorzystanie kart graficznych GPU (ang. *Graphics Processing Unit*). Metodą najbardziej przyspieszającą obliczenia, lecz wymagającą najdłuższego czasu projektowania i najbardziej kosztowną, jest zastosowanie specjalizowanych układów ASIC (ang. *Application-Specific Integrated Circuit*). Opcją pośrednią pomiędzy powyższymi dwoma rozwiązaniami jest zastosowanie układów FPGA. To podejście umożliwia osiągnięcie znacznego przyspieszenia wykonywania obliczeń i nie powoduje wielkiego wzrostu kosztów. Dodatkowo zastosowanie metody HLS ułatwia i minimalizuje czas tworzenia sprzętowej implementacji modelu ANN oraz wprowadzanie zmian w projekcie. Implementacja danego algorytmu może być optymalizowana pod kątem zużycia zasobów oraz wprowadzanych opóźnień przy użyciu dyrektyw kompilatora Vivado HLS. Dzięki temu programista posiadający minimalną wiedzę o sprzęcie jest w stanie utworzyć implementację dającą zadowalające rezultaty.

2.2. Układy GPU i FPGA w zastosowaniach *Machine Learning*

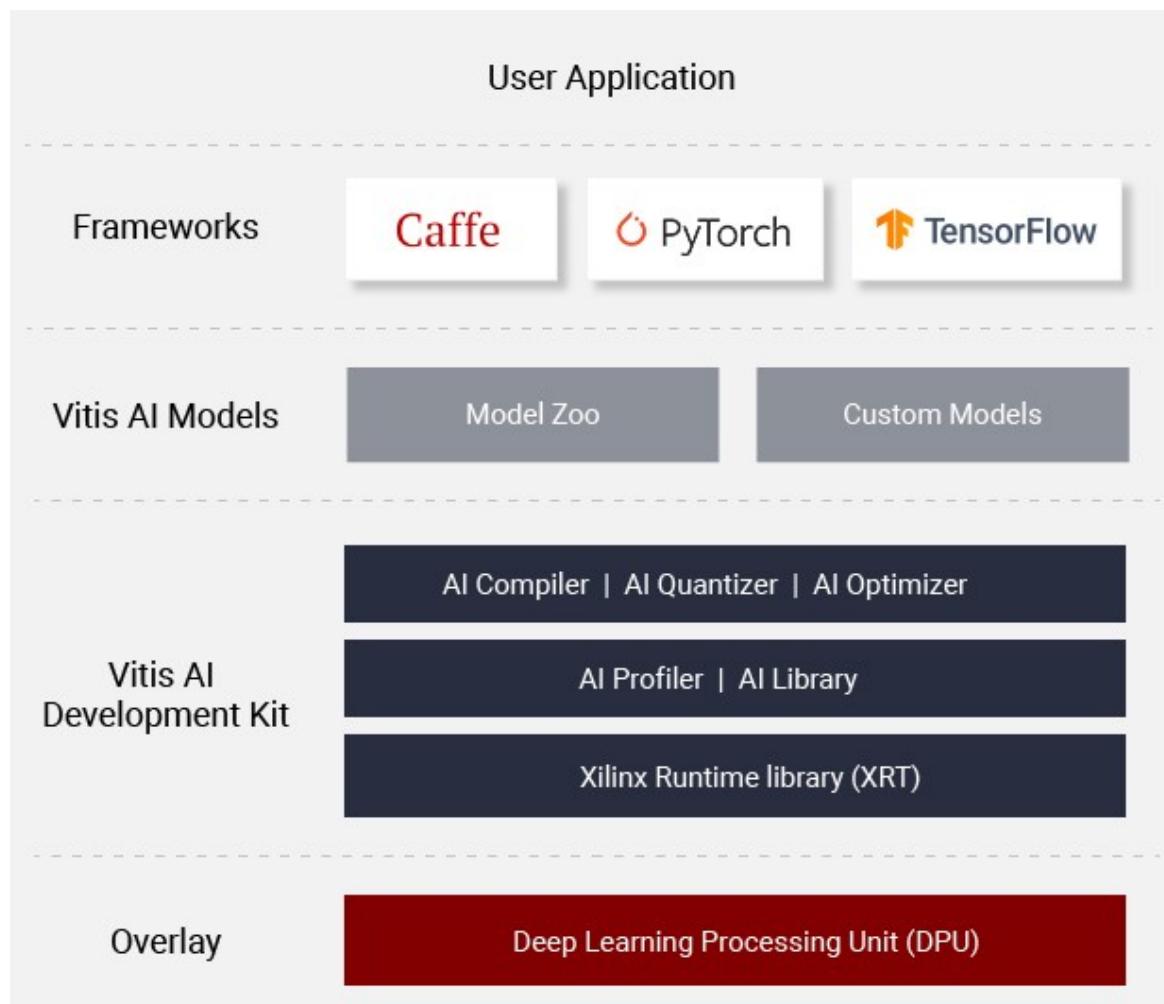
W większości przypadków Sztuczne Sieci Neuronowe są projektowane, uczone i uruchamiane na procesorze z akceleratorem obliczeń w postaci GPU. Jednak wydajne układy graficzne nie zawsze są dostępne w systemach wbudowanych oraz wprowadzają dodatkowe koszty systemu i duże zużycie mocy.

Alternatywnym sposobem na przyspieszenie obliczeń związanych z implementacją algorytmów ANN jest zastosowanie układów FPGA. Dodatkowym atutem po stronie układów FPGA jest mniejsze zużycie mocy. Dzięki odpowiednim metodom optymalizacji,

2. Cel i zakres pracy

przy wykorzystaniu układów FPGA można osiągnąć znacznie mniejsze zużycie zasobów i przyspieszenie obliczeń.

Przewagą zastosowania układów GPU jest łatwość implementacji skomplikowanych obliczeń dzięki zastosowaniu architektury *CUDA* (ang. *Compute Unified Device Architecture*), opracowanej przez firmę *Nvidia*. Dzięki CUDA, zrównoleglenie obliczeń jest możliwe przy użyciu języków programowania takich jak C, C++, Fortran, Python i MATLAB [9] i nie wymaga tyle wiedzy o sprzęcie co użycie metody HLS. Pomimo to technika HLS jest cały czas rozwijana. Powstają narzędzia dedykowane do zastosowań akceleracji obliczeń w Sztucznych Sieciach Neuronowych. W trakcie pisania tej pracy pojawiło się nowe narzędzie producenta *Xilinx* – *Vitis AI*, zapewniające wsparcie w tworzeniu rozwiązań ML z użyciem układów FPGA. *Vitis AI* jest zbiorem bibliotek i API (ang. *Application Programming Interface*) stworzonym do zastosowań wydajnego uruchamiania algorytmów AI [10]. To pokazuje, że synteza wysokiego poziomu w układach FPGA charakteryzuje się pozytywną tendencją rozwoju.



Rysunek 2.1. Struktura narzędzia Vitis-AI [11]

2.3. Implementacja Sztucznych Sieci Neuronowych w układach FPGA

Algorytmy wykorzystujące Sztuczne Sieci Neuronowe są dziś wykorzystywane w wielu zastosowaniach. Rozwój dziedziny Deep Learning spowodował powstanie modeli sieci zawierających wiele parametrów. Co za tym idzie, wzrosło zapotrzebowanie na metody akceleracji obliczeń w Sztucznych Sieciach Neuronowych. Większość obliczeń w ramach danej warstwy można zrównoleglić, więc zastosowania układów FPGA daje duże możliwości optymalizacji.

Najczęściej spotykanym podejściem jest zastosowanie ANN w układach FPGA do przyspieszenia obliczeń związanych z wnioskowaniem (ang. *inference*) nauczonej wcześniej sieci na nowych danych w czasie rzeczywistym. Model jest uczyony na komputerze PC przy użyciu bibliotek takich jak *keras*, *Caffe* czy *PyTorch*, najczęściej z wykorzystaniem GPU. Po przeprowadzeniu walidacji model sieci jest eksportowany, odpowiednio przetwarzany i implementowany w układzie FPGA.

2.3.1. Uczenie Sztucznej Sieci Neuronowej w układach FPGA

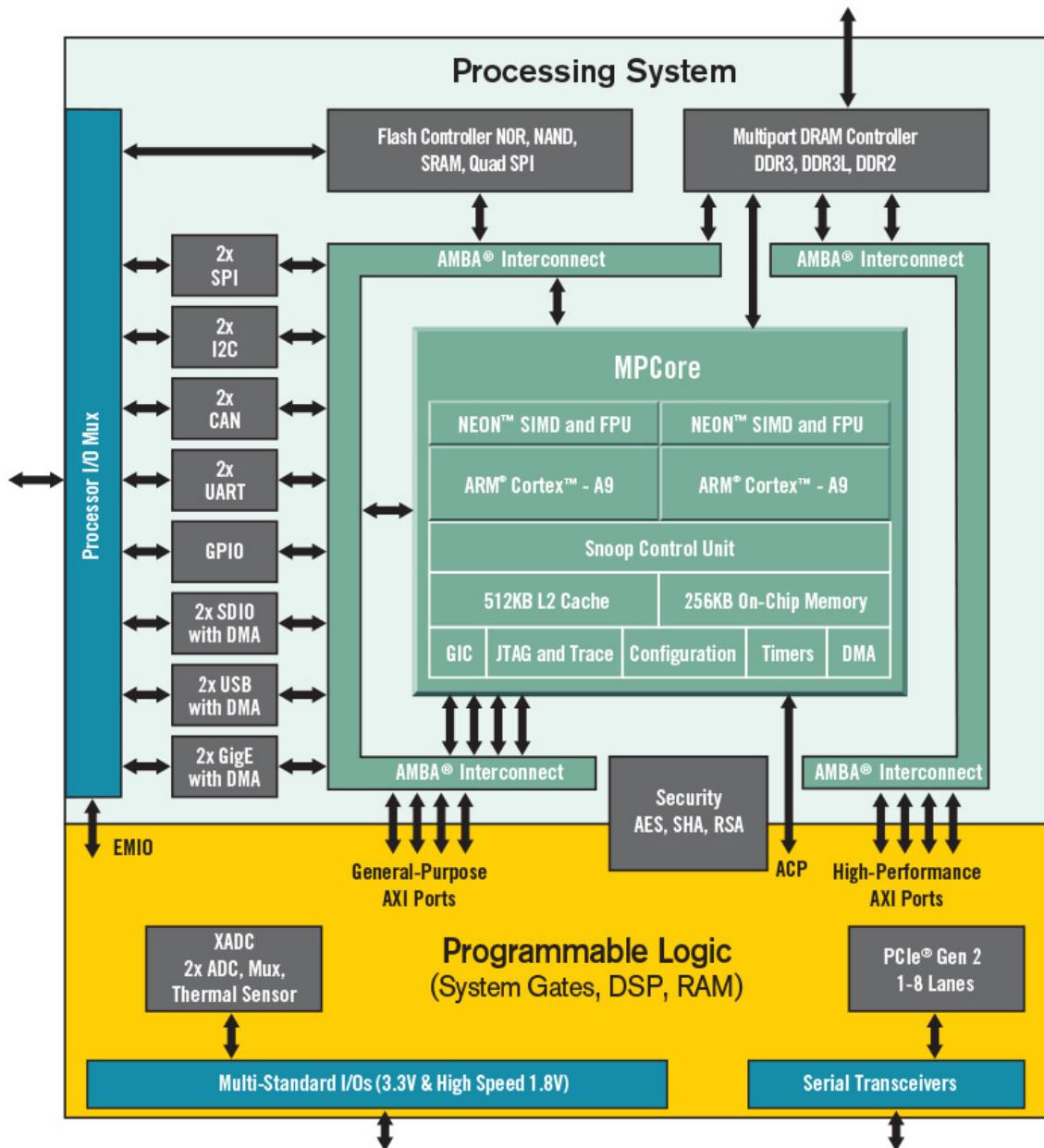
Istnieje również możliwość zaimplementowania algorytmu uczenia ANN na układzie FPGA. Takie rozwiązanie pozwala na zrównoleglenie obliczeń, które jest istotne w przypadku wielu epok uczenia sieci, jednak wiąże się z następującymi problemami:

- algorytm uczenia wymaga dużych zasobów pamięci
- skomplikowana implementacja algorytmu utrudnia wprowadzanie zmian w architekturze sieci
- implementacja modelu ze zmiennymi parametrami utrudnia optymalizację w HLS

3. Wybór sprzętu

3.1. Xilinx Zynq-7000

Przy wyborze płytki wzięto pod uwagę posiadane interfejsy oraz cenę sprzętu i dostępność dokumentacji. Podjęto decyzję o zastosowaniu układu FPGA firmy *Xilinx* z racji na duże wsparcie techniczne. W celu optymalizacji kosztów systemu wybrano podstawowe dwa układy z rodziny *Zynq-7000 SoC Family* (Rys. 3.1) XC7Z010 oraz XC7Z020. Aby mieć pewność, że parametry układu będą wystarczające, wybrano układ XC7Z020.



Rysunek 3.1. Architektura serii ZYNQ-7000 SoC [12]

3. Wybór sprzętu

Zgodnie z porównaniem zawartym w Tabeli 3.1, płytka Z-turn Board MYS-7Z020-C-S ma cenę sporo mniejszą od innych płytEK z układem XC7Z020. Na stronie producenta można znaleźć pełną dokumentację dotyczącą płytEK Z-turn. Wadą płytEK jest małe zaangażowanie społeczności w projekty z jej wykorzystaniem, jednak tak konkurencyjna cena ułatwia podjęcie wyboru.

Tabela 3.1. Porównanie cen płytEK z układami Zynq firmy Xilinx

Nazwa płytEK	Układ SoC	Cena
Z-turn Board MYS-7Z010-C-S	XC7Z010-1CLG400C	99\$ ¹
Z-turn Board MYS-7Z020-C-S	XC7Z020-1CLG400C	119\$ ¹
Zybo Z7-10 Development Board	XC7Z010-1CLG400C	199\$ ²
Zybo Z7-20 Development Board	XC7Z020-1CLG400C	299\$ ²
ZedBoard Zynq-7000	XC7Z020-CLG484-1	449\$ ³

3.2. Z-turn Board

Z-turn Board (Rys. ?? jest komputerem jednopłytkowym (ang. SBC – *Single Board Computer*), opartym o układ SoC Xilinx Zynq-7020 (XC7Z020-1CLG400C), zawierającym dwurdzeniowy procesor ARM Cortex-A9 (866MHz) i układ FPGA Artix 7. Parametry układu FPGA przedstawiono w Tabeli 3.2.

Tabela 3.2. Specyfikacja techniczna układu Artix-7

Logic Cells	LUT	Flip-Flop	BRAM (ilość bloków po 36Kb)	DSP
85K	53200	106400	4.9Mb (140)	220

Producentem płytEK jest firma MYIR Tech Limited (ang. *Make Your Ideas Real*), dostarczająca sprzęt bazujący na procesorach ARM oraz oprogramowanie do swoich produktów[13]. Biorąc pod uwagę parametry, płytKA charakteryzuje się wysokim stosunkiem ceny do jakości, podstawowa wersja kosztuje 99\$. Dla porównania płytKA Zybo Z7-20 kosztuje 199\$. Zaletą płytEK firmy *Digilent* (ZedBoard i Zybo) jest duże wsparcie producenta i spora ilość materiałów szkoleniowych.

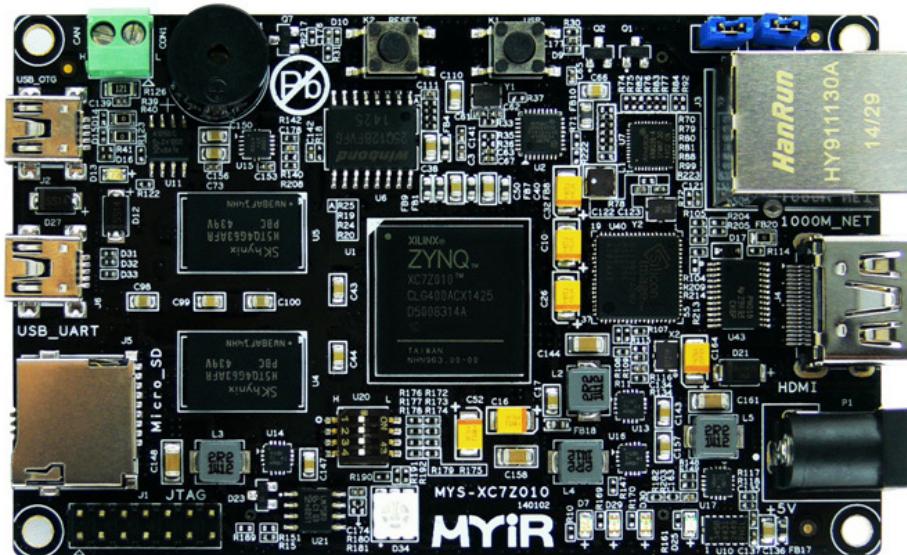
3.2.1. Interfejsy komunikacyjne

PłytkA Z-turn posiada interfejsy UART oraz Ethernet, które zostały wykorzystane do komunikacji komputera PC z systemem przy użyciu portu szeregowego i protokołu SSH (ang. *Secure Shell*). Istnieje również możliwość podłączenia wyświetlacza bezpośrednio do płytEK przy użyciu portu HDMI oraz innych peryferiów przy użyciu portu USB. Dodatkowo producent oferuje płytKE rozszerzeniową Z-turn IO-Cape (Rys. 3.3), która zawiera porty do podłączenia kamery przez protokół DVP (ang. *Digital Video Port*) oraz wyświetlacza LCD.

¹ <http://www.myirtech.com/list.asp?id=502>

² <https://store.digilentinc.com/zybo-z7-zynq-7000-arm-fpga-soc-development-board/>

³ <https://store.digilentinc.com/zedboard-zynq-7000-arm-fpga-soc-development-board/>



Rysunek 3.2. Płytki Z-turn-Board 7020 [14]

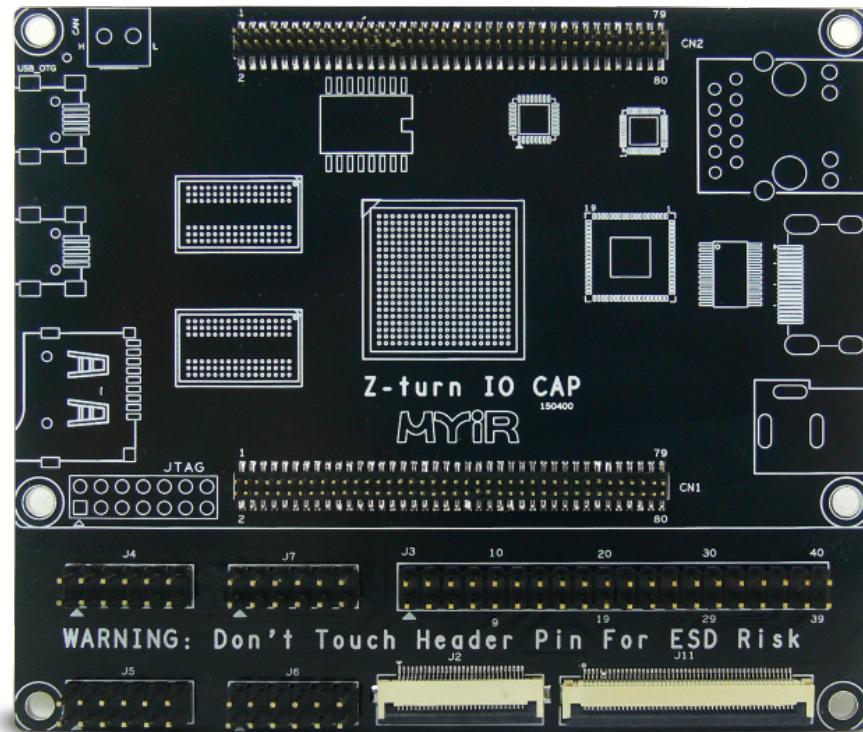
3.3. Kamera

Aby przetestować działanie systemu w czasie rzeczywistym, wykorzystano zewnętrzny moduł kamery. Przy wyborze sprzętu istotna była ceną modułu, interfejs komunikacji oraz kompatybilność z SBC Z-turn Board. W przypadku wykorzystania kamery w czasie rzeczywistym bardzo ważne jest niskie opóźnienie w wysyłaniu kolejnych ramek obrazu i duża szybkość transferu obrazu.

