FAKULTA INFORMAČNÍCH TECHNOLOGIÍ ČVUT V PRAZE

Zadání bakalářské práce

Název:	Akcelerace neuronových sítí na cloudové kartě s FPGA
Student:	Jiří Tlamicha
Vedoucí:	Ing. Miroslav Skrbek, Ph.D.
Studijní program:	Informatika
Obor / specializace:	Počítačové inženýrství 2021
Katedra:	Katedra číslicového návrhu
Platnost zadání:	do konce letního semestru 2024/2025

Pokyny pro vypracování

Seznamte se s cloudovou kartou Alveo a prozkoumejte možnosti implementace AI aplikací. Zaměřte se na nástroje Vitis a Vitis AI a uvažujte akceleraci ve variantě s DPU (Deep Processing Unit) a s využitím vysoko-úrovňové syntézy do VHDL, kde zvážíte použití nástroje HLS4ML. Různé akcelerační přístupy ověřte nejprve na menších modelech neuronových sítích. Poté a po dohodě s vedoucím práce vyberte větší model neuronové sítě schopný klasifikovat objekty v obraze. Pro tento model navrhněte a implementujte prototypovou aplikaci, která bude zpracovávat jednotlivé snímky nebo videa z kamer humanoidních robotů, klasifikovat objekty a poskytovat robotům informace o objektech v obraze pro další aplikační využití. Implementované neuronové sítě zhodnoťte z různých pohledů např. potřebných zdrojů FPGA, výpočetního výkonu, spotřeby a real-time zpracování. Vše řádně zdokumentujte. Konkrétní rozsah práce a použité datové sady stanovte po dohodě s vedoucím práce.

Elektronicky schválil/a prof. Ing. Hana Kubátová, CSc. dne 20. ledna 2024 v Praze.

Bakalářská práce

AKCELERACE NEURONOVÝCH SÍTÍ NA CLOUDOVÉ KARTĚ S FPGA

Jiří Tlamicha

Fakulta informačních technologií
Katedra číslicového návrhu
Vedoucí: Ing. Miroslav Skrbek, Ph.D.
16. května 2024

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Chtěl bych upřímně poděkovat především vedoucímu této bakalářské práce Ing. Miroslavu Skrbkovi, Ph.D. za jeho ochotu vždy pomoci. Bez něj by tato práce nemohla vzniknout. Dále bych rád též poděkoval rodině a přátelům za podporu a trpělivost.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací. Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 2373 odst. 2 zákona č. 89/2012 Sb., občanský zákoník, ve znění pozdějších předpisů, tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, avšak pouze k nevýdělečným účelům. Toto oprávnění je časově, teritoriálně i množstevně neomezené.

V Praze dne 16. května 2024

Abstrakt

Tato práce se zabývá akcelerací neuronových sítí na cloudové FPGA kartě Alveo U55C. Zkoumány jsou zejména framework HLS4ML využívající vysoko-úrovňovou syntézu a framework Vitis AI využívající akceleraci pomocí DPU. Oba frameworky jsou nejprve otestovány na malých sítích pro datovou sadu Iris. Poté jsou provedeny experimenty se sadou různě velkých konvolučních neuronových sítí trénovaných na datové sadě CIFAR-10. Jsou nalezeny limity obou přístupů. Experimenty jsou vyhodnoceny z hlediska délky času zpracování a spotřeby energie. Poté je pomocí Vitis AI provedena akcelerace modelu YOLOv7 pro detekci objektů v obraze. Ten je poté použit v prototypové aplikaci pro zpracování obrázků z humanoidních robotů Nao. Model je také vyhodnocen z hlediska propustnosti a latence.

