Uniwersytet Jagielloński w Krakowie

Wydział Fizyki, Astronomii i Informatyki Stosowanej

Wojciech Lepich

Nr albumu: 1146600

Rozpoznawanie cyfr przez sieć neuronową zaimplementowaną na układzie FPGA

Praca licencjacka
na kierunku Informatyka

Praca wykonana pod kierunkiem
dr. Grzegorza Korcyla
z Zakładu Technologii Informatycznych

Kraków 2020

Oświadczenie autora pracy

Świadom odpowiedzialności prawnej oświadczam, że niniejsza praca dyplomowa została napisana przeze mnie samodzielnie i nie zawiera treści uzyskanych w sposób niezgodny z obowiązującymi przepisami.

Oświadczam również, że przedstawiona praca nie była wcześniej przedmiotem procedur związanych z uzyskaniem tytułu zawodowego w wyższej uczelni.

.....	.....
Kraków, dnia	Podpis autora pracy

Oświadczenie kierującego pracą

Potwierdzam, że niniejsza praca została przygotowana pod moim kierunkiem i kwalifikuje się do przedstawienia jej w postępowaniu o nadanie tytułu zawodowego.

.....	.....
Kraków, dnia	Podpis kierującego pracą

Spis treści

1 Wstęp

Tutaj wstęp

2 Teoria

2.1 Architektura FPGA

Field-Programmable Gate Array (FPGA) to układy scalone, które mogą być elektronicznie przeprogramowane bez potrzeby demontażu samego układu z urządzenia. W porównaniu do układów ASIC znacznie taniej zaprojektować pierwszy działający układ. Elastyczna natura układów FPGA wiąże się z większym zużyciem powierzchni krzemu, opóźnień oraz zużycia energii. (FPGA architecture: survey and challenges)

Dodać przypis

Podstawowa struktura układów FPGA składa się z różnych bloków logicznych, które mogą być łączone ze sobą w zależności od wymagań projektowych. Przykładami takich bloków są: DSP (jednostka przeprowadzająca obliczenia dodawania/mnożenia), LUT (Look-Up Table, de facto tablica prawdy dowolnej funkcji boolowskiej), Flip Flop (przechowują wynik LUT), BRAM (Block RAM, pamięć dwuportowa, jest w stanie przechowywać względnie dużą ilość danych).

Układy FPGA przeważnie pracują na kilku-, kilkunastukrotnie niższych częstotliwościach niż CPU. Osiągają wysoką wydajność dzięki masywnemu zrównolegleniu obliczeń.

Programowanie FPGA polega na pisaniu logiki w językach HDL (Hardware Description Language) takimi jak VHDL czy też Verilog. Napisana logika definiuje zachowanie układu FPGA. Gotowy opis logiki syntetyzuje się, czyli generuje połączenia pomiędzy zasobami układu. Kolejnym etapem jest implementacja — odzworowanie połączeń w konkretnym układzie.

HLS (High-Level Synthesis) to proces ułatwiający pisanie skomplikowanej logiki. Algorytmy można pisać w językach wysokiego poziomu, takich jak C, C++, SystemC. Przygotowany kod jest transpilowany poprzez odpowiedni kompilator HLS do języka RTL (Register-Transfer Level; język opisu sprzętu na poziomie bramek i rejestrów), a ten może być zaimplementowany na układzie.

2.2 Przetwarzanie obrazu

Cyfrowe przetwarzanie obrazu jest problemem wymagającym dużych mocy obliczeniowych ze względu na ilość danych do przetworzenia. Nieskompresowany kolorowy obraz z pikselami w formacie RGB (po 8 bitów na kolor) o wysokości 720 pikseli i szerokości 1280 pikseli to 22118400 bitów ($\approx$ 2,5MB). Obraz przetwarzany w czasie rzeczywistym, na przykład z kamery, zwielokrotnia tę liczbę o liczbę klatek na sekundę (przy trzydziestu klatkach na sekundę liczba danych rośnie do około 79 megabajtów na sekundę). Należy również pamiętać, że dane są dwuwymiarowe co jest ważne przy problemach związanych z rozpoznawaniem wzorców, klasyfikacją przedmiotów na obrazie, filtrowania w celu rozmazania lub wyostrzenia obrazów, itp.