Producent płytka Z-turn Board oferuje kilka modułów kamer. Wśród nich znajdują się dwa moduły, które zostały przetestowane: MY-CAM002U USB Digital Camera Module (Rys.3.7) oraz MY-CAM011B BUS Camera Module. (Rys. 3.4). W Tabeli 3.3 przedstawiono porównanie testowanych modułów kamer.

Tabela 3.3. Porównanie testowanych modułów kamer

	MY-CAM011B	MY-CAM002U
Maksymalna rozdzielcość	1600x1200 pikseli	1280x800 pikseli
Pobór mocy w stanie aktywnym	224 mW	110 mW
Format wyjścia	8/10-bit RAW RGB YUV422/YCbCr422 RGB565/555 GRB422	10-bit RAW RGB
Maksymalny transfer obrazu	15 fps (1600x1200) 30 fps (800x600) 30 fps (1280x720) 24 fps (1366x768)	30 fps (1280x800) 60 fps (640x480) 30 fps (1280x720)
Cena modułu	25\$ ⁴	19\$ ⁵



Rysunek 3.3. Płytkę rozszerzeniowa Z-turn IO Cape [15]

3.3.1. Moduł MY-CAM011B

Kamera MY-CAM011B zapewnia większą rozdzielcość maksymalną i posiada interfejs równoległy DVP. Skutkuje to mniejszymi opóźnieniami w wysyłaniu kolejnych ramek obrazu niż w przypadku kamery MY-CAM002U, podłączanej przez USB.

Pomimo znajdującego się na płytce IO-Cape portu interfejsu DVP, moduł kamery MY-CAM011B niestety okazał się niekompatybilny z płytą Z-turn Board. Na schemacie płytki rozszerzeniowej IO-Cape widać (Rys. 3.6), że sygnał zegara (CAM_XCLK) nie jest dołączony do żadnego portu płytki Z-Turn Board. Piny CAM_PWRDN i CAM_RST również nie są dołączone do złącza J2 od strony płytki Z-turn Board.

Dołączenie samego sygnału zegara do konektora FPC byłoby problematyczne, więc zdecydowano się na zastosowanie adaptera⁶ złącza FPC/FFC o rastrze 0,5 mm na otwory DIP raster 2,54 mm. Dzięki temu możliwe było podłączenie modułu kamery przy użyciu przewodów do złącza J3 płytki rozszerzeniowej IO-Cape [17].

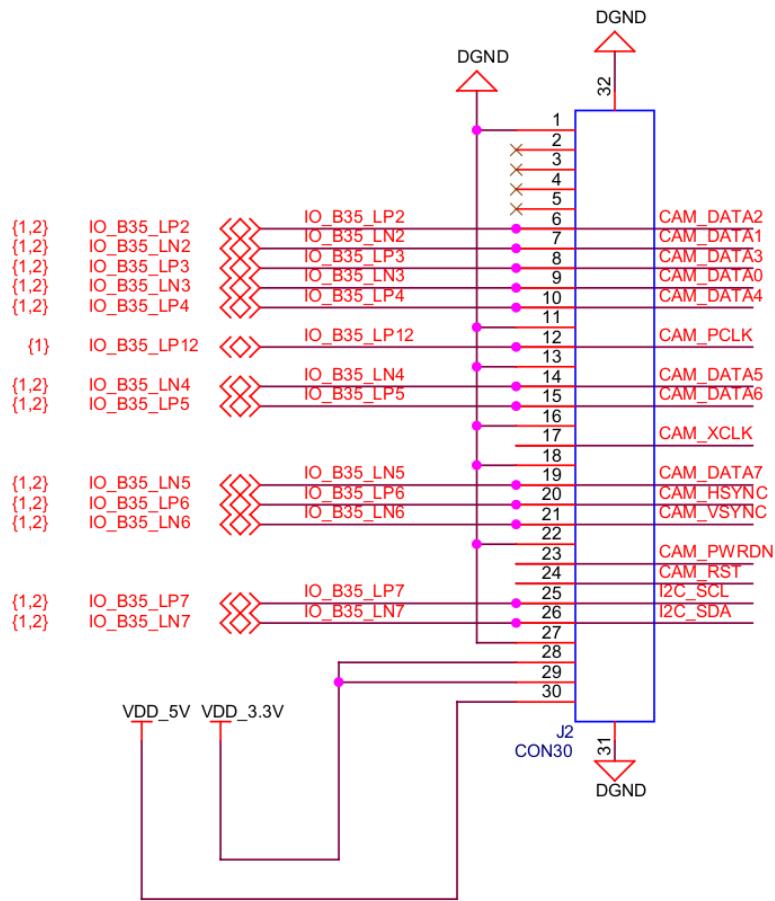
Po podłączeniu kamery, zainstalowaniu odpowiednich sterowników i *device-tree* oraz przeskanowaniu magistrali I^2C przy użyciu narzędzia *i2c-tools*, moduł kamery zwraca poprawnie swój adres. Jednak sterownik sensora kamery *ov2659.c* nie rozpoznaje podłączonego urządzenia.

⁴ <http://www.myirtech.com/list.asp?id=534>

⁵ <http://www.myirtech.com/list.asp?id=462>

⁶ <https://kamami.pl/zlaczka-fpc-zif/> 579387-adapter-zlaczka-fpcffc-05mm-30-pin-na-dip.html





Rysunek 3.5. Schemat portu DVP na płytce rozszerzeniowej IO-Cape [17]



Rysunek 3.6. Adapter złącza FPC/FFC o rastrze 0,5 mm na otwory DIP [18]



Rysunek 3.7. Moduł kamery MY-CAM001U USB Digital Camera Module [19]

4. Implementacja

W początkowej fazie projektu przetestowano wiele różnych modeli ANN w celu znalezienia odpowiedniej architektury sieci do zastosowania w przetwarzaniu obrazów. Początkowo zastosowano splotową sieć neuronową, która przy niewielkiej liczbie parametrów dawała bardzo dobre wyniki klasyfikacji odręcznie pisanych cyfr z bazy MNIST. Jednak w celu zaimplementowania algorytmu w układzie FPGA ograniczono model do postaci architektury MLP. Początkowym założeniem była możliwość zmiany parametrów sieci z poziomu aplikacji.

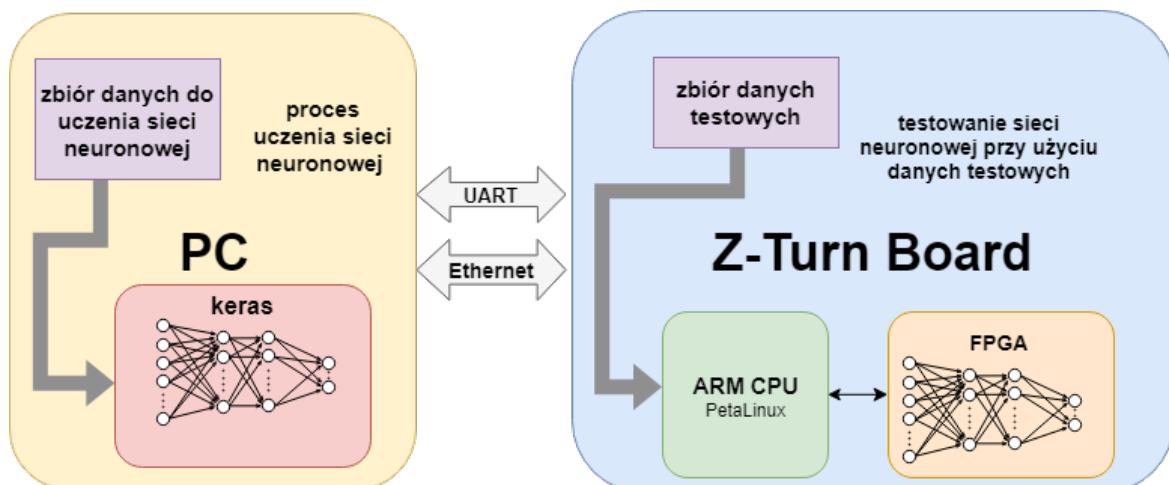
4.1. Budowa systemu

Na początku pracy przyjęto założenie wykonywania uczenia sieci na komputerze PC z wykorzystaniem pakietu *keras*, a następnie eksport modelu w postaci plików tekstowych zawierających parametry sieci. Zbiór obrazów przedstawiających odręcznie pisane cyfry, został podzielono na testowy (10000 obrazów) oraz uczący (60000 obrazów). Ponadto system powinien umożliwiać rozpoznawanie i klasyfikację cyfr na obrazie z kamery w czasie rzeczywistym.

4.1.1. Schemat blokowy systemu

System można podzielić na dwie główne części (Rys. 4.1):

- aplikacja wykorzystująca pakiet keras, uruchamiana na komputerze PC
- część uruchamiana na płytce Z-Turn Board



Rysunek 4.1. Schemat blokowy systemu

4.2. Wybór narzędzi

Użycie w pracy płytki z układem Zynq determinuje użycie narzędzi wspieranych przez firmę Xilinx. Zdecydowano się na użycie najnowszej (w momencie rozpoczęcia projektu)

4. Implementacja

wersji oprogramowania 2019.2. Producent zaleca [20] instalację programu Vivado na jednym ze wspieranych systemów operacyjnych:

- Microsoft Windows 7 SP1 Professional (64-bit), English/Japanese
- Microsoft Windows 10.0 1809 Update; 10.0 1903 Update (64-bit), English/Japanese
- Red Hat Enterprise Workstation/Server 7.4, 7.5, and 7.6 (64-bit)
- SUSE Linux Enterprise 12.4 (64-bit)
- CentOS 7.4, 7.5, and 7.6 (64-bit)
- Ubuntu Linux 16.04.5 LTS; 16.04.6 LTS; 18.04.1 LTS; 18.04.02 LTS (64-bit)
- Amazon Linux 2 LTS (64-bit).

W projekcie wykorzystywane było również narzędzie Petalinux, które wymaga zainstalowania na maszynie z systemem operacyjnym Linux. Zgodnie z dokumentacją [21] jest to jedna z trzech dystrybucji:

- Red Hat Enterprise Workstation/Server 7.4, 7.5, 7.6 (64-bit)
- CentOS Workstation/Server 7.4, 7.5, 7.6 (64-bit)
- Ubuntu Linux Workstation/Server 16.04.5, 16.04.6, 18.04.1, 18.04.02 (64-bit)

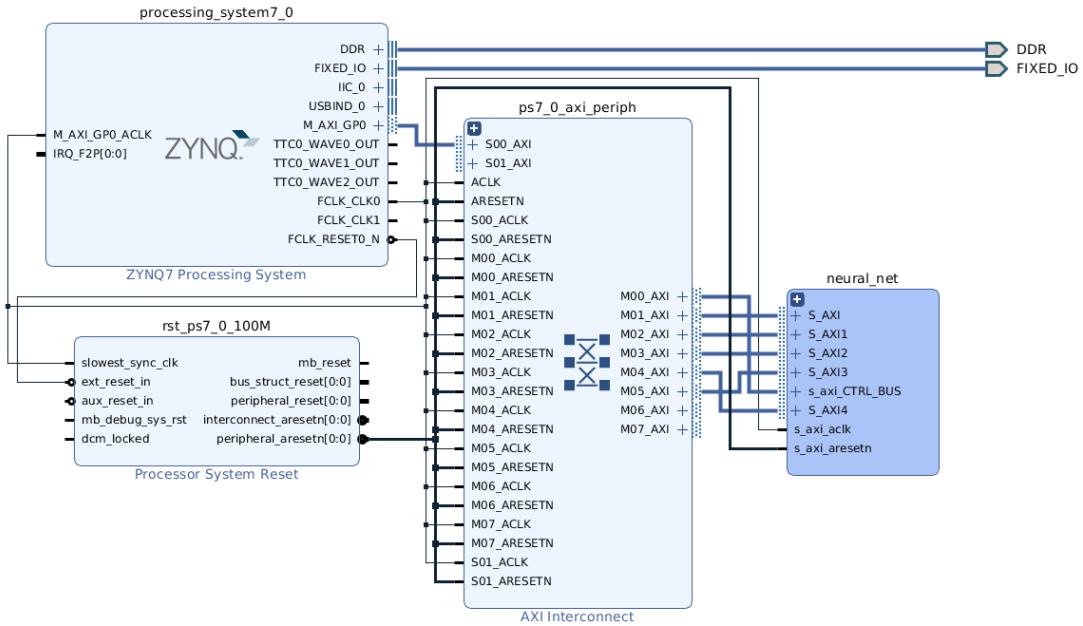
Aby zapewnić poprawne działanie narzędzi oraz z racji na sporą popularność i duże wsparcie społeczności wybrano dystrybucję Ubuntu 18.04.02 LTS. Przy instalacji Petalinuxa warto również zwrócić uwagę, że zalecane jest aż 100 GB wolnego miejsca na dysku twardym.

4.3. Projekt systemu w środowisku Vivado

Zdecydowano się na zastosowanie bloków pamięci BRAM do komunikacji pomiędzy blokiem IP implementowanym z użyciem HLS a procesorem. Zrzut ekranu przedstawiający schemat systemu w środowisku Vivado umieszczono na Rys. 4.2. Znajduje się na nim m.in. blok IP *ZYNQ7 Processing System*, przedstawiający procesor, oraz blok *AXI Interconnect*, umożliwiający podłączenie peryferiów do procesora. Po podwójnym kliknięciu na blok procesora pojawia się okno, w którym można ustawić odpowiednią konfigurację układu.

4.3.1. Pamięć Block RAM

Układy FPGA zawierają bloki dwuportowej pamięci BRAM, które można wykorzystać do komunikacji PS – PL pomiędzy logiką programowalną (ang. PL – *Programmable Logic*) a procesorem (ang. PL – *Processing System*). Zaletą pamięci BRAM jest możliwość równoległego dostępu do danych z dwóch niezależnych portów, jednak ilość pamięci, jaką można wykorzystać, jest mocno ograniczona. W przypadku układu Zynq-7020 znajdującego się na płytce Z-turn jest to 140 bloków po 36Kb (łącznie 4.9Mb). Pamięć BRAM umożliwia dostęp z poziomu sterownika w systemie PetaLinux do danych przetworzonych przez



Rysunek 4.2. Zrzut ekranu przedstawiający schemat systemu w środowisku Vivado

blok HLS. Zdecydowano się na zastosowanie bloków pamięci BRAM do przechowywania parametrów modelu sieci oraz wejść i wyników obliczeń.

W programie Vivado pamięć alokowana jest przy użyciu bloku *Block Memory Generator* i podłączana do procesora dzięki blokom *AXI BRAM Controller* (Rys. 4.3) oraz *AXI Interconnect* (Rys. 4.2). Po odpowiednim podłączeniu bloków pamięci, w zakładce *Address Editor* można ustawić rozmiar każdego z dodanych bloków BRAM.

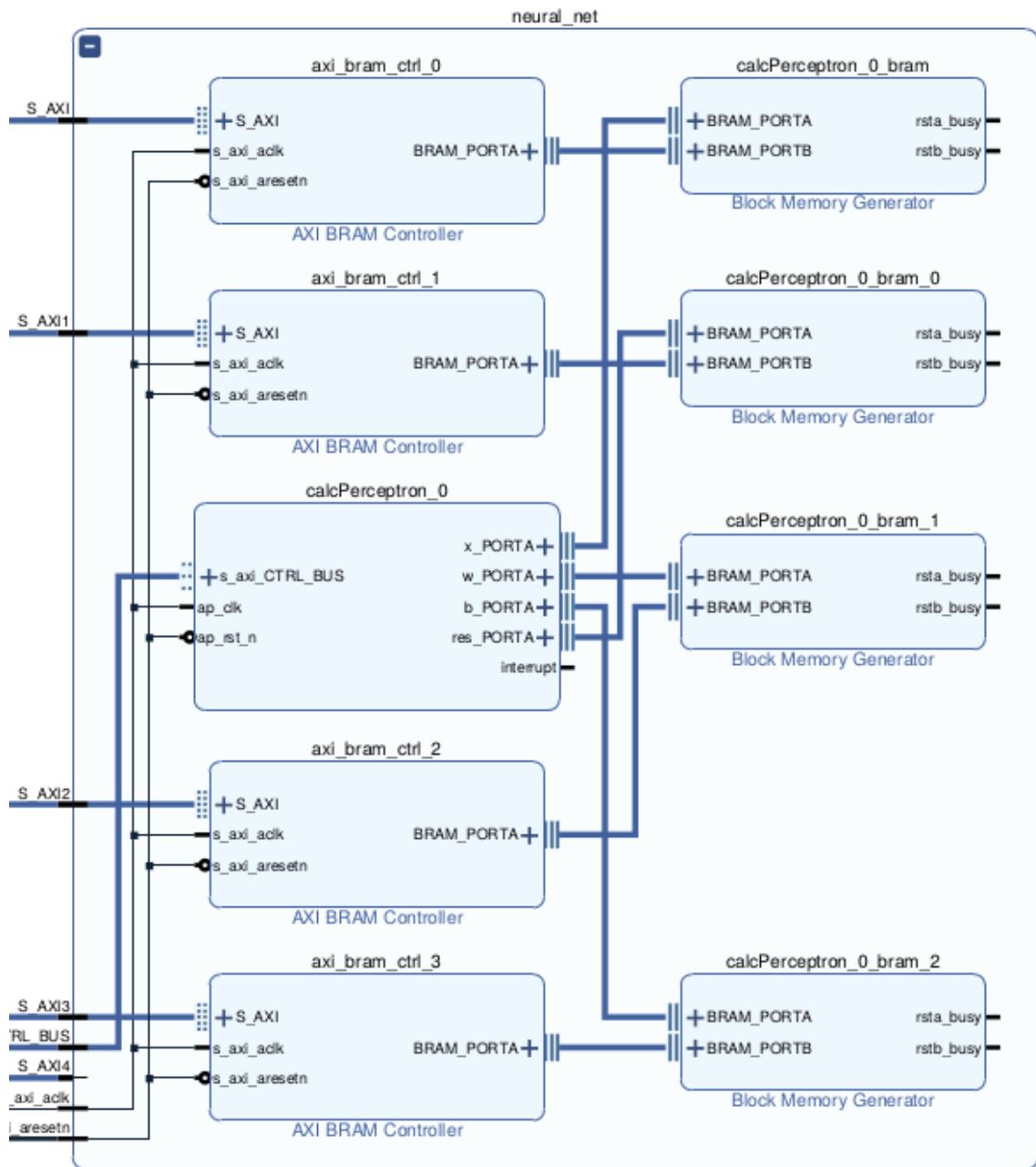
4.4. Wykorzystanie metody HLS

Przy użyciu metody HLS możliwe jest stworzenie własnego bloku IP (ang. Intellectual Property), który następnie jest umieszczany w katalogu IP i można go wielokrotnie wykorzystać w projekcie RTL (ang. Register Transfer Level). Do projektu z użyciem HLS (Rys. 4.4) potrzebny jest plik z algorytmem w języku C/C++ lub System C, plik testowy napisany w języku C (ang. *test bench*) oraz plik z opisem ograniczeń sprzętowych (ang. *constraints*). Kolejne etapy projektu z wykorzystaniem metody HLS [22]:

1. Kompilacja, wykonanie (symulacja) i debugowanie algorytmu napisanego w języku C
2. Synteza algorytmu w języku C w implementację RTL
3. Wygenerowanie raportu i analiza projektu (optymalizacja)
4. Zweryfikowanie implementacji RTL
5. Spakowanie implementacji RTL w blok IP

Zastosowanie syntezy wysokiego poziomu umożliwia przeniesienie algorytmu napisanego w języku C/C++ lub System C na implementację w układzie FPGA. Dodatkową

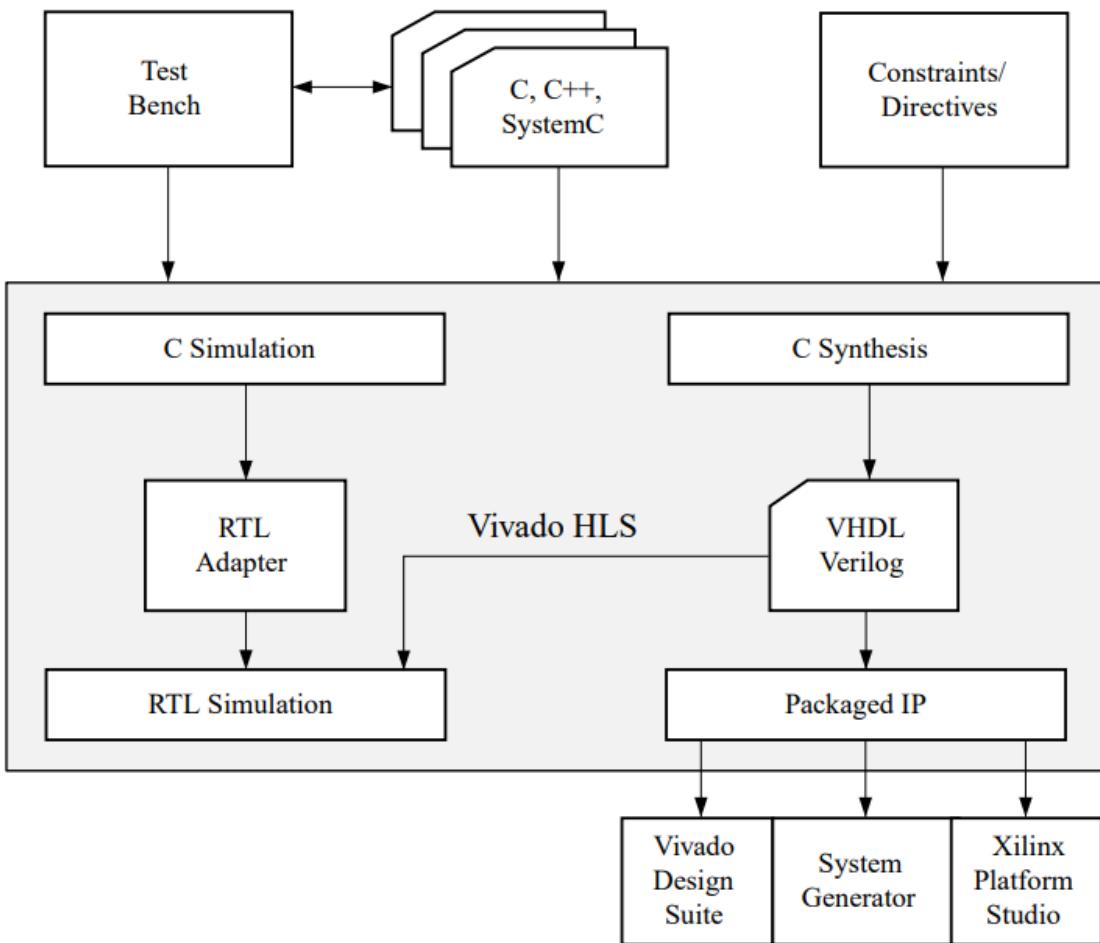
4. Implementacja



Rysunek 4.3. Zrzut ekranu przedstawiający szczegółowy schemat części systemu *neural_net* w środowisku Vivado

zaletą metody HLS jest dostępność bibliotek do przetwarzania obrazów oraz ułatwiających implementację operacji matematycznych. Podczas pracy nad projektem wykorzystano bibliotekę *<hls_math.h>* zawierającą operacje matematyczne (np. funkcję *sigmoid*), podobnie jak biblioteka *<cmath.h>*. Obie biblioteki można wykorzystać w implementacji akceleratora HLS, jednak użycie biblioteki *<cmath.h>* może powodować powstanie różbieżnych wyników obliczeń podczas symulacji i syntezy.

Przy tworzeniu nowego projektu w Vivado HLS trzeba wybrać urządzenie, na którym



Rysunek 4.4. Proces projektowania przy użyciu metody HLS [22]

uruchamiany będzie blok IP. Jeśli płytnka nie jest widoczna na liście, należy dodać pliki opisujące ją w odpowiednim katalogu, gdzie zostało zainstalowane narzędzie Vivado HLS.

4.5. Optymalizacja w Vivado HLS

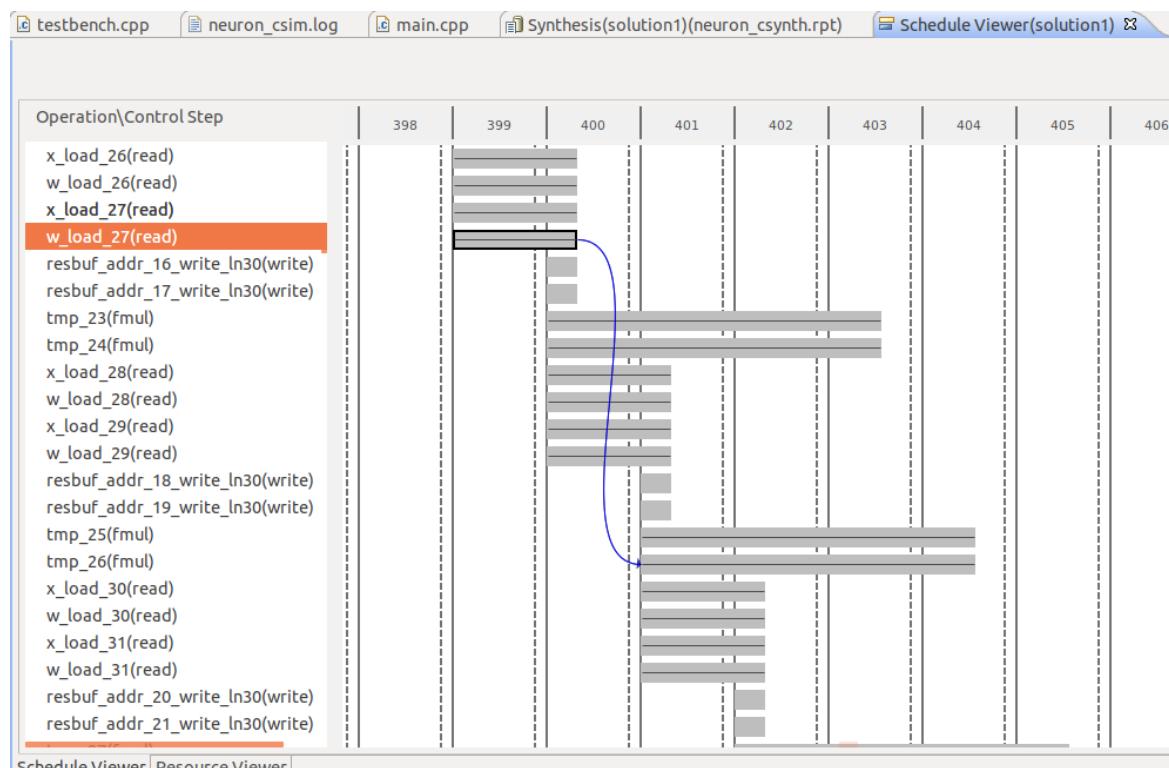
Metoda HLS umożliwia optymalizację implementowanego algorytmu pod kątem zużycia zasobów układu FPGA oraz wprowadzanych opóźnień (ang. *latency*). Zaimplementowany algorytm w języku C++ można optymalizować, stosując m.in. następujące dyrektywy:

- rozwijanie pętli – *loop unroll*
- przetwarzanie potokowe – *pipeline*
- partycjonowanie tablic – *array partition*
- definiowanie zależności – *dependence*

Wykorzystanie tych metod umożliwia poprawę wydajności implementacji, jednak zazwyczaj wiąże się ze zużyciem większej ilości zasobów logiki programowalnej. W narzędziu Vivado HLS dyrektywy są implementowane przy użyciu *pragm*, umieszczanych w odp-

4. Implementacja

wiednie miejsce w kodzie. Po wprowadzeniu zmian w kodzie i przeprowadzeniu syntezy narzędzie Vivado HLS generuje raport, który umożliwia wstępne sprawdzenie efektów optymalizacji. Na Rys. 4.6 przedstawiono raport po przeprowadzeniu syntezy implementacji, w której wszystkie pętle są zależne od parametrów modelu ustawionych z poziomu aplikacji. Stąd, widoczne w raporcie znaki zapytania w tabeli *Latency*, gdyż opóźnienia zależą od parametrów funkcji akceleratora. Poza opóźnieniami warto przeanalizować ilość zużytych zasobów, a w szczególności pole *Utilization*, informujące o procentowym zużyciu zasobów. Dodatkowo przydatnym narzędziem jest *Schedule Viewer* dostępny w zakładce *Analysis* (Rys. 4.5), umożliwiający analizę czasu wykonania instrukcji w kolejnych cyklach zegara.



Rysunek 4.5. Widok narzędzia Schedule Viewer w Vivado HLS

W prezentowanym przypadku analiza na poziomie implementacji HLS nie była możliwa ze względu na zbyt dużą ilość zmiennych parametrów, ustalanych na etapie uruchomienia aplikacji. Przeprowadzono testy z poziomu aplikacji w systemie Petalinux dla różnych modeli sieci. Na tym etapie projektu w celu pełnego skorzystania z metod optymalizacji w Vivado HLS, podjęto decyzję o wybraniu modelu sieci, który zostanie na stałe zaimplementowany, bez możliwości zmiany liczby neuronów i warstw sieci.

4.5.1. Rozwijanie pętli

Każda pętla w danej implementacji wprowadza pewne opóźnienie (ang. *latency*) w działaniu kodu. Przykładowo, gdy pętla składa się z 4 iteracji, nie ma możliwości, żeby

Synthesis(solution1)(calcPerceptron_csynth.rpt) ☒

Performance Estimates

- Timing
 - Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.685 ns	1.25 ns
- Latency
 - Summary

Latency (cycles)	Latency (absolute)	Interval (cycles)				
min	max	min	max	min	max	Type
?	?	?	?	?	?	?none
- Detail
 - Instance
 - Loop

Loop Name	Latency (cycles)		Iteration Latency	Initiation Interval				
	min	max		achieved	target	Trip Count	Pipelined	
- calcPerceptron_label2	?	?		?	-	-	?	no
+ calcPerceptron_label3	?	?		?	-	-	?	no
++ calcPerceptron_label0	?	?	12	-	-	-	?	no
+ calcPerceptron_label1	?	?	3	-	-	-	?	no
- Loop 2	?	?	20	-	-	-	?	no

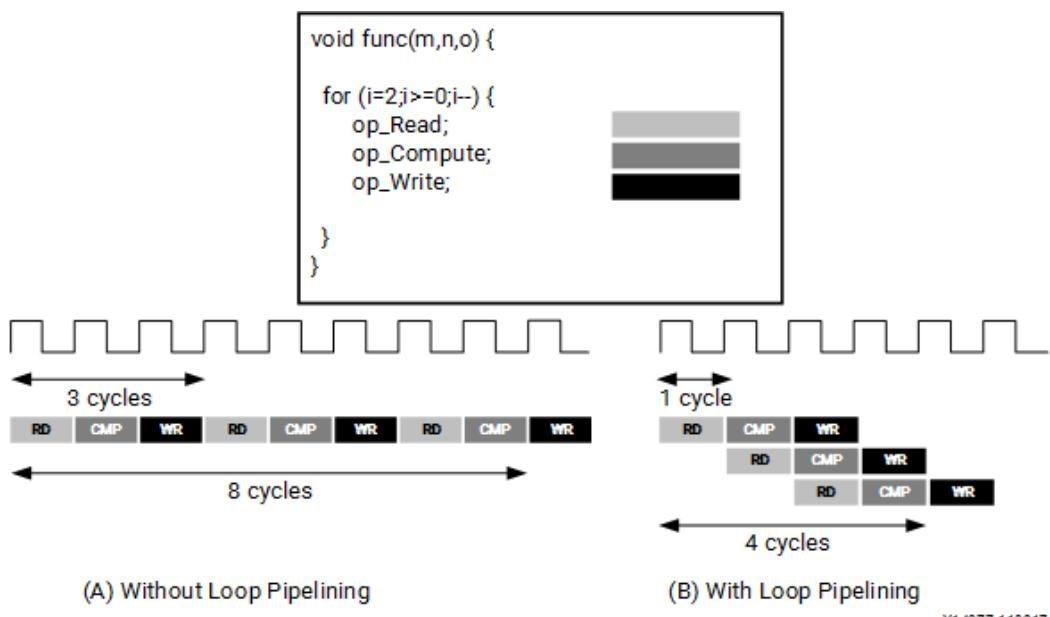
Utilization Estimates

Rysunek 4.6. Raport po przeprowadzeniu syntezy w Vivado HLS

kod wykonał się szybciej niż w 4 cyklach zegara. Aby przyspieszyć wykonanie kodu, można zastosować rozwijanie pętli (ang. *loop unrolling*). W przypadku rozwijania całej pętli wszystkie iteracje danej pętli wykonują się równolegle. W przypadku pętli o wielu iteracjach i ograniczonych zasobach można zastosować rozwijanie częściowe. Użytkownik podaje wartość współczynnika, która informuje o tym, ile iteracji pętli rozpoczęcie wykonanie jednocześnie.

4.5.2. Pipeline

Zastosowanie dyrektywy *pipeline* umożliwia wykonanie jednej iteracji funkcji lub pętli przed zakończeniem wykonywania poprzedniej. Argumentem dyrektywy PIPELINE jest *II* (ang. *Initiation Interval*), który informuje o tym, ile taktów zegara będzie pomiędzy rozpoczęciem wykonywania kolejnych instrukcji. W zależności od poziomu skomplikowania implementacji i zależności zmiennych w danej pętli, osiągnięcie $II=1$ może wymagać wprowadzenia zmian w kodzie lub zastosowania dodatkowych dyrektyw. Wpływ zastosowania dyrektywy *pipeline* na ilość cykli zegara potrzebnych do wykonania pętli zaprezentowano na Rys. 4.7.



Rysunek 4.7. Wynik zastosowania dyrektywy *pipeline* w celu optymalizacji pętli, (A) – bez dyrektywy *pipeline*, (B) – z zastosowaniem dyrektywy *pipeline* [23]

4.5.3. Dyrektywa *Dependence*

Kompilator Vivado HLS automatycznie wykrywa zależności pomiędzy zmiennymi wewnętrz pętli i pomiędzy różnymi iteracjami pętli (ang. *loop-carry dependence*) [24]. W niektórych złożonych implementacjach może dojść do nieprawidłowego zinterpretowania zależności. W tym wypadku można zastosować dyrektywę DEPENDENCE na zmiennej znajdującej się wewnątrz danej pętli lub funkcji, podając jako parametr wartość *false*.

4.5.4. Partycjonowanie pamięci

W przypadku posiadania w kodzie zmiennych tablicowych, zawierających dużo elementów, można zastosować dyrektywę ARRAY PARTITION. Powoduje to rozdzielenie pamięci do większej liczby bloków, których liczba zależy od wartości parametru *factor* podanego w dyrektywie. Partycjonowanie pamięci umożliwia zwielokrotnienie dostępu (zapis i odczyt) do pamięci. W przypadku zastosowania dyrektywy:

```
#pragma HLS ARRAY_PARTITION variable=w block factor=4 dim=1
```

pamięć przypisana do zmiennej *w* zostanie podzielona na 4 bloki zawierające $\frac{N}{4}$ elementów, gdzie N jest początkowym rozmiarem tablicy.

4.5.5. Arytmetyka zmiennoprzecinkowa

Kolejną metodą optymalizacji w metodzie HLS jest zamiana arytmetyki zmiennoprzecinkowej (ang. *floating point*) na zapis stałoprzecinkowy (ang. *fixed point*). Narzędzie Vivado HLS zawiera bibliotekę *ap_fixed.h*, która ułatwia definiowanie zmiennych o wybranym typie stałoprzecinkowym oraz implementowanie operacji matematycznych z użyciem różnych typów stałoprzecinkowych. Programista może wybrać liczbę bitów przeznaczonych na część całkowitą i ułamek danej zmiennej, co umożliwia odpowiednie dopasowanie typu zmiennych do wykonywanych obliczeń. Dzięki temu możliwe jest zmniejszenie zużycia zasobów oraz akceleracja obliczeń przy zachowaniu założonej precyzji. Przy dobieraniu liczby bitów w typie stałoprzecinkowym należy wziąć pod uwagę zakres wartości, jakie będzie w stanie przechowywać, co bywa skomplikowane w przypadku złożonych algorytmów z dużą ilością operacji matematycznych.