Klíčová slova Vitis AI, DPU, HLS4ML, FPGA, Alveo U55C, neuronová síť, CNN, YOLOv7

Abstract

This thesis researches the acceleration of neural networks on cloud FPGA card Alveo U55C. In particular, it explores the HLS4ML framework, which uses high-level synthesis, and the Vitis AI framework, which uses DPU acceleration. First both frameworks are tested on small networks trained for the Iris dataset. Then they are used for experiments with a set of convolutional neural networks of different sizes trained for the CIFAR-10 dataset. Limits of both frameworks are found. Experiments are evaluated in terms of processing time and power consumption. Then the Vitis AI is used to accelerate the YOLOv7 model to detect objects in images. This model is then used in a prototype application for processing of images from humanoid Nao robots. This model is also evaluated in terms of throughput and latency.

Keywords Vitis AI, DPU, HLS4ML, FPGA, Alveo U55C, neural network, CNN, YOLOv7

Seznam zkratek

CNN	Convolutional Neural Network
DPU	Deep-learning Processor Unit
DSP	Digital Signal Processor
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
HLS	High-Level Synthesis
HLS4ML	High-Level Synthesis for Machine Learning
IoU	Intersection over Union
IP	dle kontextu buď Intellectual Property, nebo Internet Protocol
LUT	Lookup Table
NNDCT	Neural Network Deep Compression Toolkit (interní název pro vai_q_pytorch)
PTQ	Post Training Quantization
QAT	Quantization Aware Training
SSD	Single-Stage Detector
TSD	Two-Stage Detector
VART	Vitis AI Runtime
venv	Python Virtual Environment
XIR	Xilinx Intermediate Representation
XRM	Xilinx FPGA Resource Manager
XRT	Xilinx Runtime

Úvod

Umělá inteligence a strojové učení v poslední době zažívají obrovský rozmach. Neuronové sítě se dnes využívají v nepřeberném množství aplikací, od analýzy a agregace senzorových dat v robotice, přes predikce finančních trhů až po velké generativní modely pro vytváření obrázků či chatboty typu ChatGPT. Ovšem pro provoz těchto neuronových sítí je často třeba extrémně vysoký výpočetní výkon. Existující modely se přitom stále zvětšují a s tím dále rostou i nároky na hardware.

Poptávka po akceleraci neuronových sítí je vysoká. Neuronové sítě jsou navíc ze své podstaty vysoce paralelní. Díky tomu lze použít různé typy akcelerátorů. Dnes nejčastější je použití GPU karet. Další možností je akcelerace pomocí FPGA. Ta dnes však není příliš obvyklá. Lze se s ní setkat spíše u menších sítí v systémech, kde je kladen důraz na nízké latence zpracování.

Tato práce naproti tomu zkoumá možnosti cloudové akcelerace neuronových sítí na kartách s FPGA. Důraz je kladen hlavně na akceleraci konvolučních sítí pro klasifikaci a detekci objektů v obraze. Pro experimenty je použita konkrétně karta Alveo U55C od společnosti AMD. Práce se nezabývá akcelerací učení neuronových sítí, pouze akcelerací dopředného chodu, tzv. inference.

V teoretické části jsou nejprve probrány oblasti strojového učení použité v této práci. Dále je stručně popsána karta Alveo U55C a její použití. Poté je rozebrána problematika akcelerace neuronových sítí na FPGA. Analyzovány jsou zejména frameworky HLS4ML (High-Level Synthesis for Machine Learning) a Vitis-AI. HLS4ML je open-source Python modul pro převod modelů neuronových sítí z frameworků typu PyTorch nebo Tensorflow do kódu v jazyce C vhodného pro vysoko-úrovňovou syntézu. Celý model včetně vah je pak možné syntetizovat do bitstreamu pro FPGA. Vitis-AI je oproti tomu sada nástrojů od firmy AMD určená pro akceleraci neuronových sítí pomocí tzv. DPU (Deep-learning Processor Unit), což jsou předem připravená procesorová jádra speciálně určená pro akceleraci neuronových sítí, která lze nahrát do FPGA.