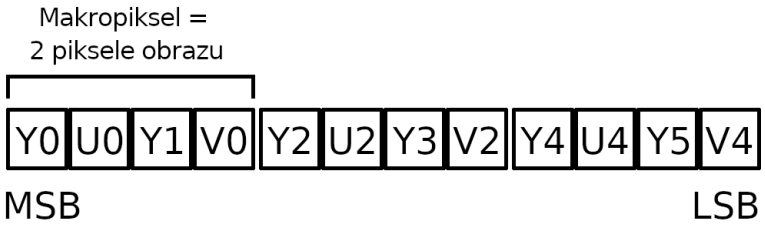
2.2.1 Formaty pikseli

Jest wiele modeli przestrzeni barw (a co za tym idzie, sposobów kodowania pikseli) między innymi:

- RGB, używany w aparatach, skanerach, telewizorach
- CMYK, używany w druku wielobarwnym

- HSV
- YUV

Składowe dwóch ostatnich przestrzeni barw oddzielają informację o jasności od informacji o kolorach. Model barw YUV składa się z kanału luminacji Y oraz kanałów kodujących barwę U oraz V, są to kolejno składowa niebieska i składowa czerwona. W projekcie użyty jest format pikseli YUY2 (znany też pod nazwą YUYV), w którym na dwa piksele przypadają 32 bity. Licząc od najstarszego bitu pierwsze osiem



Rysunek 1: Schemat formatu pikseli YUV2

bitów przypada na Y0, to jest luminacja pierwszego piksela, następne osiem bitów na U0, kolejne osiem bitów to luminacja drugiego piksela, a pozostałe bity to składowa czerwona V0. Dla obydwóch pikseli składowe U i V są wspólne. Co istotne w projekcie, łatwo oddzielić luminację, która jest używana w przetwarzaniu obrazu.

2.3 Sieci neuronowe

Sztuczna sieć neuronowa (SSN) jest modelem zdolnym do odwzorowania złożonych funkcji. Najprostsze sieci są zbudowane ze sztucznych neuronów, z których każdy posiada wiele wejść oraz jedno wyjście, które może być połączone z wejściami wielu innych neuronów. Każde z wejść neuronu jest związane ze znalezioną w procesie trenowania wagą. Wartość wyjścia to obliczony wynik funkcji aktywacji z sumy ważonych wejść. Sieć może mieć wiele warstw neuronów ukrytych, których wejściami są wyjścia neuronów z poprzedniej warstwy.

Sieci neuronowe są stosowane w problemach związanych z predykcją, klasyfikacją, przetwarzaniem i analizowaniem danych. Do ich zastosowania nie jest potrzebna znajomość algorytmu rozwiązania danego problemu. Obliczenia w sieciach są wykonywane równolegle w każdej warstwie, dzięki czemu implementacja sieci na układzie FPGA może działać wielokrotnie szybciej niż na CPU, pomimo niższej częstotliwości układu.

3 Opis projektu

3.1 Zarys projektu

Celem projektu jest implementacja systemu do rozpoznawania cyfr w czasie rzeczywistym. Cel zrealizowano poprzez implementację wtyczki GStreamera, wykorzystującej sieć neuronową, na układzie Xilinx Zynq MPSoC oraz stworzenie odpowiedniego potoku danych korzystając z bibliotek GStreamer. Zadaniami spoczywającymi na innych elementach potoku jest obsługa kamery, kadrowanie i skalowanie obrazu oraz wyświetlenie go na końcowym urządzeniu.

3.2 Platforma

Sprzęt wykorzystany w projekcie to Xilinx Zynq UltraScale+ MPSoC ZCU104. Na jednym układzie znajduje się czterordzeniowy procesor ARM Cortex-A53, dwurdzeniowy procesor ARM Cortex-R5, układ graficzny Mali-400 oraz zasoby FPGA. Całość projektu została oparta o platformę Xilinx reVISION. Przetwarzane dane dostarczane są z kamery USB, która była dołączona w zestawie z płytą Zynq. Urządzeniem końcowym jest telewizor połączony przewodem HDMI z płytą.