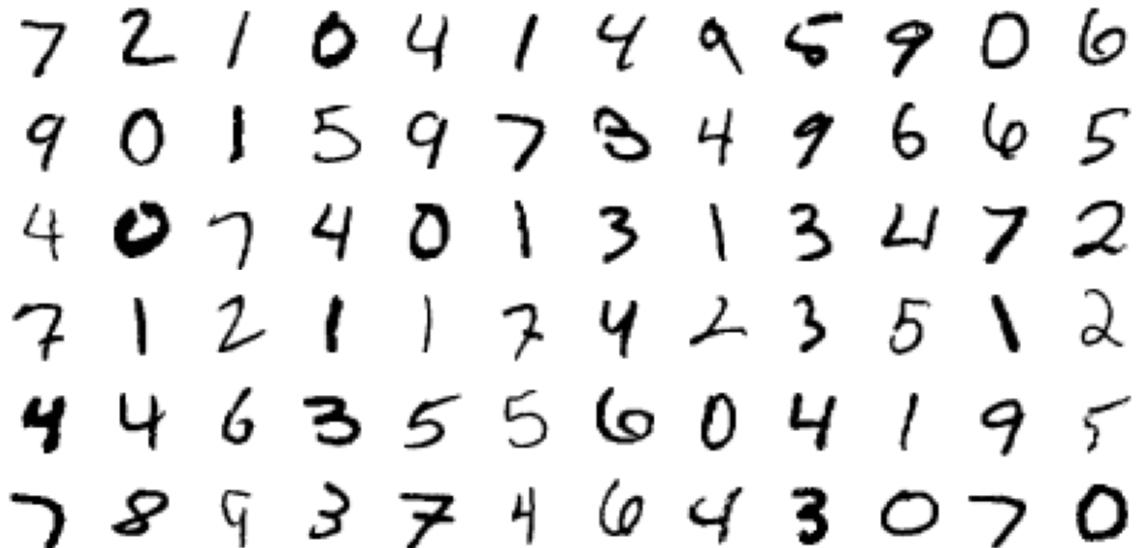
4.6. Zbiór danych wejściowych

W procesie uczenia oraz testowania poprawności działania modelu sztucznej sieci neuronowej wykorzystano zbiór odręcznie pisanych cyfr MNIST (ang. THE MNIST DATABASE of handwritten digits)[25]. Jest to baza 60000 obrazów do przeznaczonych do uczenia sieci oraz 10000 do walidacji. Każdy obraz przedstawia jedną cyfrę w formacie 28×28 pikseli. Fragment zbioru MNIST przedstawiono na Rys. 4.8.

4.7. Opracowanie modelu ANN

Przy projektowaniu algorytmu ANN bardzo ważnym aspektem jest odpowiednie dopasowanie modelu do danych wejściowych. Najczęściej odbywa się to poprzez wielokrotne testowanie systemu dla różnych parametrów sieci. W tej pracy początkowym wyborem była architektura MLP. Dane wejściowe w postaci obrazów 28×28 pikseli są konwertowane na tablicę 784 wartości z przedziału od 0 do 1. Fragment kodu przedstawia Listing 1. Model zawiera następujące warstwy:

- warstwę wejściową (784 wejścia)



Rysunek 4.8. Fragment zbioru odręcznie pisanych cyfr MNIST [25]

- warstwę ukrytą (16 neuronów)
- warstwę wyjściową (10 neuronów).

Listing 1. Implementacja modelu ANN MLP z jedną warstwą ukrytą

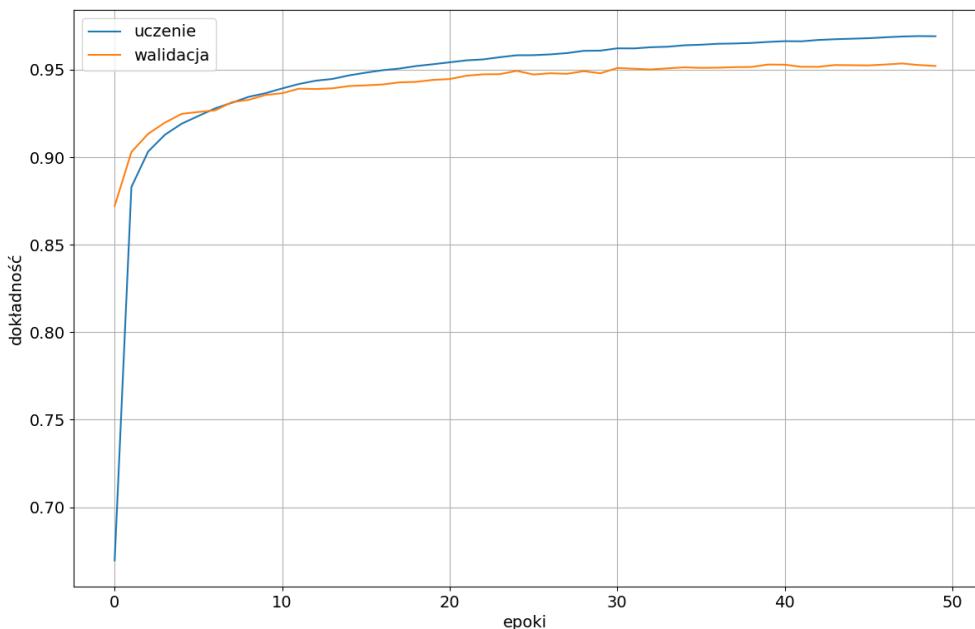
```
55 model = Sequential()  
56 model.add(Flatten())  
57 model.add(Dense(16, use_bias=True, activation='sigmoid'))  
58 model.add(Dense(num_classes, use_bias=True, activation='sigmoid'))
```

4.7.1. Uczenie Sztucznej Sieci Neuronowej

Podczas uczenia i testowania modelu użyto zbioru MNIST, który zainportowano, wykorzystując funkcję z pakietu keras. Dokonano uczenia sieci neuronowej przy użyciu zbioru 60000 obrazów i testowania modelu, podając na wejście sieci 10000 obrazów. Osiągnięto dokładność na poziomie 94,97%. Na Rys. 4.9 przedstawiono, jak zmieniała się dokładność (ang.*accuracy*) w kolejnych epokach dla zbioru uczącego i walidacyjnego.

4.7.2. Implementacja modelu przy użyciu narzędzia Vivado HLS

Korzystając z narzędzia Vivado HLS, napisano program w języku C++ implementujący zaprojektowany wcześniej model sieci. W procesie uczenia ustalono wartości wag i biasów. Dwa główne pliki projektu w narzędziu Vivado HLS to core.cpp, zawierający implementację algorytmu ANN oraz test_core.cpp, który służy do przetestowania algorytmu po przeprowadzeniu syntezy (ang. *test bench*). Pliki zawierające wartości wejściowe, a także wagi i biasy są importowane do programu test_core.cpp.



Rysunek 4.9. Wykres zmian dokładności w kolejnych epokach – ANN z jedną warstwą ukrytą

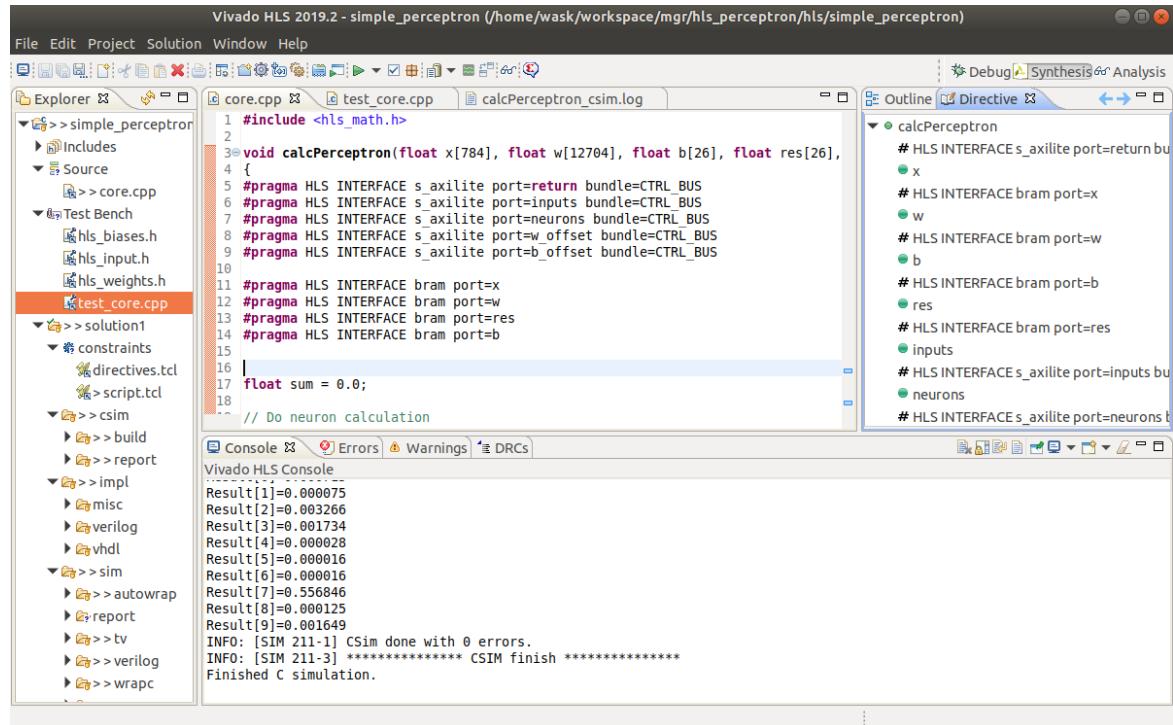
Pierwszym etapem jest Symulacja C, która jest wstępnią weryfikacją poprawności algorytmu. Do przeprowadzenia symulacji wykorzystano dane wyeksportowane przy użyciu skryptu *keras2fpga.py* i zapisane w plikach *hls_biases.h*, *hls_weights.h* oraz *hls_input.h*. Wynikiem symulacji jest plik *.log*, w którym można znaleźć informacje o tym, jak przebiegało wywołanie testowanych funkcji. 4.10 Symulację w języku C wykonuje się dużo szybciej niż późniejszą symulację RTL, więc stosuje się ją jako pierwszy etap przed podjęciem dalszych kroków, które zajmują więcej czasu.

Następnie wykonywana jest synteza oraz kosymulacja, umożliwiająca weryfikację poprawności syntezy. Ponadto narzędzie generuje raport, który przedstawia informacje na temat zużycia zasobów i opóźnień czasowych. Po prawidłowym przeprowadzeniu kosymulacji należy użyć opcji *Export RTL*, co umożliwia dodanie nowego bloku IP do projektu w narzędziu Vivado.

4.8. Synteza projektu w narzędziu Vivado

Aby przetestować działanie nowego bloku IP w narzędziu Vivado HLS, należy dodać i odpowiednio podłączyć blok do schematu blokowego (ang. *Block Design*) w narzędziu Vivado. Następnie należy sprawdzić, czy urządzenia zostały właściwie zaadresowane w zakładce *Adress Editor* i wprowadzić ewentualne zmiany. Gotowy projekt można poddać automatycznej weryfikacji za pomocą funkcji *Validate Design* i uruchomić syntezę i implementację. Po poprawie przeprowadzonej implementacji narzędzie umożliwia otarcie

4. Implementacja



Rysunek 4.10. Wynik poprawnie przeprowadzonej symulacji w Vivado HLS

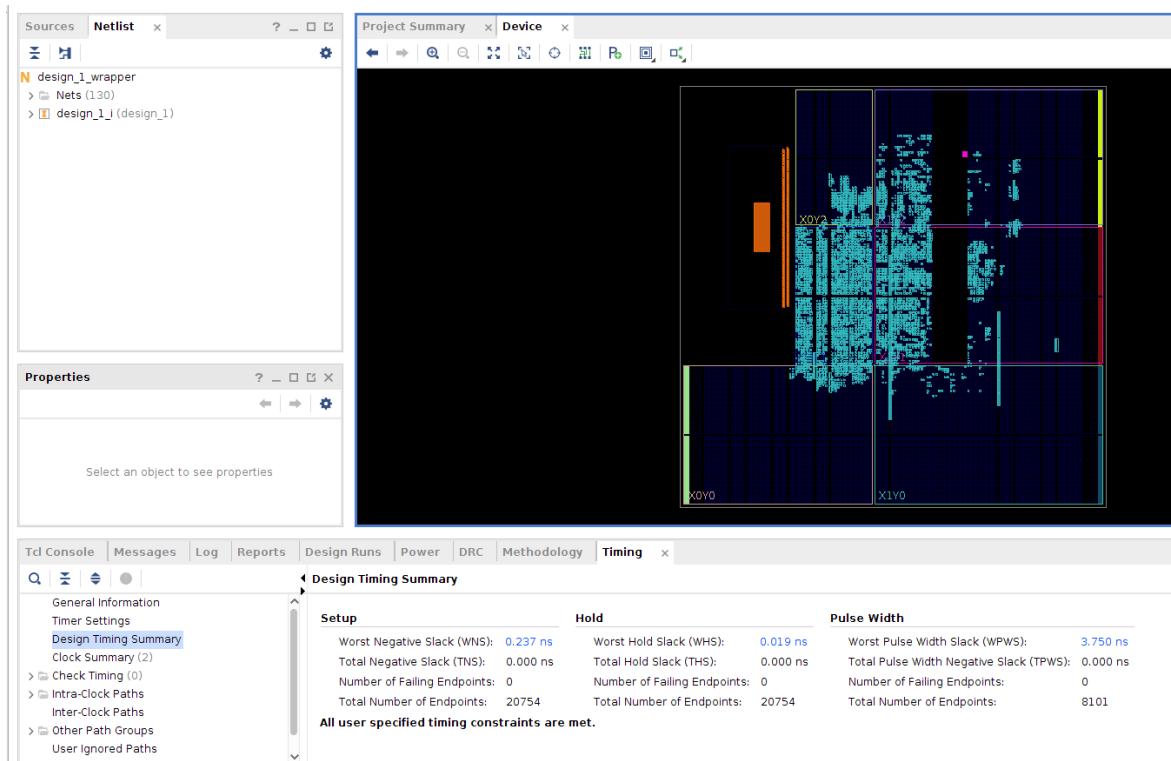
realizacji sprzętowej projektu (opcja *Open Implemented Design*). Zakładka *Timing* (Rys. 4.11) umożliwia sprawdzenie, czy w zaimplementowanym projekcie zostały spełnione wymagania czasowe, a w oknie *Device* można zweryfikować jakie zasoby zostały użyte w implementacji.

Następnym krokiem jest wygenerowanie *Bitstreamu* i eksport sprzętu w postaci pliku z rozszerzeniem *.xsa*. Dzięki temu projekt sprzętu stworzony w programie Vivado można użyć w narzędziu *Vitis* lub *Petalinux*. Aplikacja uruchamiana w trybie *standalone* bez systemu operacyjnego w programie Vitis umożliwia szybką weryfikację poprawności działania zaprojektowanego systemu. Dlatego przed przejściem do implementacji w systemie Petalinux przystąpiono do stworzenia projektu oprogramowania w programie Vitis.

4.9. Implementacja przy użyciu narzędzia Vitis

Po wyeksportowaniu projektu sprzętu w narzędziu Vivado można uruchomić program Vitis. Narzędzie umożliwia stworzenie nowego projektu platformy, dzięki opcji *New Platform Project* (Rys. 4.12). Przy tworzeniu projektu należy wybrać odpowiedni plik *.xsa* (Rys. 4.13), wyeksportowany wcześniej z narzędzia Vivado.

Następnie należy stworzyć projekt aplikacji, za pomocą opcji *New Application Project* i wybranemu odpowiedniego, stworzonego wcześniej projektu platformy. Narzędzie Vitis zawiera szereg aplikacji przykładowych, z których warto skorzystać przy uruchamianiu systemu po raz pierwszy. Przed zbudowaniem projektu, aby zapewnić komunikację płytki Z-turn Board z komputerem za pomocą konwertera USB-UART znajdującego się na płytce,



Rysunek 4.11. Wynik poprawnie przeprowadzonej implementacji w narzędziu Vivado

należy zmienić następujące ustawienia BSP (ang. *Board Support Package*) dla domeny *standalone*:

- `stdin` z `ps_7uart_0` na `ps_7uart_1`
- `stdout` z `ps_7uart_0` na `ps_7uart_1`

Po zbudowaniu projektu platformy można przejść do pisania kodu aplikacji. Warto zaznaczyć, że do uruchomienia i debugowania aplikacji na płytce Z-turn Board, potrzebny jest programator JTAG *Xilinx Platform Cable*. Aby obejrzeć wyjście standardowe programu, należy otworzyć okno *Vitis Serial Terminal* i wybrać odpowiedni port USB. Trzeba również pamiętać o wybraniu odpowiedniej opcji uruchamiania płytka, poprzez ustawienie zworki na JP1 zwarte i JP2 rozwarste.

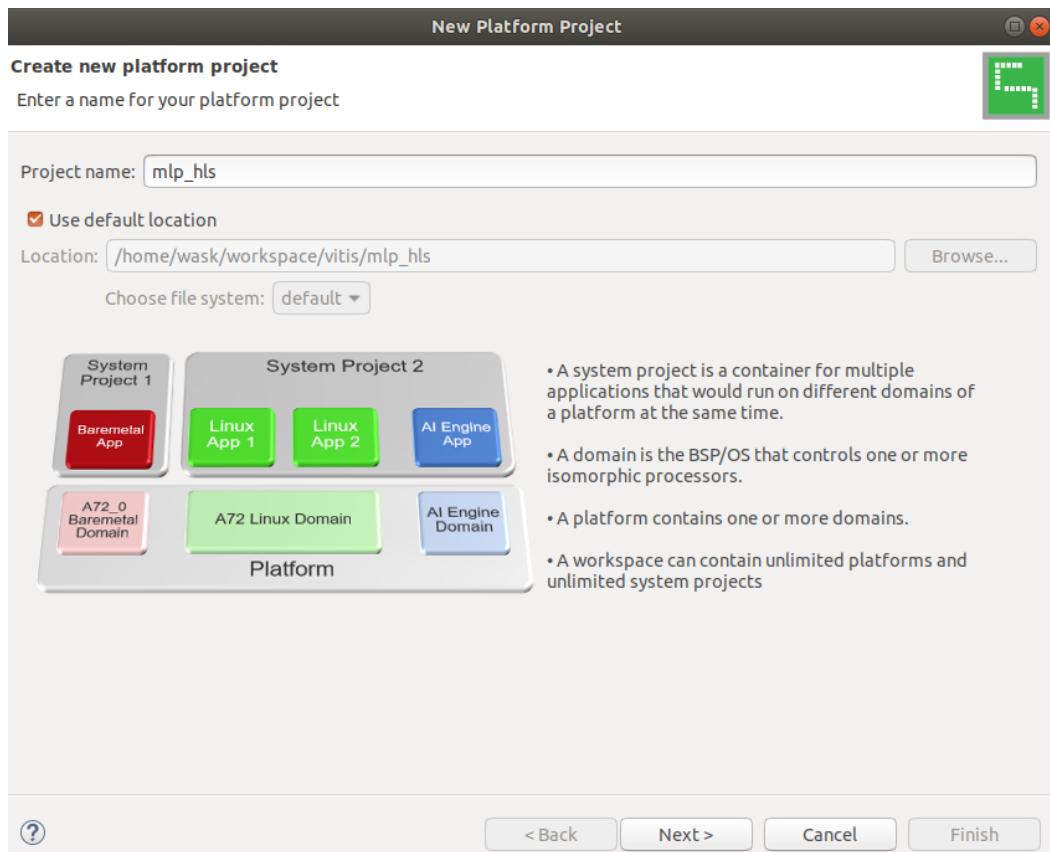
Jak widać na Rys. 4.14, algorytm działa poprawnie. Po uruchomieniu i sprawdzeniu poprawności obliczeń w programie Vitis można przystąpić do konfigurowania systemu operacyjnego przy użyciu narzędzia Petalinux.