V praktické části jsou poté oba postupy aplikovány nejprve na malou neuronovou síť pro datovou sadu Iris a poté na soubor konvolučních sítí různých velikostí pro klasifikaci obrázků z datové sady CIFAR-10. Nakonec je implementována prototypová aplikace pro detekci a klasifikaci objektů v obrázcích z kamer humanoidních robotů založená na modelu YOLOv7.

Cíle práce

Hlavním cílem práce je prozkoumat způsoby akcelerace neuronových sítí pomocí nástrojů Vitis-AI a HLS4ML. Cílem je najít hranice schopností obou frameworků a zdokumentovat potenciální problémy, které při jejich využití mohou nastat. Implementované modely potom budou vyhodnoceny z hlediska propustnosti, latence a spotřeby.

Dalším cílem práce je realizovat prototypovou serverovou aplikaci, která bude přijímat snímky z kamer humanoidních robotů, detekovat v nich objekty pomocí neuronové sítě akcelerované na FPGA kartách a posílat robotům zpět informace o objektech.

Cíle práce lze rozepsat do následujících bodů:

- zprovoznit akceleraci jednoduché neuronové sítě pomocí HLS4ML na kartě Alveo U55C,
- zprovoznit akceleraci jednoduché neuronové sítě pomocí Vitis AI na kartě Alveo U55C,
- vyzkoušet akceleraci konvolučních sítí různých velikostí a nalézt potenciální limity obou způsobů,
- změřit propustnost, latence a spotřeby implementovaných sítí,
- zprovoznit akceleraci sítě schopné detekce a klasifikace objektů v obrazu,
- napsat prototypový server pro zpracování obrázků touto sítí,
- napsat příslušnou prototypovou klientskou aplikaci pro humanoidní roboty Nao,
- změřit propustnost, latenci a spotřebu této neuronové sítě.

Kapitola 1
Teoretická část

1.1 Neuronové sítě pro zpracování obrazu

Výzkumu strojového učení už bylo věnováno velké úsilí a dnes tak existuje celá řada různých typů a modelů neuronových sítí. Celá oblast je navíc stále velmi dynamická a v současné době se rychle rozvíjí. Tato sekce se proto snaží stručně shrnout a popsat technologie a pojmy, které jsou použité v této práci.

1.1.1 Konvoluční neuronové sítě

Konvoluční neuronová síť, anglicky *Convolutional Neural Network* (CNN), je typ neuronové sítě vhodný zejména pro zpracování prostorových dat, nejčastěji obrázků. Lze je často najít v modelech pro detekci objektů v obraze, pro segmentaci obrazu nebo i generování textového popisu obrázku. CNN se typicky skládají ze třech typů vrstev, konvolučních vrstev, kompresních (*pooling*) vrstev a plně propojených vrstev. [1]

Konvoluční vrstva je základním stavebním blokem CNN. Provádí se v ní skalární násobení a suma prvků dvou matic. První je matice trénovatelných parametrů, tzv. *kernel*. Ten se posouvá přes matici obrázku a násobí se vždy s její odpovídající podmaticí. Výstupem je tedy opět matice s prvky odpovídajícími jednotlivým pozicím kernelu. Kernelů v jedné vrstvě je obvykle více, jeden pro každý vstupní kanál. Jejich soubor se pak nazývá *filter*. Jeho výstupem je matice vzniklá sečtením výstupních matic všech jeho kernelů. Filtrů poté může být ve vrstvě libovolné množství. Výstup z konvoluční vrstvy potom bude mít tolik kanálů, kolik má vrstva filtrů. [1, 2]

Pooling vrstva také používá podmatice vzniklé posunem okénka fixní velikosti přes vstupní matici. Ale nenásobí je s kernelem, nýbrž na nich pouze provede některou agregační operaci. Nejčastěji to je výběr maxima, takové vrstvě se potom říká *max pooling*.