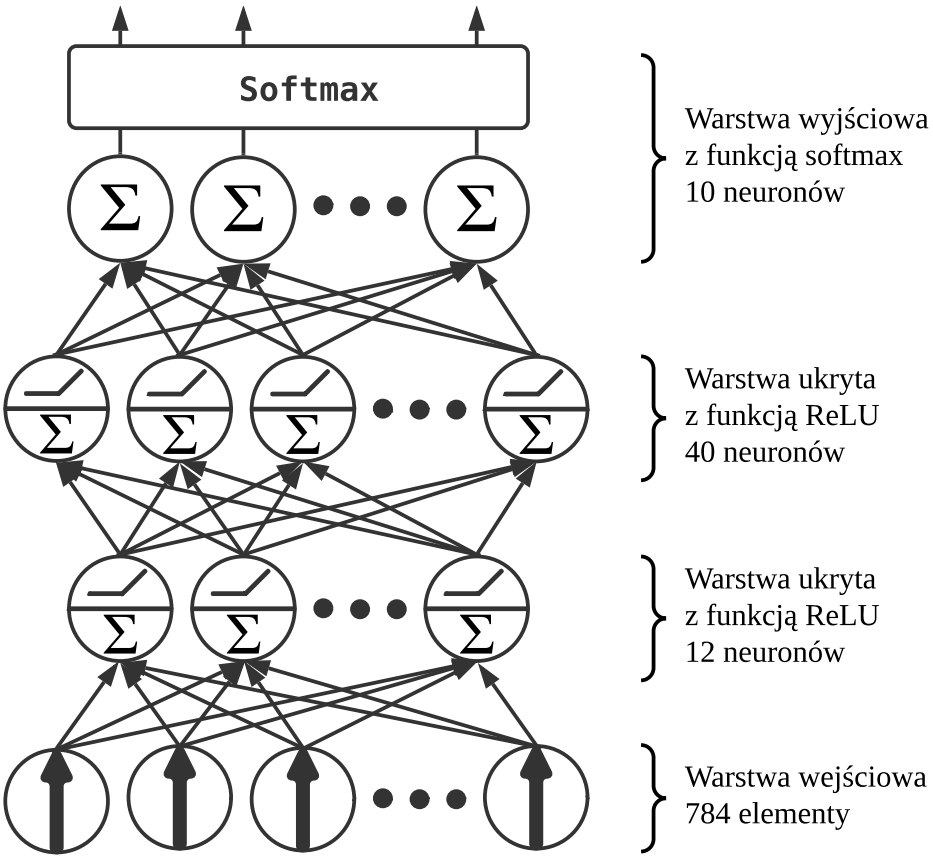
3.3 Sieć neuronowa

Architektura sieci została dobrana uwzględniając dostępne zasoby programowalnej logiki na płycie, a także możliwości sprzętu na którym dokonywana była jej synteza. Dla problemu klasyfikowania obrazów dobrze nadają się sieci splotowe (konwolucyjne, ang. convolutional neural networks — CNN), których przykładem jest popularna sieć LeNet-5. Architektura ta zawiera zarówno w pełni połączone warstwy oraz warstwy splotowe i łączące. Niestety z powodu ograniczeń sprzętowych w pracy nie została użyta ta architektura.

przypis

Model wykorzystany w projekcie posiada 2 warstwy ukryte, posiadające kolejno 12 i 40 neuronów aktywowanych funkcją ReLU, i warstwy wyjściowej złożonej z 10 neuronów z funkcją aktywacji softmax. Do stworzenia sieci wykorzystano bibliotekę TensorFlow. Sieć uczona była na danych z bazy MNIST składającej się łącznie z 70000 przykładów cyfr na obrazach o wielkości 28×28 pikseli, z których każdy przedstawiony jest jako wartość od 0 (kolor czarny) do 255 (kolor biały). Próbki zostały podzielone na zbiór uczący, liczący 60000 próbek, oraz zbiór do testów z pozostałych 10000 cyfr. Cyfry składają się z białych pikseli, tło jest czarne. Dokonano prób trenowania sieci przetworzonymi danymi, w których kolory były odwrócone, natomiast wytrenowane modele w trakcie testów nie przekraczały progu czterdziestu procent dobrze zaklasyfikowanych obrazów.

przypis



Rysunek 2: Schemat użytej architektury.