4.10. Test z wykorzystaniem systemu operacyjnego Petalinux

PetaLinux Software Development Kit (SDK) jest narzędziem zawierającym wszystkie niezbędne elementy do budowania, rozwijania, testowania i wdrażania systemów wbudowanych opartych na systemie Linux. PetaLinux jest przeznaczony głównie do systemów, opartych o układy FPGA i składa się z trzech najważniejszych elementów:

- prekonfigurowany obraz binarny systemu

4. Implementacja



Rysunek 4.12. Tworzenie nowego projektu w programie Vitis

- system Linux konfigurowalny do zastosowania na sprzęcie firmy Xilinx
- PetaLinux SDK zawierający narzędzia służące do projektowania i wdrażania systemu

4.10.1. Wstępna konfiguracja systemu Petalinux

Zgodnie z zaleceniami producenta, projekcie wykorzystano wersję narzędzia Petalinux kompatybilną z wersją Vivado – 2019.2. Każdorazowo, przed użyciem Petalinuxa należy pamiętać o załadowaniu ustawień przy pomocy komendy:

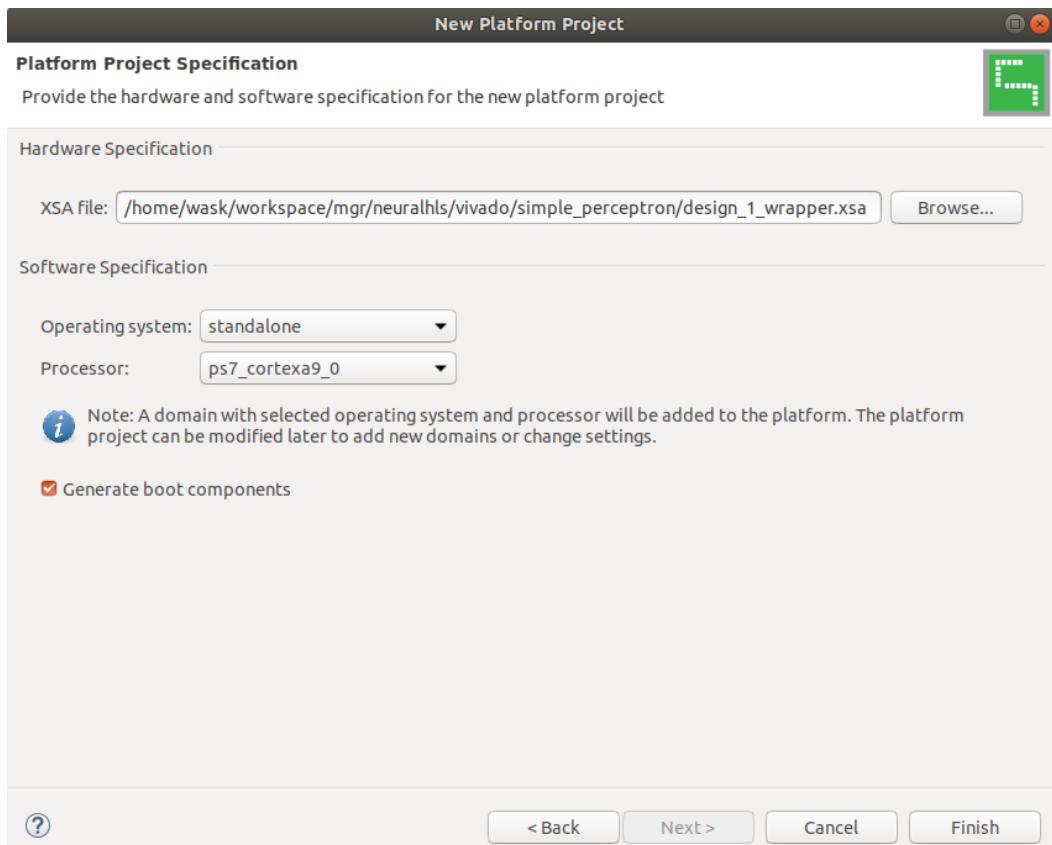
```
source <katalog-instalacyjny>/petalinux/2019.2/settings.sh
```

Narzędzie zużywa dużo przestrzeni dyskowej, jeśli wolnego miejsca będzie zbyt mało, zostanie wyświetlony odpowiedni komunikat. Następnie można przejść do tworzenia nowego projektu z ustawieniami domyślnymi dla płyt z układami *Zynq*:

```
petalinux-create -t project -n <nazwa_projektu> --template_zynq
```

Projekt sprzętu w programie Vivado został wyeksportowany do pliku z rozszerzeniem *.xsa*. Aby zimportować plik opisujący konfigurację sprzętu, utworzonego w programie Vivado stosujemy następującą komendę:

```
petalinux-config --get-hw-description <katalog_projektu_vivado>
```



Rysunek 4.13. Wybór pliku .xsa z opisem konfiguracji sprzętu w programie Vitis

Po wywołaniu komendy *petalinux-config* pojawia się interfejs graficzny narzędzia Petalinux, który umożliwia zmianę ogólnych ustawień projektu (Rys. 4.15). Przechodząc do zakładki *Image Packaging Configuration* (Rys. 4.15), można ustawić lokalizację zewnętrznego systemu plików oraz wybrać partycję karty sd, na której będzie się znajdował. Stosując opcję -c komendy *petalinux-config* można dostosować ustawienia jądra Linuxa (argument *kernel*) oraz systemu plików (argument *rootfs*).

4.10.2. Konfiguracja jądra systemu Petalinux

Po wywołaniu komendy *petalinux-config -c kernel* w terminalu pojawia się menu konfiguracyjne jądra systemu Petalinux. W oknie jest wiele opcji, jednak najczęściej zmian wprowadzono w zakładce *Device Drivers*.

W projekcie podjęto decyzję o zastosowaniu kamery podłączonej przy użyciu portu USB. Aby umożliwić aplikacji korzystającej z biblioteki OpenCV dostęp do urządzenia, potrzebne były odpowiednie sterowniki [26] dostępne w zakładce *Multimedia Support*. Zastosowany w projekcie moduł kamery MY-CAM002U jest kompatybilny ze sterownikiem UVC (ang. *USB Video Class*). W konfiguracji jądra dodano również interfejs V4L2, który daje dostęp do modułu kamery z poziomu aplikacji z przestrzeni użytkownika.

Aby uzyskać dostęp z poziomu systemu Petalinux do bloku IP zaimplementowanego w narzędziu Vivado HLS, należy zainstalować odpowiednie sterowniki urządzeń lub utwo-

4. Implementacja

The screenshot shows the Vitis IDE interface. On the left, the code editor displays `main.cpp` with several lines of C++ code related to a neural network. On the right, the `Vitis Serial Terminal` window shows the execution results of the application. The terminal output includes the neural network's test message, initial data loading, and the resulting 10 digits for each sample.

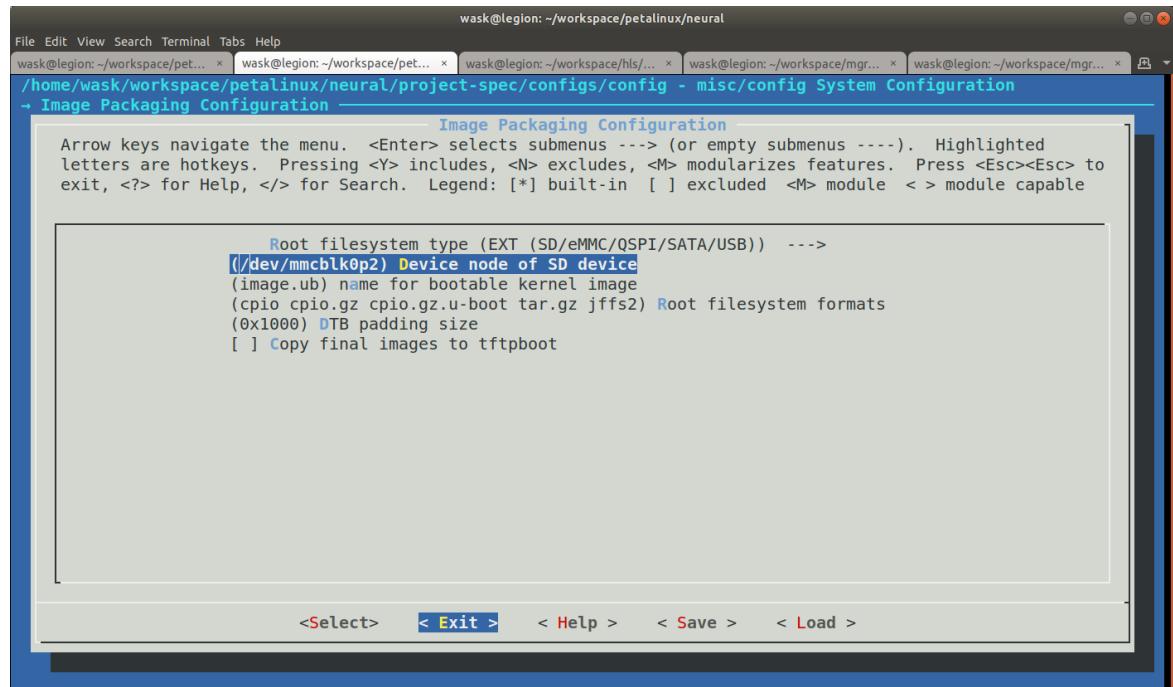
```

86     XCalcperceptron_Set_b_offset(&calcPerceptron, 16);
87 }
88
89 int main() {
90 // first layer -> set offset to 0
91 printf("Neural Network MNIST test for 10 digits\n");
92 init_PerceptronCore();
93
94 for (int j=0; j<10;j++) {
95
96     printf("load initial data for sample %d\n", j);
97
98     init_load_data(j); //set sample 0-9
99
100    XCalcperceptron_Start(&calcPerceptron);
101    while(!XCalcperceptron_IsDone(&calcPerceptron));
102
103    load_2_layer_data();
104
105    XCalcperceptron_Start(&calcPerceptron);
106    while(!XCalcperceptron_IsDone(&calcPerceptron));
107
108    for (int i = 0; i < 10; i++) {
109        printf("result[%d]: %f\n", i, resHW[i]);
110    }
111
112 }
113
114 printf("End of test\n");
115
116 return 0;
117 }
118 }
```

Connected to: Serial (/dev/ttyUSB2, 115200, 0, 8)

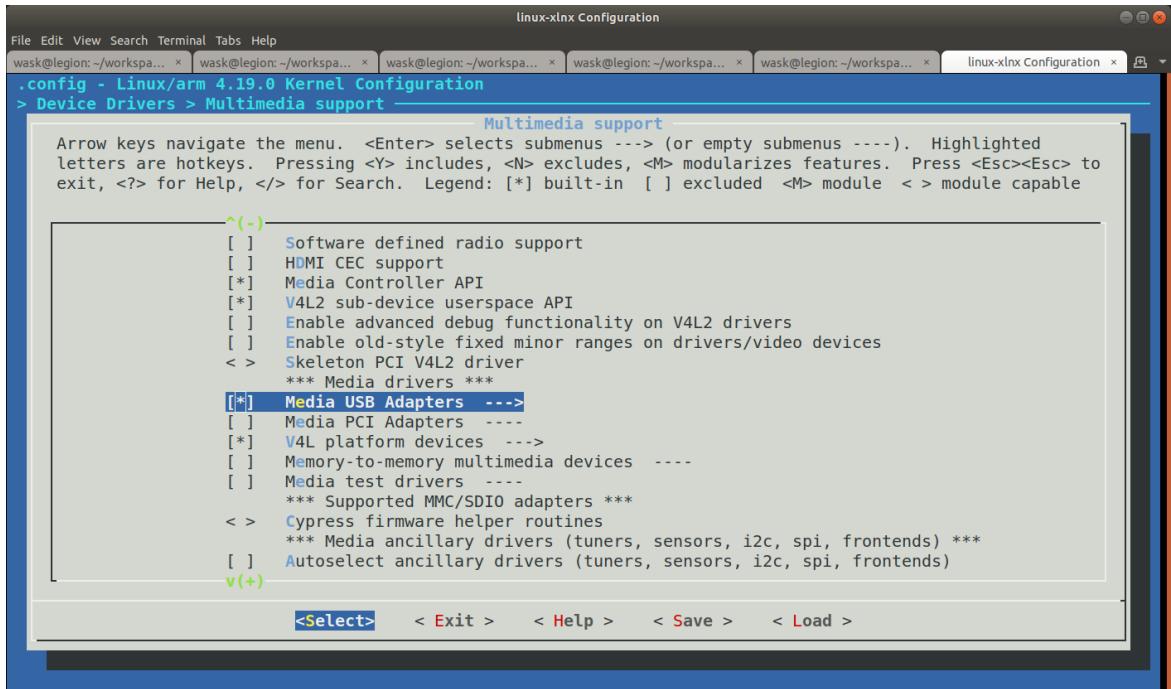
Connected to /dev/ttyUSB2 at 115200
Neural Network MNIST test for 10 digits
load initial data for sample 0
result[0]: 0.000006
result[1]: 0.000000
result[2]: 0.000011
result[3]: 0.000088
result[4]: 0.000001
result[5]: 0.000002
result[6]: 0.000000
result[7]: 0.112744
result[8]: 0.000001
result[9]: 0.000082
load initial data for sample 1
result[0]: 0.000458
result[1]: 0.000498
result[2]: 0.024287
result[3]: 0.000338
result[4]: 0.000000
result[5]: 0.000055
result[6]: 0.000153
result[7]: 0.000001
result[8]: 0.000001
result[9]: 0.000000
load initial data for sample 2
result[0]: 0.000000
result[1]: 0.023842
result[2]: 0.000084
result[3]: 0.000005

Rysunek 4.14. Wynik uruchomienia aplikacji w programie Vitis



Rysunek 4.15. Wstępna konfiguracja przy użyciu funkcji petalinux-config

rzyć własne. Dla wielu typów urządzeń tworzenie nowego sterownika nie jest konieczne. Alternatywnym rozwiązaniem jest zastosowanie sterownika UIO (ang. *Userspace Input*



Rysunek 4.16. Konfiguracja jądra PetaLinuxa

Output Driver) [27], który umożliwia dostęp do urządzenia w aplikacji z przestrzeni użytkownika. Znacznie upraszcza to proces tworzenia oprogramowania do obsługi urządzenia i zmniejsza ryzyko powstawania trudnych w debugowaniu i często niebezpiecznych dla działania systemu operacyjnego błędów. Jednak należy pamiętać, że sterowniki UIO nie są przeznaczone dla dowolnego typu sprzętu. Stosuje się je do obsługi urządzeń, generujących przerwania, posiadających pamięć, którą można zmapować i sterować urządzeniem poprzez pisanie do tej pamięci.

4.10.3. Konfiguracja systemu plików

Narzędzie Petalinux umożliwia również dostosowanie systemu plików do potrzeb użytkownika. W menu konfiguracyjnym jest wiele pakietów i bibliotek do różnych zastosowań. W zakładce *apps* użytkownik ma dostęp do aplikacji, które stworzył w obrębie danego projektu. Narzędzie Petalinux umożliwia utworzenie nowej aplikacji przy użyciu komendy:

```
petalinux-create -t apps -n <nazwa_aplikacji>
```

W projekcie zdecydowano się na użycie skryptów napisanych w języku Python i biblioteki OpenCV do rejestracji obrazu i wysyłania go poprzez port Ethernet do komputera PC. W tym celu zaznaczono następujące opcje w konfiguracji systemu plików:

- python i python-math
- python-numpy
- packagegroup-petalinux-v4lutils

4. Implementacja

- packagegroup-petalinux-opencv

Kolejnym etapem w procesie konfigurowania systemu Petalinux jest dostosowanie drzewa urządzeń (ang. *device-tree*). Drzewo urządzeń można edytować za pomocą plików system-user.dtsi oraz pl-custom.dtsi dostępnych w katalogu:

```
<katalog_projektu>/project-spec/meta-user/recipes-bsp/device-tree
```

W przypadku wykorzystania sterowników UIO należy wpisać w polu *compatible* każdego z węzłów urządzeń (pamięci BRAM) wartość "*generic-udio*" [28]. Dostęp do urządzeń jest zapewniony dzięki plikom /dev/uioX, gdzie X to numer urządzenia (zaczynając od 0 dla pierwszego urządzenia).

4.10.4. Przygotowanie obrazu systemu

Po dostosowaniu pliku *device-tree* można uruchomić komplikację petalinuxa komendą *petalinux-build*. Gdy system zostanie zbudowany, należy spakować wszystkie potrzebne pliki przy użyciu funkcji *petalinux-package*. Wynikiem tej operacji są dwa pliki dostępne w katalogu images/linux BOOT.bin i image.ub, które należy przekopiować na odpowiednią partycję karty sd. Jeśli w konfiguracji Petalinuxa wyłączona została opcja wspierania Initial RAM filesystem, należy dodatkowo wypakować system plików na drugiej partycji na karcie sd. Tak przygotowaną kartę sd można umieścić w slocie na płytce Z-turn, podłączyć zasilanie i uruchomić system. Istnieje opcja komplikacji jedynie wybranej aplikacji stworzonej przez użytkownika przy pomocy komendy *petalinux-build -c <nazwa_aplikacji>*. W przypadku wprowadzenia zmian w systemie, przed ponownym wypakowaniem systemu plików, należy sformatować partycję zawierającą system plików.

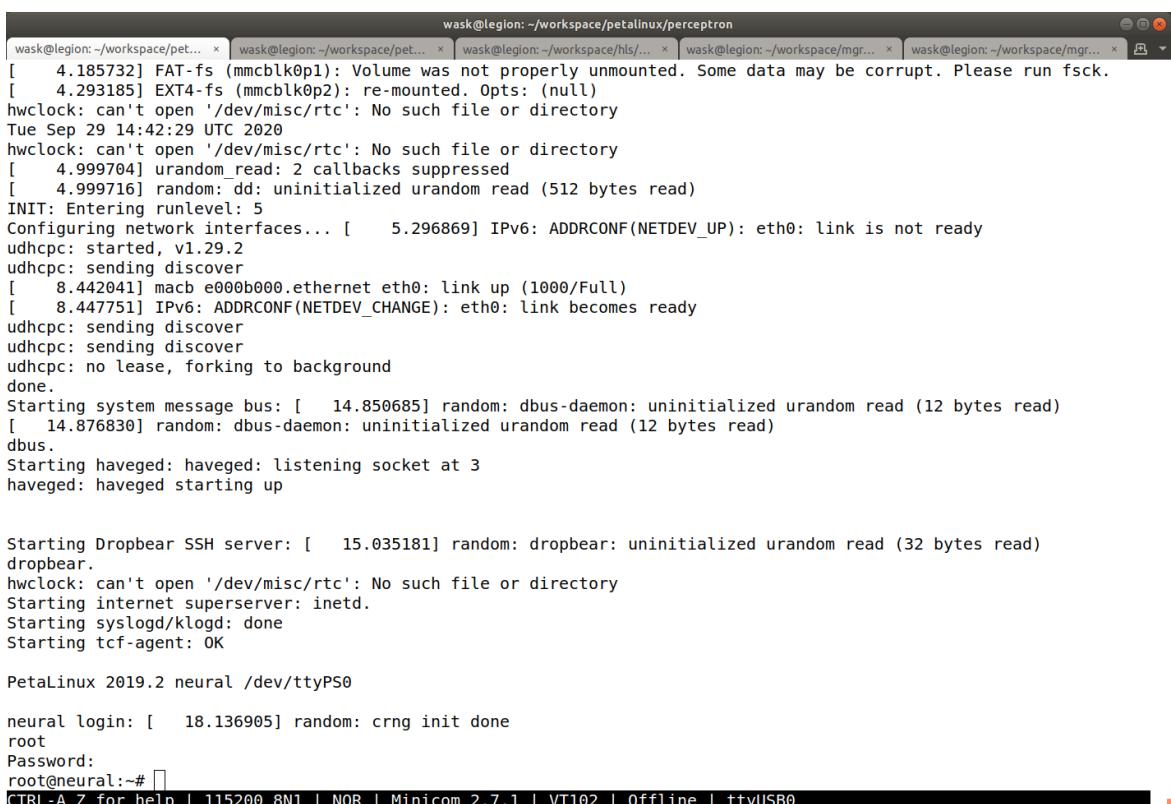
4.10.5. Uruchomienie systemu Petalinux

Komunikacja płytki Z-turn z komputerem PC odbywa się na dwa sposoby:

- przez interfejs UART przy użyciu aplikacji Minicom (Rys. 4.17)
- przez port Ethernet za pomocą kilenta SSH

Oba rozwiązania były stosowane równolegle na każdym etapie projektu.