V plně propojené vrstvě jsou, jak název napovídá, všechny neurony propojené se všemi prvky vstupu. [1]

Za konvoluční a plně propojené vrstvy se navíc přidávají nelineární aktivační funkce. Jejich hlavním účelem je přidat do sítě nelinearitu a umožnit tak modelu klasifikovat nelineární vzory. Používané aktivační funkce jsou například: sigmoid, softmax, tanh, ReLU, leakyReLU, parametricReLU, ReLU6, swish nebo hard-swish. [3]

Výše zmíněně vrstvy se poté v různých počtech a parametrech propojí za sebe a vznikne konvoluční neuronová síť. První vrstvy na začátku detekují jednodušší vzory. Ty další potom postupně na základě těchto informací detekují vzory komplexnější. [1]

Konkrétními příklady klasických konvolučních neuronových síti pro klasifikaci obrazu jsou třeba sítě LeNet-5 [4] nebo AlexNet [5].

1.1.2 Detekce objektů v obraze

Modely schopné detekce a klasifikace objektů v obraze lze rozdělit do tří kategorií: jednoúrovňové, anglicky single-stage detectors (SSD), dvouúrovňové, anglicky two-stage detectors (TSD) a detektory založené na transformerech (ale ty zde zkoumány nejsou). Dvouúrovňové detektory nejprve vyberou potenciální regiony s objekty a ve druhém kroku je provedena klasifikace a lokalizace v daných regionech. Jednoúrovňové detektory naproti tomu neprovádí výběr regionů, ale použijí předem definované body, okolo kterých se pokusí detekovat a lokalizovat objekty. Příklady TSD jsou Region-based CNN (R-CNN), Fast Region-based CNN (Fast R-CNN) nebo Faster Region-based CNN (Faster R-CNN). Příkladem SSD jsou modely z rodiny YOLO (You Only Look Once). TSD obecně dosahují vyšší přesnosti ale nižší rychlosti. SSD jsou oproti tomu rychlejší a vhodnější pro real-time využití, avšak jejich přesnost bývá o něco nižší. Nicméně poslední modely SSD, jako YOLOv7 [6], dokážou dosáhnout i vysoké přesnosti. [7]

1.1.2.1 YOLOv7

Pro prototypovou aplikaci byl zvolen právě model YOLOv7 [6]. Model se stejně jako další YOLO modely skládá ze 3 základních částí: backbone, neck a head. Na vstupu je do backbone předán obrázek fixní velikosti (640 × 480 px). V backbone jsou pomocí sady konvolučních vrstev rozpoznány základní vzory (*features*) různých velikostí. V části neck jsou pomocí tzv. *feature pyramid* zkombinovány vzory různých velikostí. A v části head nakonec dojde ke konečné predikci tříd a pozic objektů pomocí tzv. *bounding boxů*, což jsou obdélníky ohraničující detekované objekty.

Samotná síť má 3 sady výstupů, přičemž každý má vlastní část head a detekuje objekty jiných velikostí. Každá z těchto *hlav* má k sobě přiřazeny obvykle 3 konkrétní předem definované velikosti boxů (tzv. *anchor sizes*) a soustavu bodů v mřížce rozložených po obrázku (tzv. *anchor points*). Ty dohromady tvoří soustavu boxů (*anchor boxes*). Pro každý takový box má pak model jeden výstup určující odchylky detekovaného bounding boxu na souřadnicích x a y, odchylky výšky a šířky boxu, míru pravděpodobnosti přítomnosti objektu a pak míry pravděpodobnosti pro každou z detekovaných tříd objektů. Nutno říct, že odchylky souřadnic bounding boxů jsou relativní k velikosti mřížky s *anchor points* a jejich velikosti zase k *anchor sizes*.