3.4 hls4ml

3.4.1 Idea hls4ml

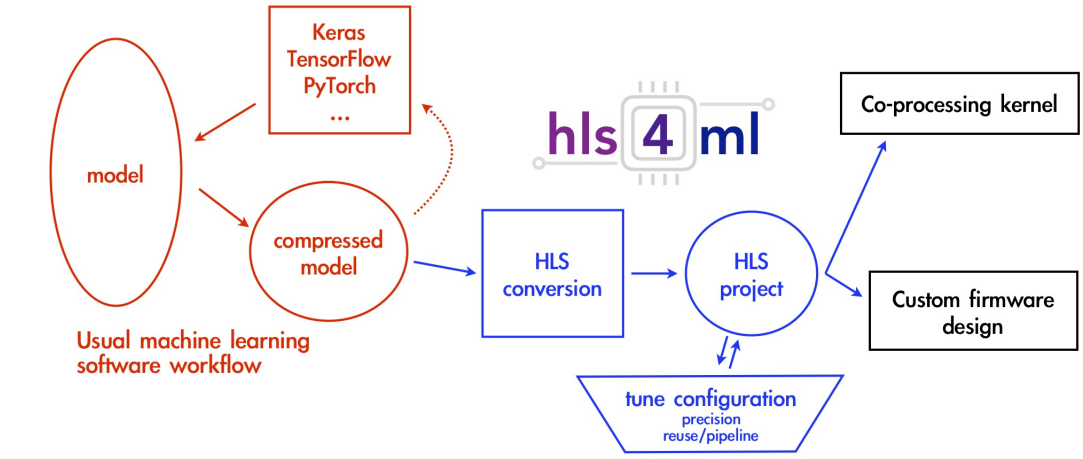
Celem projektu hls4ml jest wygenerowanie kodu C++ na podstawie zapisanego modelu z TensorFlow. Czerwona część schematu pokazuje ogólną organizację pracy przy projektowaniu odpowiedniego modelu uczenia maszynowego. Niebieska część należy do hls4ml, który tłumaczy dostarczony model z wagami do syntetyzowalnego kodu, który następnie można włączyć do większego projektu lub zaimplementować jako samodzielną część na FPGA. Generowany projekt jest parametryzowany przez plik konfiguracyjny yml zawierający ścieżkę do pliku z zapisanym modelem wraz z wagami, typy danych kodujące wartości wag, nazwę docelowego układu FPGA raz parametry optymalizacji dotyczące zużycia zasobów — większa ilość zasobów oznacza zrównoleglenie większej części obliczeń.

3.4.2 Precyzja danych

Typ danych używany w przekonwertowanym modelu to duże liczby stałoprzecinkowe (`ap_fixed`) oraz liczby całkowite (`ap_int`). Precyzję obu typów można ustalić do jednego bita. Obliczenia przeprowadzane na liczbach o mniejszej precyzji są umożliwiają większe zrównoleglenie obliczeń, natomiast zbyt niska precyzja może poskutkować bezużytecznością zsyntetyzowanej sieci. Aby odpowiednio dobrać precyzję wag skorzystano z pythonowej biblioteki hls4ml.profiling.

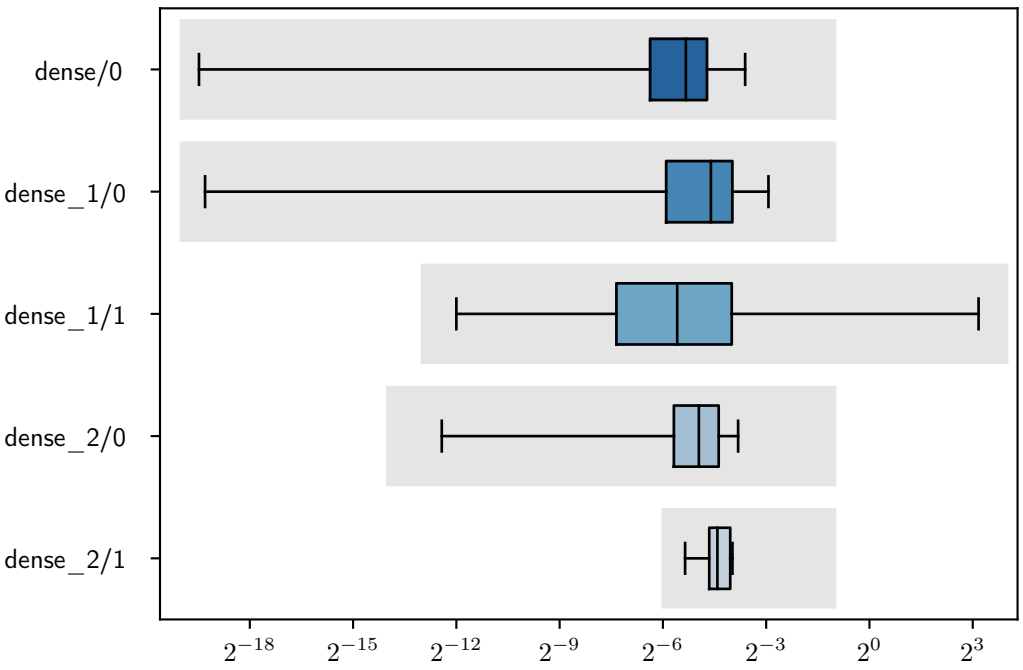
Program korzystający z funkcji dostarczanych przez tę bibliotekę analizuje plik