Zaletą konsoli podłączonej za pomocą portu szeregowego jest to, że są w niej wyświetlane komunikaty jądra Linuxa. Ułatwia to debugowanie błędów, które pojawiają się w fazie uruchamiania systemu operacyjnego. Zaletą protokołu SSH jest duża przepustowość i możliwość wysyłania nawet dużych plików.



```
wask@legion: ~/workspace/petalinux/perceptron
[    4.185732] FAT-fs (mmcblk0p1): Volume was not properly unmounted. Some data may be corrupt. Please run fsck.
[    4.293185] EXT4-fs (mmcblk0p2): re-mounted. Opts: (null)
hwclock: can't open '/dev/misc/rtc': No such file or directory
Tue Sep 29 14:42:29 UTC 2020
hwclock: can't open '/dev/misc/rtc': No such file or directory
[    4.999704] urandom_read: 2 callbacks suppressed
[    4.999716] random: dd: uninitialized urandom read (512 bytes read)
INIT: Entering runlevel: 5
Configuring network interfaces... [    5.296869] IPv6: ADDRCONF(NETDEV_UP): eth0: link is not ready
udhcpc: started, v1.29.2
udhcpc: sending discover
[    8.442041] macb e000b000.ethernet eth0: link up (1000/Full)
[    8.447751] IPv6: ADDRCONF(NETDEV_CHANGE): eth0: link becomes ready
udhcpc: sending discover
udhcpc: sending discover
udhcpc: no lease, forking to background
done.
Starting system message bus: [   14.850685] random: dbus-daemon: uninitialized urandom read (12 bytes read)
[   14.876830] random: dbus-daemon: uninitialized urandom read (12 bytes read)
dbus.
Starting haveged: haveged: listening socket at 3
haveged: haveged starting up

Starting Dropbear SSH server: [   15.035181] random: dropbear: uninitialized urandom read (32 bytes read)
dropbear.
hwclock: can't open '/dev/misc/rtc': No such file or directory
Starting internet superserver: inetd.
Starting syslogd/klogd: done
Starting tcf-agent: OK

PetaLinux 2019.2 neural /dev/ttys0
neural login: [   18.136905] random: crng init done
root
Password:
root@neural:~# 
```

Rysunek 4.17. Uruchomienie systemu PetaLinux

5. Wyniki i wnioski

Celem pracy było zbadanie wpływu zastosowania akceleratorów, realizowanych w technice HLS na szybkość działania algorytmów ANN. W projekcie zaimplementowano i przetestowano kilka różnych modeli Sztucznych Sieci Neuronowych, klasyfikujących odręcznie pisane cyfry. Aby porównać rozwiązanie, realizowane w technice HLS z implementacją przy użyciu pakietu *keras*, uruchamianą na komputerze PC każdy z modeli poddano testom, które zostały podzielone na następujące części:

- uruchomienie sieci przy użyciu zbioru testowego 10000 cyfr z bazy MNIST, w celu oszacowania dokładności i szybkości działania algorytmu.
- test wykonany w czasie rzeczywistym przy użyciu modułu kamery
- zbadanie wpływu optymalizacji w narzędziu Vivado HLS.

Zbadano wpływ różnych rozwiązań optymalizacji w narzędziu Vivado HLS na szybkość wykonywanych obliczeń oraz zajętość zasobów układu FPGA. Wyniki testów z nich zamieszczone są w dalszej części rozdziału.

5.1. Test modelu sieci z jedną warstwą ukrytą

Model Sztucznej Sieci Neuronowej z jedną warstwą ukrytą zaimplementowano w skrypcie pythona z użyciem biblioteki *keras*. Wykonano uczenie sieci w 20 epokach z wielkością mini-serii równą 128. Model poddany testom w każdym z neuronów zawierał funkcję aktywacji *sigmoid* i składał się z następujących warstw:

- warstwa wejściowa (784 wejścia)
- warstwa ukryta (16 neuronów)
- warstwa wyjściowa (10 neuronów).

Test z użyciem akceleratora HLS wykonano, podając na wejście sieci 10000 obrazów zapisanych w postaci plików tekstowych, zawierających po 784 wartości w przedziale od 0 do 1. Wynik testu, widoczny na Rys. 5.1 potwierdził poprawność działania algorytmu. Osiągnięto dokładność na poziomie 94,97%, co pokrywa się z wynikiem uzyskanym z wykorzystaniem biblioteki *keras*, na komputerze PC.

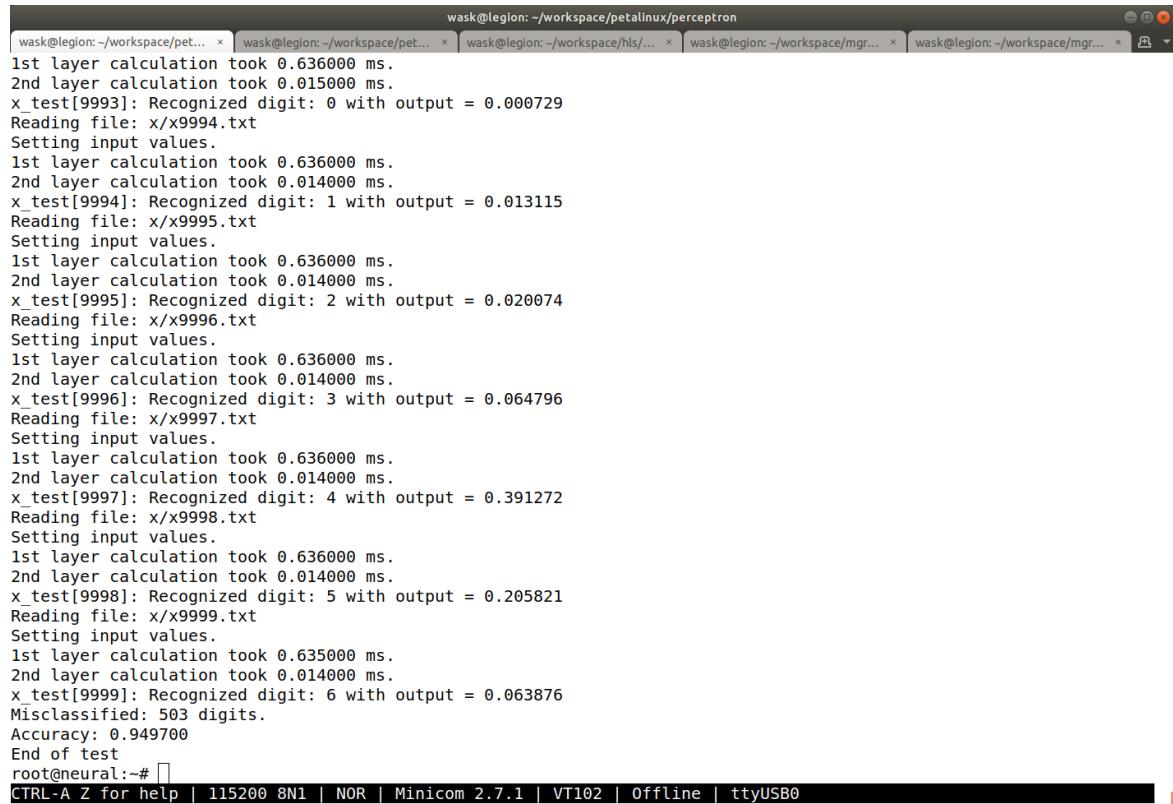
5.1.1. Test klasyfikacji cyfr z użyciem kamery

Następnym krokiem był test przeprowadzony w czasie rzeczywistym z użyciem kamery. Rozpoznawanie obiektów na obrazie w czasie rzeczywistym podzielono na 3 części:

- detekcja kształtów przypominających cyfry i odrzucenie niewłaściwych obiektów
- przygotowanie obrazów do klasyfikacji (odpowiedni rozmiar obrazu i padding)
- klasyfikacja obrazów przy użyciu ANN

Pierwszą symulację wykonano na komputerze PC przy użyciu biblioteki OpenCV i pakietu *keras*. Ze względu na sporą ilość obliczeń początkowo zdecydowano się na zarejestrowanie obrazu oraz detekcję cyfr przy użyciu biblioteki OpenCV.

5. Wyniki i wnioski



```
wask@legion: ~/workspace/petalinux/perceptron
wask@legion: ~/workspace/petalinux/perceptron
wask@legion: ~/workspace/hls/...
wask@legion: ~/workspace/mgr...
wask@legion: ~/workspace/mgr...
wask@legion: ~/workspace/mgr...

1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.015000 ms.
x test[9993]: Recognized digit: 0 with output = 0.000729
Reading file: x/x9994.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x test[9994]: Recognized digit: 1 with output = 0.013115
Reading file: x/x9995.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x test[9995]: Recognized digit: 2 with output = 0.020074
Reading file: x/x9996.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x test[9996]: Recognized digit: 3 with output = 0.064796
Reading file: x/x9997.txt
Setting input values.
1st layer calculation took 0.636000 ms.
2nd layer calculation took 0.014000 ms.
x test[9997]: Recognized digit: 4 with output = 0.391272
Reading file: x/x9998.txt
Setting input values.
1st layer calculation took 0.635000 ms.
2nd layer calculation took 0.014000 ms.
x test[9998]: Recognized digit: 5 with output = 0.205821
Reading file: x/x9999.txt
Setting input values.
1st layer calculation took 0.635000 ms.
2nd layer calculation took 0.014000 ms.
x test[9999]: Recognized digit: 6 with output = 0.063876
Misclassified: 503 digits.
Accuracy: 0.949700
End of test
root@neural:~#
```

Rysunek 5.1. Wynik testu uruchomionego na SBC Z-turn – ANN z jedną warstwą ukrytą

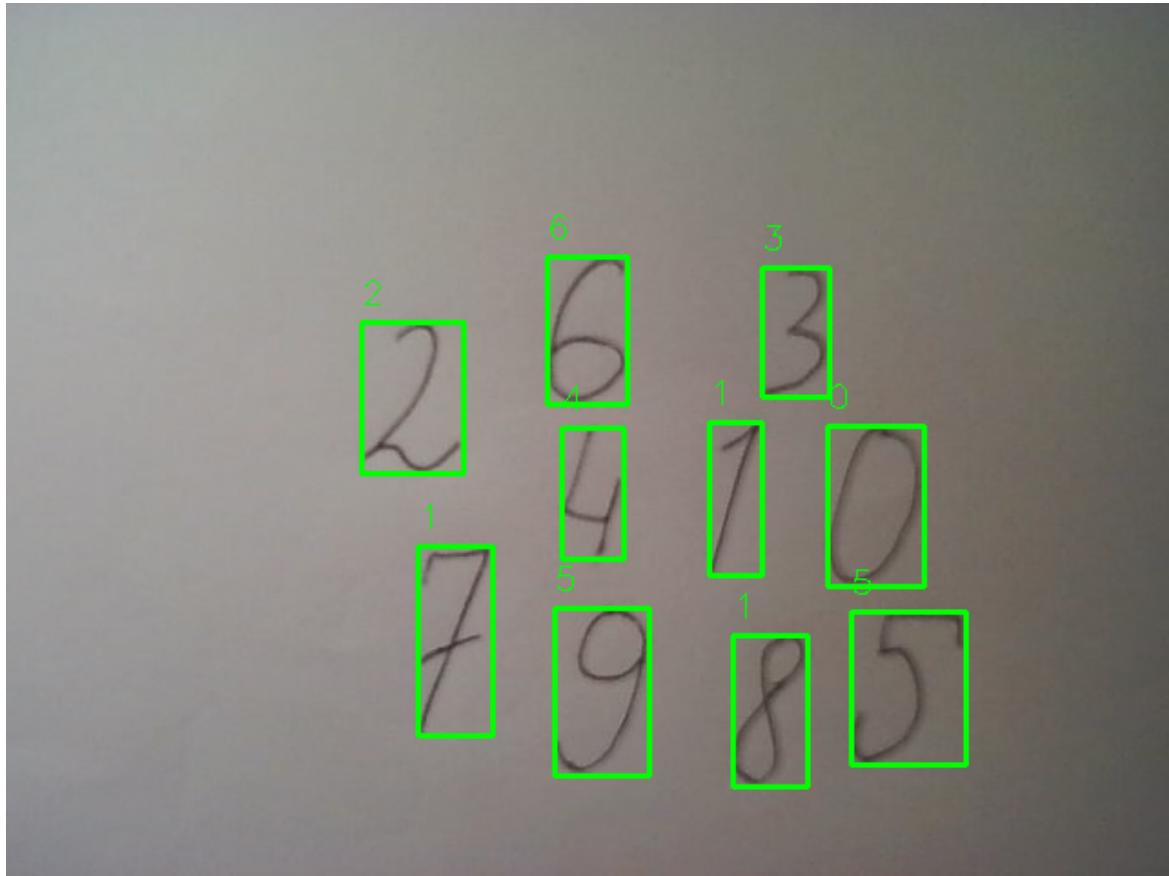
Obraz był rejestrowany przy użyciu funkcji *cv2.VideoCapture()* w rozdzielczości 640×480 pikseli. Następnym krokiem było przekształcenie barwy obrazu na skalę szarości, rozmycie obrazu oraz za pomocą funkcji *cv2.adaptiveThreshold* przekształcenie w obraz binarny z odwróconymi kolorami. Istotne jest, żeby obraz zawierał białą cyfrę na czarnym tle, ponieważ takie obrazy były w zbiorze uczącym. Następnie użyto funkcji *cv2.findContours*, która zwraca współrzędne prostokątów, w które wpisane są kontury znalezione przez algorytm. Po wyeliminowaniu niewłaściwych konturów można przejść do przygotowania obrazów do klasyfikacji.

Odpowiednio przycięty do rozmiaru 28×28 pikseli i wycentrowany obraz ręcznie pisanej cyfry może zostać poddany klasyfikacji za pomocą nauczonego wcześniej modelu ANN. W wyniku testu otrzymano wyniki przedstawione na Rys. 5.2.

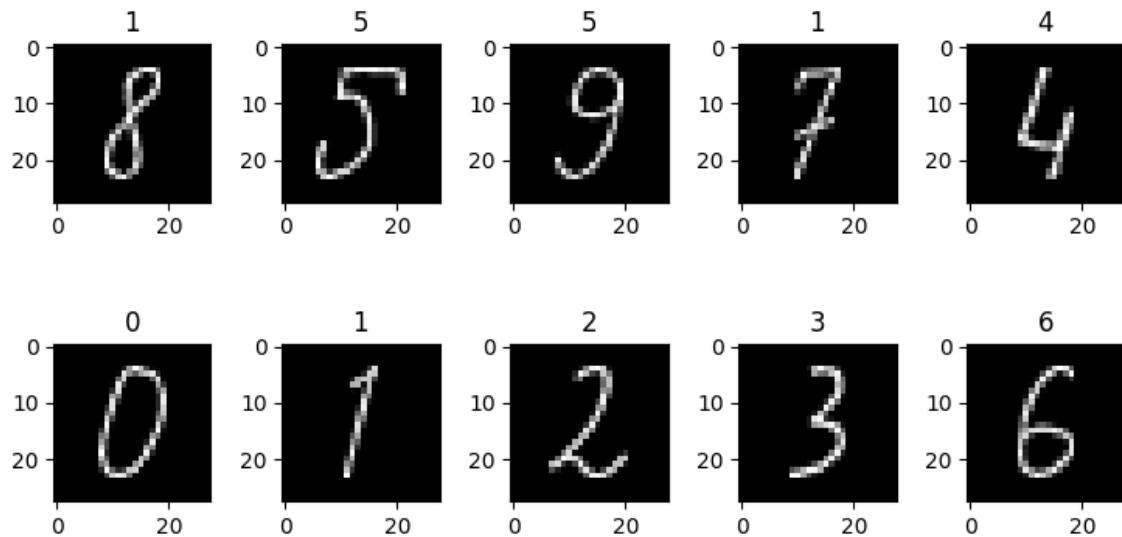
Rysunek Rys. 5.3 zawiera znalezione na obrazie cyfry i wynik klasyfikacji (nad każdą z cyfr). Widać, że 3 z 10 cyfr zostały sklasyfikowane nieprawidłowo, co daje dokładność klasyfikacji sieci 70%.

5.1.2. Test na płytce Z-turn z użyciem kamery

W teście na płytce Z-turn zastosowano metodę rejestracji obrazu taką jak na komputerze PC, jednak do rozpoznania cyfr wykorzystano akcelerator obliczeń ANN zaimplementowany w układzie FPGA. Aby umożliwić użytkownikowi wyświetlanie obrazu w czasie rzeczywistym, wykorzystano pakiety *pickle* i *socket* do wysyłania kolejnych ramek obrazu



Rysunek 5.2. Ramka obrazu podczas testu ANN z jedną warstwą ukrytą uruchomionego na PC

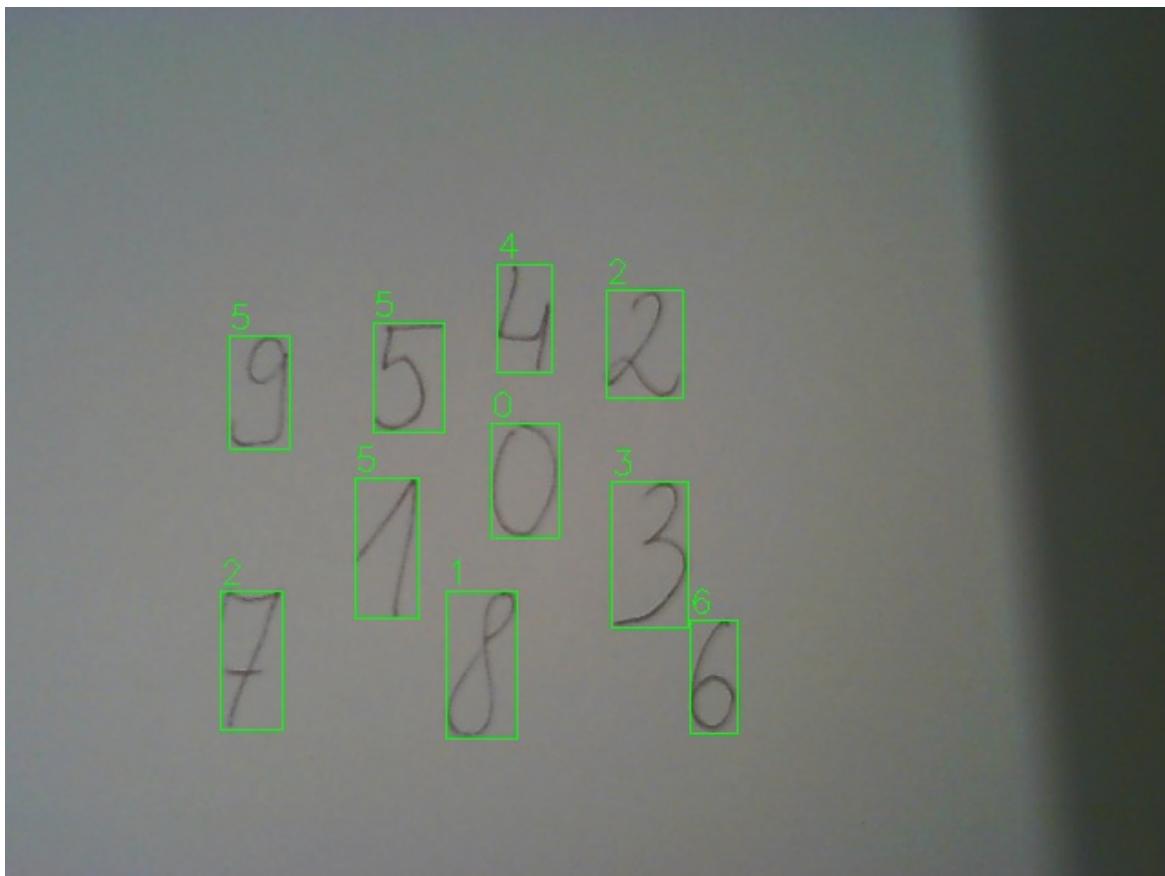


Rysunek 5.3. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na PC

przez protokół TCP (ang. *Transmission Control Protocol*). Aplikacja *camera_server.py* po uruchomieniu na płytce Z-turn oczekuje na połączenie pod zadanym adresem IP

5. Wyniki i wnioski

10.42.0.1. Uruchomienie aplikacji *camera_client* na komputerze PC podłączonym z płytą przy użyciu kabla *Ethernet* powoduje rozpoczęcie rejestrowania obrazu, detekcję i zapisanie obrazów przedstawiających cyfry do plików tekstowych. Działający w tle sterownik UIO *rtdigitrecognition* zawiera mechanizm *inotify*, pozwalający na monitorowanie zmian w pliku *input.txt*, zawierającym dane wejściowe sieci i reagowanie w przypadku zapisania nowych danych. Wynik klasyfikacji odręcznie pisanej cyfry jest zapisywany do pliku *out.txt* i odczytywany w aplikacji rejestrującej obraz, w celu wyświetlenia wyniku przy odpowiedniej cyfrze. Wynik testu przedstawiono na Rys. 5.4.