Wstawić źródło w przypisie



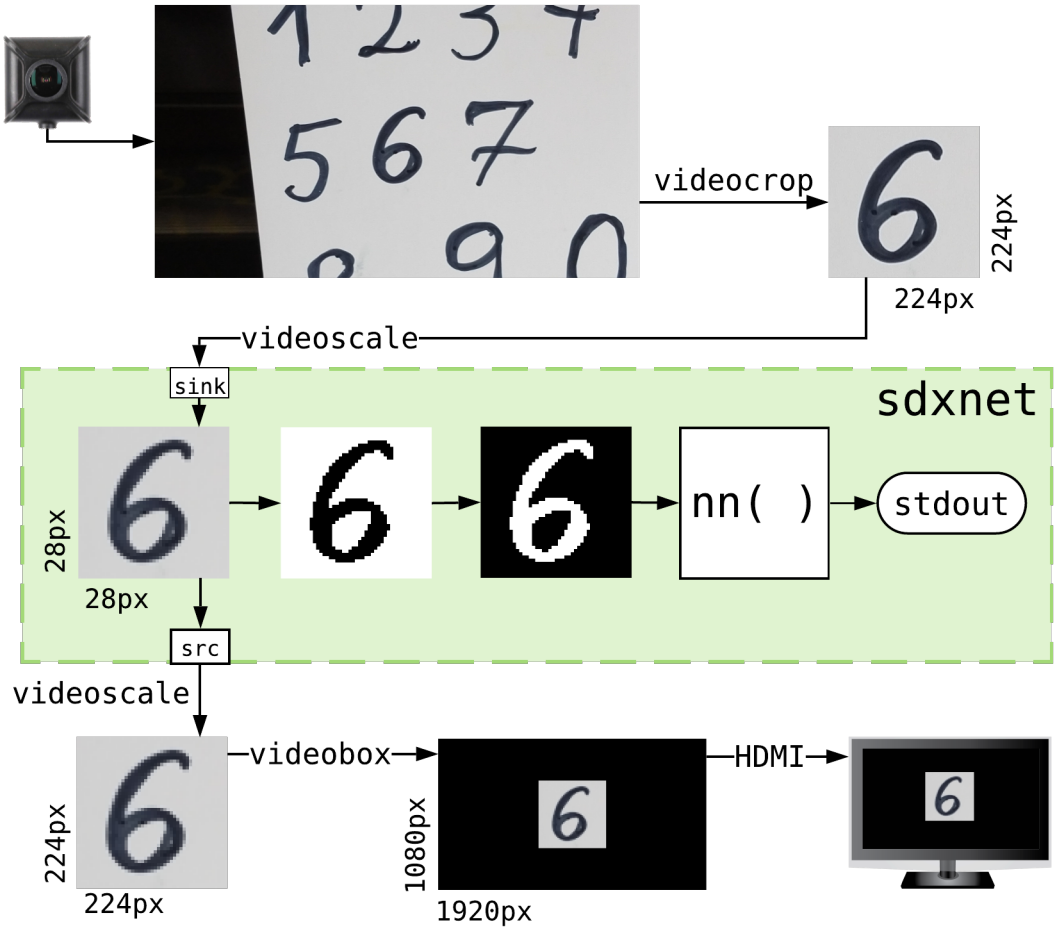
Rysunek 3: Schemat pracy z hls4ml

konfiguracyjny yml oraz model z pliku h5. Wynikiem działania programu jest wykres przedstawiający rozkład wartości wag każdej z warstw modelu otrzymanych w procesie trenowania. Szare pole w tle wykresu przedstawia zakres wartości, które obejmowane są przez precyzję określoną w pliku konfiguracyjnym. Dobrym punktem początkowym jest wybranie takiej liczby bitów dla każdej z warstw, która obejmuje wszystkie możliwe wagi. Dalsze ustalanie precyzji można wykonać w trakcie analizy wyników symulacji.



Rysunek 4: Rozkład wartości wag modelu

3.5 GStreamer

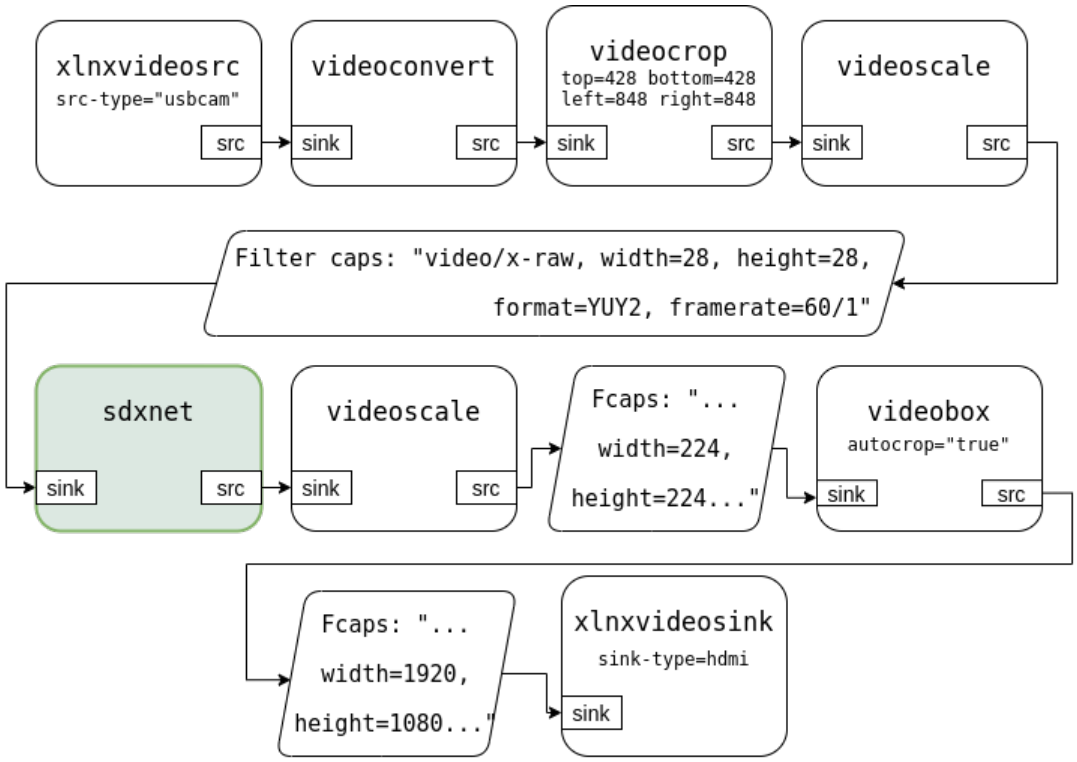


Rysunek 5: Schemat logiczny przetwarzania obrazu

Zsyntetyzowana sieć jest częścią projektu. Potrzebne jest również dostarczenie danych do sieci oraz przedstawienie wyniku. Do tego celu skorzystano z biblioteki GStreamer, dzięki której można tworzyć grafy z komponentów (pluginów, elementów) przetwarzających media, zarówno audio jak i video. Każdy z elementów grafu składa się z co najmniej jednego źródła (source), lub ujścia (sink), może mieć również wiele wejść i wyjść. W grafie pierwszy element nie może mieć wejść, natomiast konieczne jest aby posiadał co najmniej jedno wyjście. Poprawnie przygotowany graf nie powinien mieć komponentów oferujących źródło, które nie są z niczym połączone. Pluginy mają ujednolicony interfejs, dzięki czemu można w łatwy sposób włączyć do grafu własny element. Wtyczki charakteryzują się pewnymi własnościami, znanymi jako „caps". Określają one jakie media jest w stanie przetworzyć dana wtyczka (na przykład format pikseli, maksymalny rozmiar obrazu). Łączone ze sobą elementy dokonują negocjacji parametrów mediów, takich jak rozdzielczość obrazu, format pikseli, ilość klatek na sekundę oraz innych.

3.5.1 xlnxvideosrc i xlnxvideosink

Są to pluginy dostarczone przez firmę Xilinx wraz z platformą reVISION. Obydwa korzystają biblioteki Xilinx `video_lib` Pierwszy z nich ułatwia odczytywanie da-



Rysunek 6: Schemat grafu. Na zielono plugin z siecią neuronową

nych ze źródeł, dla których potrzebne byłyby dodatkowe działania. Są to między innymi kamera USB (użyta w projekcie), HDMI, MIPI CSI (sprzętowy interfejs do transmisji obrazów i wideo). Sam element zbudowany jest w oparciu o element v4l2src, dostępny w standardowej instalacji GStreamera. Xlnxvideosink również jest oparty o inny element — kmssink. Zapewnia odpowiednią konfigurację połączenia z wyświetlaczami podłączonymi przez HDMI oraz DisplayPort.

Przypis UG1221, s.32

3.5.2 videoconvert

Element mający za zadanie dostosować wszystkie parametry obrazu tak, aby móc połączyć ze sobą dwa niekompatybilne pod względem „caps" elementy. Ta niekompatybilność może być spowodowana na przykład tym, że dwie wtyczki potrzebują innego formatu pikseli i jednocześnie nie oferują możliwości konwersji z jednego formatu na inny.