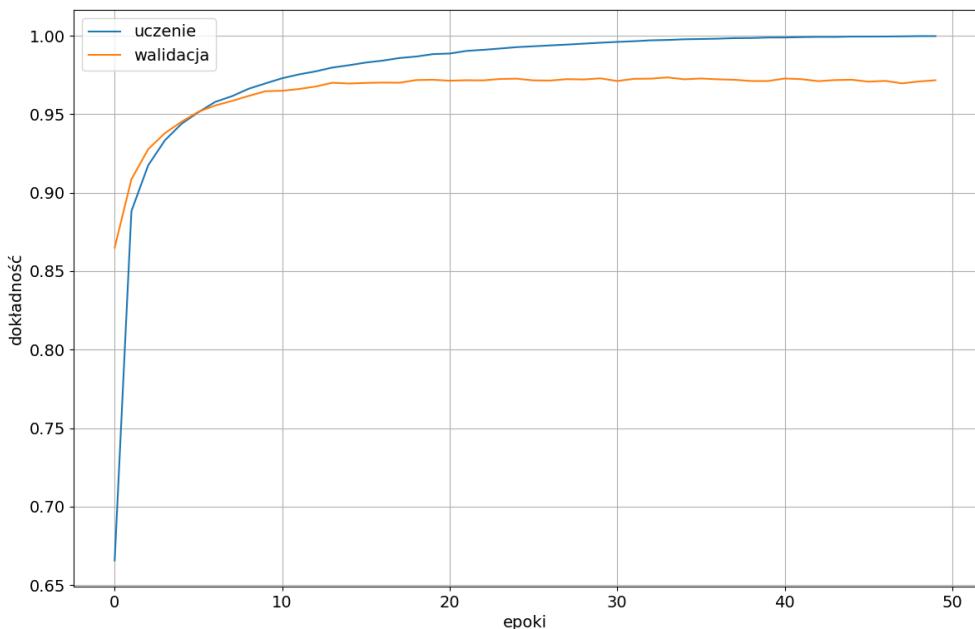


Rysunek 5.4. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na płytce Z-turn Board

Po otrzymaniu wyników pierwszego testu podjęto decyzję o modyfikacji modelu Sztucznej Sieci Neuronowej. Pierwszą zmianą było dodanie kolejnej warstwy ukrytej.

5.2. Test modelu posiadającego dwie warstwy ukryte

Dodano do istniejącego modelu kolejną warstwę ukrytą zawierającą 64 neurony. Po wykonaniu 50 epok uczenia sieci otrzymano dokładność na poziomie 97,17%. Wykres z przebiegu uczenia sieci przedstawiający dokładność klasyfikacji w kolejnych epokach przedstawiono na Rys. 5.5. Na wykresie widać, że dokładność po 20 epoce uczenia przestaje rosnąć, więc zwiększanie liczby epok nie da lepszych rezultatów i może prowadzić do przeuczenia sieci.



Rysunek 5.5. Wykres zmian dokładności w kolejnych epokach – ANN z dwoma warstwami ukrytymi

5.3. Implementacja akceleratora obliczeń ANN w Vivado HLS

Parametry modelu sieci ANN stworzonego, nauczzonego i przetestowanego w skrypcie Pythona z użyciem biblioteki keras są przekazywane do funkcji w Vivado HLS. Funkcja `calcPerceptron()` jako argument przyjmuje następujące dane:

- dane wejściowe x (obraz odręcznie pisanej cyfry)
- wagi sieci neuronowej w
- biasy sieci neuronowej b
- adres tablicy na dane wyjściowe sieci res
- model sieci w postaci tablicy $model$ zawierającej:
 - liczbę warstw sieci
 - liczbę wejść sieci
 - liczbę neuronów w każdej warstwie.

Zmieniając wartości tablicy $model$, użytkownik może zmieniać parametry sieci z poziomu aplikacji. Ograniczeniami są rozmiary tablic zawierających parametry sieci, które muszą być ustalone na etapie tworzenia projektu oraz ilość pamięci BRAM w układzie *Zynq*.

Ponadto z poziomu aplikacji użytkownik ma możliwość ustawienia parametru progu przycinania sieci (ang. *pruning threshold*). Dzięki temu możliwe jest wyeliminowanie redundantnych wag sieci, które nie mają znacznego wpływu na wartość wyjścia sieci i skrócenie obliczeń.

5.4. Dostosowanie parametrów w implementacji HLS

Implementacja umożliwiająca zmianę parametrów modelu sieci z poziomu aplikacji jest bardzo wygodnym rozwiązaniem, umożliwiającym szybkie przeprowadzenie testów dla różnych modeli sieci. Jednak zdefiniowanie w implementacji HLS parametrów sieci w postaci zmiennych, spowodowało znaczne ograniczenia w możliwościach optymalizacji.

5.4.1. Optymalizacja pętli

W przypadku optymalizacji pętli dwie najczęściej wykorzystywane dyrektywy to PIPELINE i LOOP UNROLL. Gdy pętla jest zagnieżdzona przy użyciu PIPELINE dodatkowo wykonywana jest dyrektywa LOOP FLATTEN, powodująca zamianę pętli zagnieżdzonej w pojedynczą pętlę. Wykonanie dyrektywy LOOP FLATTEN nie jest możliwe na pętli zagnieżdzonej, zawierającej zmienną w warunku zakończenia pętli wewnętrznej.

Pierwszym napotkanym problemem był brak możliwości pełnego rozwinięcia pętli przy użyciu dyrektywy:

```
#pragma HLS UNROLL
```

spowodowany zapisaniem warunku zakończenia pętli w zmiennej, będącej parametrem funkcji bloku HLS. W wyniku wykonania syntezy kompilator Vivado HLS zwracał komunikat informujący o braku możliwości pełnego rozwinięcia pętli.

Aby osiągnąć maksymalną akcelerację obliczeń i optymalizację zasobów zmieniono założenia projektu. Parametry modelu zostały ustalone na etapie implementacji bloku HLS. Dzięki temu możliwe było osiągnięcie znacznie lepszych wyników optymalizacji w narzędziu Vivado HLS.

Następnym problemem związanym z rozwijaniem pętli był komunikat Vivado HLS o braku możliwości wykonania operacji wczytania danych z jednej z tablic, spowodowany użyciem pamięci BRAM, posiadającej tylko dwa porty. Sugerowanym rozwiązaniem jest użycie innego bloku pamięci pozwalającego na równoległy odczyt dużej ilości danych lub partycjonowanie tablicy.

5.4.2. Dobranie parametru *II*

Wykonano analizę wpływu zmian parametru *II* dyrektywy PIPELINE na zajętość zasobów układu FPGA oraz czas wykonywania obliczeń. Pożądaną wartością tego parametru jest 1, jednak bywa, że wiąże się to z dużym zużyciem zasobów. W ramach kompromisu pomiędzy szybkością działania a zużyciem mocy można zmieniać parametr *II*. Wyniki testów zamieszczone w Tabeli 5.1.

5.5. Arytmetyka stałoprzecinkowa

Kolejną metodą optymalizacji kodu jest zastosowanie arytmetyki stałoprzecinkowej. Przy użyciu typu *ap_fixed<W, I>* możliwe jest ustalenie liczby bitów przeznaczonych na

Tabela 5.1. Wpływ zmian parametru H na wydajność systemu

	$H = 1$	2	3	4
opóźnienie	59.963 μs	68.630 μs	68.928 μs	70.606 μs
BRAM	5%	4%	4%	4%
DSP	15%	10%	9%	8%
FF	20%	20%	20%	19%
LUT	59%	59%	58%	58%

część całkowitą I oraz liczby bitów W, przeznaczonej na całą zmienną. Ponadto biblioteka pozwala na ustalenie, co się stanie w przypadku przekroczenia maksymalnej lub minimalnej wartości, jaka może być reprezentowana przez ustaloną liczbę bitów.

W kodzie akceleratora HLS zaimplementowano oddzielne typy dla każdej ze zmiennych reprezentujących parametry sieci. Po wykonaniu analizy raportu dla różnych konfiguracji typów stałoprzecinkowych okazało się, że wprowadzenie arytmetyki stałoprzecinkowej w tym projekcie powoduje duże zużycie zasobów (ponad 100%), przy nieznacznym spadku opóźnień. W związku z tym zdecydowano się na wykorzystanie typu zmiennoprzecinkowego.

Wykonano testy dla różnych typów wykorzystanych do deklaracji zmiennych w kodzie HLS. Użycie typu *ap_fixed* spowodowało zmniejszenie wprowadzanych opóźnień, ale liczba układów DSP urosła do wartości 362%, co uniemożliwiło implementację tego rozwiązania. Dla każdego z typów najpierw uruchomiono symulację w Vivado HLS, aby sprawdzić poprawność wykonywanych obliczeń. Typem o najmniejszej liczbie bitów, który dawał jeszcze poprawny wynik w symulacji był *ap_fixed<10,6>*. Oprócz typów stałoprzecinkowych przetestowano również typy zmiennoprzecinkowe *half* (16-bitowy) oraz *float* (32-bitowy). Test wykonano na modelu zawierającym wszystkie funkcje aktywacji typu sigmoid.

Tabela 5.2. Porównanie implementacji z wykorzystaniem różnych typów zmiennych

	float	half	ap_fixed<12,7>	ap_fixed<10,6>	ap_fixed<16,7>
opóźnienie	77.703 μs	78.063 μs	63,723 μs	63,723 μs	63,723 μs
BRAM	7%	3%	2%	2%	3%
DSP	10%	9%	362%	362%	362%
FF	36%	19%	28%	24%	39%
LUT	62%	57%	49%	49%	50%
H	4	4	4	4	4

W celu sprawdzenia, czy zajętość zasobów jest związana z obliczaniem funkcji sigmoid, powtórzono test na modelu zawierającym funkcje aktywacji ReLU oraz softmax w ostatniej warstwie. Wyniki przedstawiono w Tabeli 5.3. Widać, że zmiana funkcji aktywacji nie miała znaczącego wpływu na zużycie zasobów układu FPGA. Czas wykonywania obliczeń

5. Wyniki i wnioski

dla typów zmiennoprzecinkowych jest nieco krótszy, lecz dla typów stałoprzecinkowych zużycie elementów DSP nadal przekracza 36%.

Tabela 5.3. Porównanie implementacji z wykorzystaniem różnych typów zmiennych – funkcja aktywacji ReLU

	float	half	ap_fixed<12,7>	ap_fixed<10,6>
opóźnienie	58.306 μs	59.963 μs	75,750 μs	75,720 μs
BRAM	10%	5%	2%	2%
DSP	19%	15%	361%	361%
FF	39%	20%	25%	21%
LUT	69%	59%	46%	46%
H	1	1	1	1

5.6.

Aby przetestować wydajność klasyfikacji danego modelu sieci w implementacji programowej, uruchamianej na komputerze PC z użyciem pakietu *keras*, wykorzystano metodę *model.predict()*. Jako argument funkcja przyjmuje zbiór testowy z bazy MNIST. Implementacja modelu w HLS z możliwością zmiany parametrów sieci nie dała zadowalających rezultatów. Wyniki dla różnych architektur sieci zestawiono w Tabeli 5.4.

Tabela 5.4. Porównanie czasu wykonania implementacji software'owej przy użyciu pakietu keras z implementacją w układzie FPGA z wykorzystaniem akceleratora HLS – pierwsze podejście

implementowany model	aplikacja z użyciem keras	akcelerator HLS
FC10-sig-FC16-sig-FC16-sig-FC10-sig	0.572 ms	0.635 ms
FC16-sig-FC10-sig	0.569 ms	0.941 ms
FC64-sig-FC16-sig-FC10-sig	0.535 ms	3.796 ms
FC16-sig-FC10-soft	0.457 ms	0.938 ms
FC16-relu-FC10-soft	0.435 ms	0.942 ms

Wykonanie klasyfikacji przy użyciu pakietu keras daje porównywalne wyniki dla każdego z testowanych modeli. W przypadku użycia sieci zawierającej dużą liczbę neuronów w pierwszej warstwie ukrytej czas wykonania obliczeń znacznie rośnie.

5.6.1. Wyniki po wykonaniu optymalizacji HLS

Po wstępnych testach zdecydowano zmienić założenia implementacji i dokonać optymalizacji wybranej architektury sieci neuronowej. Wybrano następujący model:

Layer (type)	Output Shape	Param #
<hr/>		
flatten_1 (Flatten)	(None, 784)	0

```

dense_1 (Dense)           (None, 16)      12560
dense_2 (Dense)           (None, 10)       170
=====
Total params: 12,730
Trainable params: 12,730
Non-trainable params: 0
=====
```

W wyniku uczenia sieci otrzymano dokładność klasyfikacji na poziomie 94,14%.

Wybrany model, zawierający wszystkie funkcje aktywacji sigmoid, zaimplementowano w programie Vivado HLS. Ustalenie wartości parametrów liczby wejść i neuronów na etapie implementacji HLS dało większe możliwości optymalizacji. Raport z syntezą umieszczono na Rys. 5.6. Widać, że parametr II (ang. *Initiation Interval*) jest duży i jest jeszcze możliwość optymalizacji algorytmu. Zużycie zasobów jest na akceptowalnym poziomie.

W wyniku pierwszych testów otrzymano średni czas klasyfikacji jednego obrazu z użyciem akceleratora HLS 0.167110 ms , a z zastosowaniem pakietu keras 0.463247 ms . Daje to akcelerację obliczeń na poziomie 2,7.

Po przeanalizowaniu kodu i komunikatów kompilatora Vivado HLS zidentyfikowano problem, powodujący tak duży parametr II. Była to zależność niewłaściwie interpretowana przez kompilator Vivado HLS w pętli wykonującej obliczenia dla danego neuronu.

Aby osiągnąć $II=1$ potrzebny był dodatkowy bufor przechowujący sumę iloczynów wejść i wag neuronu w danej iteracji. Dopiero po wyjściu z pętli, gdy sumy iloczynów są policzone dla każdego neuronu w danej warstwie, w kolejnej pętli liczona jest funkcja aktywacji każdego z neuronów. Takie podejście umożliwia osiągnięcie parametru $II=1$ dla obu pętli.

Osiągnięto parametr $II=1$ jednak drastycznie wzrosło zużycie zasobów układu FPGA, co widać na Rys. 5.7.

Zajętość elementów BRAM wyniosła ponad 4000%, co wynika z użycia pamięci BRAM do przechowania całej tablicy zawierającej wagi sieci. Aby temu zapobiec, skorzystano z dyrektywy RESOURCE, pozwalającej na wyspecyfikowanie, jakie zasoby będą użyte do zaimplementowania danej zmiennej. Zastosowano dyrektywę zapewniającą implementację tablicy wag jako dwuportową pamięć typu ROM:

```
#pragma HLS RESOURCE variable=w core=ROM_2P_BRAM
```

Początkowo, implementację wykonano podając na blok HLS sygnał zegara o częstotliwości 250Hz. Jednak okazało się, że program nie działa poprawnie i po wykonaniu klasyfikacji jednego obrazu wchodził w stan zawieszenia. Po ustawieniu zegara na częstotliwość 100 MHz problem został rozwiązany i otrzymano poprawne wyniki. W wyniku

5. Wyniki i wnioski

The screenshot shows the Vivado HLS synthesis report for the project "calcPerceptron". The tabs at the top are "test_core.cpp", "core.cpp", "calcPerceptron_csim.log", and "Synthesis(all_aptype)(calcPerceptron)".

- Timing:**
 - Summary:** Clock: ap_clk, Target: 4.00 ns, Estimated: 6.417 ns, Uncertainty: 0.50 ns.
- Latency:**
 - Summary:** Latency (cycles) table:

	Latency (cycles)	Latency (absolute)	Interval (cycles)			
min	max	min	max	min	max	Type
16575	16575	0.106 ms	0.106 ms	16575	16575	none
 - Detail:**
 - Instance:** calcPerceptron_label0: min 15232, max 15232, Iteration Latency achieved: 952, target: -, Trip Count: 16, Pipelined: no.
 - Loop:** calcPerceptron_label2: min 1340, max 1340, Iteration Latency achieved: 134, target: 134, Trip Count: 10, Pipelined: yes.
- Utilization Estimates:**
 - Summary:** Resource usage table:

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	015055	-	-
FIFO	-	-	-	-	-
Instance	0	26	2811	3550	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	8691	-
Register	-	-	21851	-	-
Total	0	26	2466227296	0	0
Available	280	220	106400	53200	0
Utilization (%)	0	11	23	51	0
 - Detail:** Properties, Warnings, DRCs.

Rysunek 5.6. Raport po przeprowadzeniu syntezy w Vivado HLS – drugie podejście

syntezy otrzymano rozwiązańe wprowadzające niewielkie opóźnienia i akceptowalne zużycie zasobów. Wyeksportowano blok HLS do narzędzia Vivado i przeprowadzono syntezę i implementację projektu. Po wygenerowaniu Bitstreamu i eksportu implementacji sprzętowej przetestowano rozwiązanie przy użyciu programu Vitis. Średni czas klasyfikacji jednej cyfry był ponad 7 razy krótszy niż w przypadku uruchomienia na CPU. Niestety spadła również dokładność klasyfikacji o ponad 10%. Wyniki przedstawiono w Tabeli 5.5.