3.5.3 videocrop

Wtyczka służąca do wykadrowania obrazu w zdefiniowanym obszarze. Wykorzystana została aby otrzymać obraz o tej samej długości i szerokości wynoszącej 224 (co jest ośmiokrotnością 28, czyli długością boku obrazów, którymi wytrenowana została sieć) wycięty ze środka wideo o rozmiarze 1920×1080.

3.5.4 videoscale

Skaluje obraz do wynegocjowanych pomiędzy sąsiadującymi elementami parametrów, przy czym pierwsza próba negocjacji to ta sama wielkość obrazu przy ujściu

jak i w źródle, aby skalowanie nie było potrzebne.

3.5.5 sdxnet

Sdxnet to wtyczka wykorzystująca sieć neuronową do rozpoznania cyfr znajdujących się na obrazie przez nią przechodzącym. Element ten został zaimplementowany na potrzeby tego projektu.

3.5.6 Filter caps

Element precyzujący parametry obrazu, które wymuszają dostosowanie się poprzedniego elementu — na przykład videoscale. Zapisuje się go w postaci ciągu znaków ujętych w cudzysłów.

3.5.7 videobox

Oferuje możliwość osadzenia obrazu w tym o innym rozmiarze rozmiarze wypełniając pozostałą przestrzeń ramką w wybranym kolorze. Własność autocrop oznacza automatyczne obliczenie wielkości ramek na podstawie parametrów określonych przez kolejny element tak, aby obraz przychodzący do videobox był wycentrowany a ramki były tej samej wielkości.

3.5.8 fpsdisplaysink

Wtyczka typu sink (mająca tylko ujście), która jako parametr pobiera inną wtyczkę tego typu, np. xlnxvideosink, zastępując w grafie tamtą. Jej użycie pozwala na sprawdzenie liczby klatek na sekundę wyświetlanego obrazu.

Todo: schemat fizyczny: hdmi -> arm (wtyczki) -> dane do sieci -> ...

3.6 Używanie sieci

3.6.1 Generowanie projektu

Architekturę sieci wraz z wagami zapisano do pliku h5. Stworzono plik konfiguracyjny hls4ml. Następnie na jego podstawie wygenerowano projekt z przekonwertowaną siecią. Wśród wygenerowanych plików znajduje się również kod służący do symulacji działania projektu. Przygotowane zostały pliki z danymi testującymi sieć — 10000 przetworzonych przykładów z bazy MNIST tak, aby cyfry były koloru czarnego, tło białego. Zakres wartości wynosi od 0 do 255.

3.6.2 Funkcja

Cała sieć jest przedstawiona jako jedna funkcja. Parametrami tej funkcji są dwie tablice: `input[]`, do której są zapisywane są dane do przetworzenia, oraz `output[]`, do której funkcja zapisuje obliczone predykcje.

```
1 void nn(
2     unsigned char input[784],
3     float         output[10]
4 );
```

Listing 1: Nagłówek funkcji

3.6.3 Interfejs

Aby móc korzystać z sieci w aplikacji uruchamianej na procesorze ARM zadeklarowano użycie interfejsu IO AXI-4 Lite.

3.6.4 Dostosowanie sieci

W celu poprawienia wyników działania sieci dokonano pewnych usprawnień. Sieć została wytrenowana oryginalnymi danymi, w których piksele tworzące cyfry mają wartości równe 255 lub tej wartości bliskie, a piksele białego są przedtawione jako 0. Ponadto rzeczywiste dane z kamery mogą być zaszumione, przedstawione obiekty zacienione, a same cyfry mogą nie być idealnie czarne.

Przy każdym wywołaniu funkcji sieci dokonywana jest transformacja danych poprzez kod pokazany na listingu 2. Wartość każdego piksela jest zamieniana na wartość 255 lub 0, zależnie od początkowej jego wartości — dla wartości mniejszych od 140 (kolor szary lub ciemniejszy) przypisany jest kolor biały, wartość 255. Dla pikseli jasnych (od 140 w górę) przypisywana wartość to 0, kolor czarny.

W ten sposób dokonuje się zarówno odpowiedniego przetworzenia danych uwzględniającego sposób wytrenowania modelu, jak również uwydatnienia cyfry oraz pozbycia się szumów obrazu i jasnych cieni.

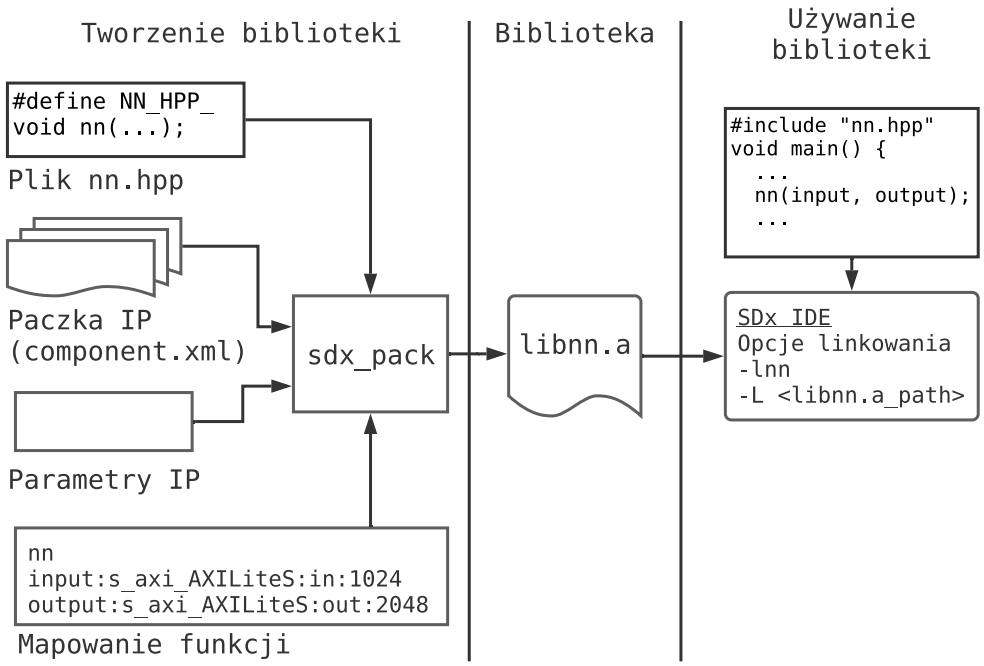
```
for(int i=0; i<784; i++) {
#pragma HLS pipeline II=1
        input1[i] = (input[i] < 140) ? 255 : 0;
}
```

Listing 2: Transformacja danych.

- `input[]` — tablica z danymi (parametr funkcji)
- `input1[]` — dane przetwarzane przez sieć

3.6.5 Tworzenie biblioteki

Zsyntetyzowany moduł sieci został wyeksportowany w środowisku Vivado HLS do paczki IP (Intelectual Property). Następnie poprzez narzędzie `sdx_pack` utworzono statyczną bibliotekę gotową do wykorzystania w innym projekcie w środowisku SDSoC (Software-Defined System on Chip, IDE do pisania aplikacji lub bibliotek uruchamianych na platformach Xilinx MPSoC).



Rysunek 7: Schemat tworzenia biblioteki

```
sdx_pack -header nn.hpp -lib libnn.a \
         -func nn -map input=s_axi_AXILiteS:in:1024 \
                  -map output=s_axi_AXILiteS:out:2048 \
         -func-end \
         -ip ip/component.xml \
         -control ap_ctrl_hs=s_axi_AXILiteS:0 \
         -primary-clk ap_clk=13.333 \
         -target-family zynquplus \
         -target-cpu cortex-a53 \
         -target-os linux \
```

Listing 3: Narzędzie sdx_pack

Wywołując narzędzie `sdx_pack` należy podać plik nagłówkowy funkcji, docelową nazwę biblioteki, mapowanie parametrów funkcji na porty modułu, ścieżkę do pliku `component.xml` wygenerowanego podczas eksportu do paczki IP, protokół kontroli modułu, odpowiedni zegar, a informacje dotyczące docelowej platformy: nazwę jej rodziny, procesora oraz systemu.

3.7 Część neuralnet

Czytanie obrazu, podział na część luma i chroma, wywołanie funkcji sieci, zapis z powrotem, synteza do biblioteki dzielonej „.so”

3.8 Część gstsdxnet

De facto plugin gstreamera, w którym są wywoływane funkcje z biblioteki dzielonej neuralnet.so,

3.9 Małe podsumowanie

4 Wyniki i dyskusja

4.1 Ewaulacja modelu

Wyniki z samego pythona z danymi testowymi z mnista

4.2 Symulacja

Tutaj wyniki z symulacji z danymi testowymi z mnista

4.3 Dane rzeczywiste

Wyniki z kamerki. Zdjęcia danych testowych, co wpływa na wynik, czy wszystko rozpoznaje itd,

5 Podsumowanie

W projekcie zostało zrobione to i to. Wyszło to tak i tak. Problem sprawiło tamto i owamto. Można to poprawić w ten sposób. Można część funkcjonalności z pipeline przenieść na fpga (w końcu przetwarzanie obrazu na fpga jest szybkie)