Summary									
Clock	Target	Estimated	Uncertainty						
ap_clk	10.00 ns	9.434 ns	1.25 ns						
Latency									
Summary									
Latency (cycles)	Latency (absolute)	Interval (cycles)							
min	max	min	max	min	max	Type			
2831	2831	28.310 us	28.310 us	2831	2831	none			
Detail									
Instance									
Loop									
Loop Name	Latency (cycles)		Initiation Interval						
	min	max	Iteration Latency	achieved	target	Trip Count Pipelined			
- memset_xbuf	783	783		1	-	- 784 no			
- Loop 2	1176	1176		3	-	- 392 no			
- memset_resbuf	25	25		1	-	- 26 no			
- memset_sum	25	25		1	-	- 26 no			
- calcPerceptron_label0	71	71	57	1	1	16 yes			
- calcPerceptron_label2	40	40	32	1	1	10 yes			
- Loop 7	256	256	32	-	-	8 no			
- Loop 8	15	15	3	-	-	5 no			
Utilization Estimates									
Summary									
Name	BRAM_18K	DSP48E	FF	LUT	URAM				
DSP	-	-	-	-	-				
Expression	-	-	0	15795	-				
FIFO	-	-	-	-	-				
Instance	0	3950273892559746							
Memory	12550	-	64	13	0				
Multiplexer	-	-	-	7576	-				
Register	0	-	103316	352	-				
Total	12550	3950377272583482			0				
Available	280	220106400	53200		0				
Utilization (%)	4482	1795	354	1096	0				
Detail									

Rysunek 5.7. Raport po przeprowadzeniu syntezy w Vivado HLS – minimalizowanie parametru II**Tabela 5.5.** Porównanie czasu wykonania implementacji software'owej przy użyciu pakietu keras z implementacją w układzie FPGA z wykorzystaniem akceleratora HLS – po optymalizacji

	czas wykonania	dokładność klasyfikacji	przyspieszenie obliczeń
CPU	0.516343 ms	91,6%	
HLS	0.070542 ms	81,0%	7.32×

5. Wyniki i wnioski

Screenshot of the Vivado HLS synthesis report for the `calcPerceptron` design.

The report tabs are: `test_core.cpp`, `core.cpp`, `calcPerceptron_csim.log`, and `Synthesis(all_aptype)(calcPerceptron)`.

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	10.234 ns	1.25 ns

Latency

Summary

Latency (cycles)	Latency (absolute)	Interval (cycles)				
min	max	min	max	min	max	Type
9924	9924	0.102 ms	0.102 ms	9924	9924	none

Detail

- Instance**
- Loop**

Loop Name	Latency (cycles)	Iteration Latency	Initiation Interval	Trip Count	Pipelined
	min	max	achieved	target	
- memset_xbuf	783	783	1	-	784 no
- Loop 2	1176	1176	3	-	392 no
- memset_resbuf	25	25	1	-	26 no
- memset_sum	25	25	1	-	26 no
- calcPerceptron_label0	7088	7088	443	-	16 no
- calcPerceptron_label2	112	112	41	8	10 yes
- Loop 7	256	256	32	-	8 no
- Loop 8	20	20	2	-	10 no

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	015782	-	-
FIFO	-	-	-	-	-
Instance	0	48	2576	5304	-
Memory	36	-	128	26	0
Multiplexer	-	-	-	15698	-
Register	0	-	37244	64	-
Total	36	48	3994836874	0	0
Available	280	220106400	53200	0	0
Utilization (%)	12	21	37	69	0

Rysunek 5.8. Raport po przeprowadzeniu syntezy w Vivado HLS – minimalizowanie parametru II oraz redukcja zużycia zasobów

6. Podsumowanie

Implementacja Sztucznych Sieci Neuronowych w układach FPGA daje wyraźne przyspieszenie czasu wykonywania obliczeń w stosunku do rozwiązań programowych. Główne założenia projektu zostały spełnione, dokonano porównania wydajności implementacji Sztucznej Sieci Neuronowej w układzie FPGA przy użyciu syntezy wysokiego poziomu do rozwiązania programowego uruchamianego na komputerze PC. W celu osiągnięcia dużego współczynnika akceleracji obliczeń wykorzystano metody optymalizacji dostępne w narzędziu Vivado HLS. Dokonano przeglądu metod umożliwiających zmniejszenie opóźnień wykonywanych operacji oraz ograniczenie zużycia zasobów układu FPGA.

Podczas projektu znaleziono elementy implementacji wprowadzające największe opóźnienia i z powodzeniem wyeliminowano lub zmniejszono ich wpływ na czas wykonania algorytmu oraz zużycie zasobów sprzętowych. W trakcie tworzenia implementacji napotkano na kilka problemów, utrudniających optymalizację w narzędziu Vivado HLS. Ostatecznie osiągnięto rozwiązanie zapewniające akcelerację obliczeń w Sztucznych Sieciach Neuronowych.

6.1. Wnioski dotyczące techniki HLS

W dziedzinie rozwijania oprogramowania można zaobserwować tendencję przechodzenia w kierunku rozwiązań wysokopoziomowych. Ta zależność jest widoczna również w przypadku programowania układów FPGA. Powstają nowe biblioteki i narzędzia usprawniające proces tworzenia implementacji akceleratorów obliczeń. W większości przypadków programista nie musi już posiadać dużej wiedzy o sprzęcie, aby osiągnąć rozwiązanie na akceptowalnym poziomie. Jednak w przypadku złożonych algorytmów bywa, że osiągnięcie implementacji optymalnej, porównywalnej z implementacją w języku HDL jest trudne lub nawet niemożliwe.

Technika HLS okazała się odpowiednim narzędziem do wykonania implementacji Sztucznych Sieci Neuronowych. Osiągnięto zadowalające rezultaty w stosunku nie-długim czasie a, możliwość implementowania algorytmu w języku C++, ułatwiła wprowadzanie zmian w projekcie. Dzięki odpowiednim dyrektywom i parametrom można odpowiednio zoptymalizować dane rozwiązanie. Ustawiając odpowiednie parametry możliwe jest osiągnięcie kompromisu pomiędzy zużyciem zasobów, a szybkością wykonywania obliczeń w algorytmie ANN. Problematyczny w procesie optymalizacji był czas wykonywania syntezy w narzędziu Vivado HLS oraz implementacji w środowisku Vivado, który w niektórych przypadkach wynosił kilka godzin.

6.2. Możliwości rozwoju projektu

Użycie w projekcie metody HLS umożliwiło znalezienie elementów implementacji wprowadzających duże opóźnienia i powodujących wykorzystanie dużej ilości zasobów

6. Podsumowanie

układu FPGA. W końcowym etapie projektu problematyczny okazał się długi czas syntezy i implementacji. Projekt można rozwinąć, stosując bardziej skomplikowane architektury Sztucznych Sieci Neuronowych. Z powodu ograniczeń czasu projektu nie udało się zaimplementować warstwy splotowej, która umożliwia zmniejszenie liczby parametrów sieci przy jednoczesnym osiągnięciu wysokiej dokładności klasyfikacji obrazów. Dodatkowym usprawnieniem może być zastosowanie bardziej wydajnego interfejsu komunikacji akceleratora HLS z procesorem w układzie Zynq, zapewniającego wydajniejszy dostęp do pamięci oraz obrazu z kamery w czasie rzeczywistym.

W pracy zaprezentowano zastosowanie Sztucznej Sieci Neuronowej do rozpoznawania obiektów znajdujących się na obrazie z kamery podłączanej przez USB w czasie rzeczywistym. Część algorytmu odpowiedzialna za detekcję cyfr na obrazie była uruchamiana na procesorze ARM przy użyciu biblioteki OpenCV, co wprowadzało spore opóźnienia i ograniczało wartość FPS oraz maksymalną liczbę cyfr możliwych do znalezienia w danym momencie. W przypadku zastosowania systemu do zadań klasyfikacji obrazów możliwym usprawnieniem projektu jest zastosowanie kamery podłączanej przez interfejs równoległy bezpośrednio do układu FPGA w celu zminimalizowania opóźnień. Dodatkowym usprawnieniem może być dalsza optymalizacja algorytmu przy użyciu techniki HLS w celu osiągnięcia maksymalnej akceleracji obliczeń i minimalnego zużycia zasobów logiki programowej.

Zaletą rozwiązania zawartego w projekcie jest duża uniwersalność. Dzięki możliwości zmiany parametrów sieci w funkcji akceleratora użytkownik ma możliwość dokonywania zmian w modelu ANN z poziomu aplikacji. Daje to duże możliwości zastosowania systemu w różnych zadaniach związanych z dziedziną *Machine Learning*. Zastosowanie układu FPGA umożliwia znalezienie optymalnego rozwiązania zapewniającego kompromis pomiędzy zużyciem mocy a szybkością działania systemu, co jest bardzo istotne w przypadku zastosowania w systemach wbudowanych.

Bibliografia

- [1] D. Kriesel, *A Brief Introduction to Neural Networks*. 2007. adr.: <http://www.dkriesel.com>.
- [2] R. Tadeusiewicz, *Sieci neuronowe*. 1993.
- [3] A. Omondi i J. Rajapakse, "FPGA Implementations of Neural Networks", 2006.
- [4] S. S. Haykin i in., *Neural networks and learning machines/Simon Haykin*. 2009.
- [5] I. Goodfellow, Y. Bengio i A. Courville, *Deep Learning*. MIT Press, 2016, <http://www.deeplearningbook.org>.
- [6] M. A. Nielsen, *Neural networks and deep learning*. Determination press San Francisco, CA, 2015, t. 2018.
- [7] R. Tadeusiewicz i M. Szaleniec, *Leksykon sieci neuronowych*. Projekt Nauka. Fundacja na rzecz promocji nauki polskiej, 2015.
- [8] *Grow your business with computer vision technology*, Dostęp zdalny (03.10.2020): <https://medium.com/computer-vision-technology-drives-the-future/computer-vision-technology-drives-the-future-af43a31f7699>.
- [9] Nvidia, *CUDA Zone*, Dostęp zdalny (02.10.2020): <https://developer.nvidia.com/cuda-zone>, 2020.
- [10] *Vitis-AI-Library*, Dostęp zdalny (03.10.2020): <https://github.com/Xilinx/Vitis-AI/tree/master/Vitis-AI-Library>, 2020.
- [11] *Vitis AI*, Dostęp zdalny (03.10.2020): <https://www.xilinx.com/products/design-tools/vitis/vitis-ai.html>.
- [12] *Zynq-7000 SoC*, Dostęp zdalny (03.10.2020): <https://www.xilinx.com/products/silicon-devices/soc/zynq-7000.html>.
- [13] MYIR, *Company Profile*, Dostęp zdalny (22.09.2020): <http://www.myirtech.com/company.asp>, 2020.
- [14] *Z-turn Board*, Dostęp zdalny (04.10.2020): <https://www.xilinx.com/products/boards-and-kits/1-7i4a8t.html>.
- [15] *Z-turn IO Cape Extension Board for Z-turn Board-Welcome to MYIR*, Dostęp zdalny (05.10.2020): <http://www.myirtech.com/list.asp?id=532>.
- [16] *MY-CAM011B BUS Camera Module*, Dostęp zdalny (04.10.2020): <http://www.myirtech.com/list.asp?id=534>.
- [17] *Z-turn IO Cape Schematic*, Dostęp zdalny (24.09.2020): <http://www.myirtech.com/download/Zynq7000/Z-TURNIOCAPE-SCH.pdf>, 2015.
- [18] *Adapter złącza FPC/FFC 0,5mm 30-pin na DIP*, Dostęp zdalny (03.10.2020): <https://kamami.pl/zlaczka-ffc-fpc-zif/579387-adapter-zlaczka-fpcffc-05mm-30-pin-na-dip.html>.
- [19] *MY-CAM001U USB Digital Camera Module*, Dostęp zdalny (04.10.2020): <https://kamami.pl/akcesoria-myir/233514-my-cam001u-usb-digital-camera-module.html>.
- [20] Xilinx, *Vivado Design Suite User Guide (UG973 (v2019.2))*, 2019.

6. Bibliografia

- [21] *PetaLinux Tools Documentation Reference Guide (UG1144 (v2019.2))*, 2019.
- [22] *Vivado Design Suite User Guide High-Level Synthesis (UG871 (v2019.2))*, 2020.
- [23] *SDAccel Development Environment Help – pragma HLS pipeline*, Dostęp zdalny (04.10.2020): https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/fde1504034360078.html.
- [24] *HLS Pragmas*, Dostęp zdalny (30.09.2020): https://www.xilinx.com/html_docs/xilinx2018_3/sdsoc_doc/hls-pragmas-okr1504034364623.html, 2019.
- [25] Y. LeCun i C. Cortes, “MNIST handwritten digit database”, 2010. adr.: <http://yann.lecun.com/exdb/mnist/>.
- [26] C. Niroshan, *Interfacing a USB WebCam and Enable USB Tethering on ZYNQ-7000 AP SoC Running Linux*, Dostęp zdalny (29.09.2020): <https://medium.com/developments-and-implementations-on-zynq-7000-ap/interfacing-a-usb-webcam-and-enable-usb-tethering-on-zynq-7000-ap-soc-running-linux-1ba6d836749d>, 2017.
- [27] L. Hans-Jürgen Koch Linux developer, *The Userspace I/O HOWTO*, Dostęp zdalny (29.09.2020): <https://www.kernel.org/doc/html/v4.14/driver-api/uio-howto.html>, 2006.
- [28] Confluence, *Testing UIO with Interrupt on Zynq Ultrascale*, Dostęp zdalny (29.09.2020): <https://xilinx-wiki.atlassian.net/wiki/spaces/A/pages/18842490/Testing+UIO+with+Interrupt+on+Zynq+Ultrascale>, 2019.

Wykaz symboli i skrótów

ANN – ang. *Artificial Neural Network*
API – ang. *Application Programming Interface*
ASIC – ang. *Application-Specific Integrated Circuit*
BSP – ang. *Board Support Package*
CUDA – ang. *Compute Unified Device Architecture*
DVP – ang. *Digital Video Port*
FPGA – ang. *Field Programmable Gate Array*
GPU – ang. *Graphics Processing Unit*
HLS – ang. *High Level Synthesis*
II – ang. *Initiation Interval*
MPSoC – ang. *Multi-Processor System-on-Chip*
PL – ang. *Programmable Logic*
PS – ang. *Processing System*
SBC – ang. *Single Board Computer*
SoC – ang. *System on Chip*
SSH – ang. *Secure Shell*
TCP – ang. *Transmission Control Protocol*
UIO – ang. *Userspace Input/Output (Driver)*
V4L2 – ang. *Video for Linux 2*

Spis rysunków

1.1.	Schemat w pełni połączonej jednokierunkowej Sieci Neuronowej (wygenerowany na stronie: http://alexlenail.me/NN-SVG/)	12
1.2.	Model neuronu [4]	13
1.3.	Wykres funkcji ReLU	14
1.4.	Wykres funkcji sigmoid	14
1.5.	Model Perceptronu Wielowarstwowego	15
1.6.	Model Głębokiego Uczenia sieci [5]	16
1.7.	Splot macierzy 6x6 przy użyciu filtra o rozmiarze 3x3	18
1.8.	Max-pooling o rozmiarze 2x2	19
1.9.	Rozpoznawanie obiektów na obrazie z kamery samochodu autonomicznego [8]	20
2.1.	Struktura narzędzia Vitis-AI [11]	22
3.1.	Architektura serii ZYNQ-7000 SoC [12]	25
3.2.	Płytki Z-turn-Board 7020 [14]	27
3.3.	Płytki rozszerzeniowa Z-turn IO Cape [15]	28
3.4.	Moduł kamery MY-CAM011B BUS Camera Module [16]	29
3.5.	Schemat portu DVP na płytce rozszerzeniowej IO-Cape [17]	30

3.6. Adapter złącza FPC/FFC o rastrze 0,5 mm na otwory DIP [18]	31
3.7. Moduł kamery MY-CAM001U USB Digital Camera Module [19]	31
4.1. Schemat blokowy systemu	33
4.2. Zrzut ekranu przedstawiający schemat systemu w środowisku Vivado	35
4.3. Zrzut ekranu przedstawiający szczegółowy schemat części systemu <i>neural_net</i> w środowisku Vivado	36
4.4. Proces projektowania przy użyciu metody HLS [22]	37
4.5. Widok narzędzia Schedule Viewer w Vivado HLS	38
4.6. Raport po przeprowadzeniu syntezy w Vivado HLS	39
4.7. Wynik zastosowania dyrektywy <i>pipeline</i> w celu optymalizacji pętli, (A) – bez dyrektywy <i>pipeline</i> , (B) – z zastosowaniem dyrektywy <i>pipeline</i> [23]	40
4.8. Fragment zbioru odręcznie pisanych cyfr MNIST [25]	42
4.9. Wykres zmian dokładności w kolejnych epokach – ANN z jedną warstwą ukrytą	43
4.10. Wynik poprawnie przeprowadzonej symulacji w Vivado HLS	44
4.11. Wynik poprawnie przeprowadzonej implementacji w narzędziu Vivado	45
4.12. Tworzenie nowego projektu w programie Vitis	46
4.13. Wybór pliku .xsa z opisem konfiguracji sprzętu w programie Vitis	47
4.14. Wynik uruchomienia aplikacji w programie Vitis	48
4.15. Wstępna konfiguracja przy użyciu funkcji petalinux-config	48
4.16. Konfiguracja jądra PetaLinuxa	49
4.17. Uruchomienie systemu PetaLinux	51
5.1. Wynik testu uruchomionego na SBC Z-turn – ANN z jedną warstwą ukrytą . .	54
5.2. Ramka obrazu podczas testu ANN z jedną warstwą ukrytą uruchomionego na PC	55
5.3. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na PC	55
5.4. Wynik testu ANN z jedną warstwą ukrytą uruchomionego na płytce Z-turn Board	56
5.5. Wykres zmian dokładności w kolejnych epokach – ANN z dwoma warstwami ukrytymi	57
5.6. Raport po przeprowadzeniu syntezy w Vivado HLS – drugie podejście	62
5.7. Raport po przeprowadzeniu syntezy w Vivado HLS – minimalizowanie parametru II	63
5.8. Raport po przeprowadzeniu syntezy w Vivado HLS – minimalizowanie parametru II oraz redukcja zużycia zasobów	64

Spis tabel

3.1. Porównanie cen płyt z układami Zynq firmy Xilinx	26
3.2. Specyfikacja techniczna układu Artix-7	26
3.3. Porównanie testowanych modułów kamer	27

5.1. Wpływ zmian parametru H na wydajność systemu	59
5.2. Porównanie implementacji z wykorzystaniem różnych typów zmiennych	59
5.3. Porównanie implementacji z wykorzystaniem różnych typów zmiennych – funkcja aktywacji ReLU	60
5.4. Porównanie czasu wykonania implementacji software'owej przy użyciu pakietu keras z implementacją w układzie FPGA z wykorzystaniem akceleratora HLS – pierwsze podejście	60
5.5. Porównanie czasu wykonania implementacji software'owej przy użyciu pakietu keras z implementacją w układzie FPGA z wykorzystaniem akceleratora HLS – po optymalizacji	63