V rámci post-processingu se spočítají skutečné pravděpodobnosti tříd objektů vynásobením s pravděpodobností přítomnosti objektu. Zkombinováním boxů a pravděpodobností tříd vzniknou výsledné bounding boxy. Z těch se odstraní ty, které mají pravděpodobnost menší než nějaký práh (tzv. *confidence threshold*). Nakonec se provede non-maximum suppresion, což znamená, že pokud mají dva boxy *IoU* (viz příští sekce) vyšší než nějaký limit (tzv. *IoU threshold*), tak se ten s menší pravděpodobností odstraní. [8, 9]

1.1.2.2 Hodnocení modelu

Aby bylo možné vyhodnotit, jak dobrý model je, jsou potřeba metriky pro hodnocení. Při detekci objektů se lze setkat s termíny *precision* (přesnost) a *recall*. Precision udává poměr true positive predikcí ku součtu true positive a false positive predikcí. Jinými slovy, kolik z predikovaných boxů bylo správných. Recall potom udává poměr true positive predikcí ku součtu true positive a false negative predikcí. Aneb kolik ze skutečně přítomných objektů v obrázku bylo detekováno.

Ještě zbývá, jak určit, že konkrétní predikce objektu je správná. K tomu slouží tzv. IoU, *Intersection over Union* (česky průnik nad sjednocením). Jak název napovídá, IoU dvou boxů je poměr jejich překrývajících se oblastí ku sjednocení jejich oblastí. IoU tedy udává na škále 0 až 1, jak moc si jsou dva boxy podobné. Aby bylo možné spočítat precision a recall, je nutné určit si limit IoU (tzv. *IoU threshold*), nad kterým považuji predikci za správnou.

Pro celkové vyhodnocení modelu se pak počítají hodnoty Average Precision (AP) a Mean Average Precision (mAP). AP se počítá jako obsah pod precision-recall křivkou (vzniklou vynesením závislosti precision na recall do grafu). mAP je potom průměr AP přes všechny třídy

a různé hodnoty confidence threshold. mAP se udává zvlášť pro různé hodnoty IoU threshold, například mAP@0.50, mAP@0.95 nebo mAP@0.5:0.95 (interval). [10, 11]

1.1.3 Datové sady

Pro využití neuronových sítí je třeba mít datové sady, na kterých se budou sítě učit a testovat. V této práci jsou použity 3 datové sady: Iris, CIFAR-10 a MS COCO.

Datová sada Iris byla poprvé použita roku 1936 R. A. Fisherem [12]. Data byla nasbírána botanikem Edgarem Andersonem. Jedná se o různé druhy kosatce a naměřené rozměry jejich okvětních lístků a kalichů. Sada je dodnes využívána pro jednoduché experimenty. [13]

CIFAR-10 je již obrazová datová sada. Byla vytvořena v roce 2009. Je pojmenována podle Canadian Institute for Advanced Research. Obsahuje 60000 malých barevných obrázků o rozměrech 32×32 rozdělených do 10 kategorií (po 6000 obrázcích). Kategorie jsou letadlo, automobil, pták, kočka, jelen, pes, žába, kůň, loď a kamion. [14, 15]

Datová sada MS COCO je velká datová sada, kterou lze použít pro detekci objektů, segmentaci nebo generování popisů obrázků. Zkratka COCO znamená Common Objects in Context. Sada obsahuje 330 tisíc obrázků. 200 tisíc z nich je anotovaných. Objekty jsou v ní rozděleny do 80 různých kategorií. Projekt YOLOv7 využívá MS COCO jako výchozí datovou sadu. [16, 17, 6]

1.1.4 Nástroje pro implementaci neuronových sítí

Pro učení i nasazení neuronových sítí lze dnes použít hotové frameworky pro hluboké učení. Ty umožňují vytvářet neuronové sítě jednoduše z již připravených modulů. Příkladem může být knihovní třída *Conv2D* z nástroje Keras, která na základě předaných parametrů implementuje konvoluční vrstvu neuronové sítě. To vývoj velice zjednodušuje a použití těchto nástrojů se proto stalo naprostým standardem.

Jako příklady těchto frameworků lze uvést třeba TensorFlow, Keras, PyTorch, Caffe, Theano nebo Deeplearning4j. V poslední době se největší popularitě těší zejména tři prvně jmenované. TensorFlow a PyTorch jsou oba plnohodnotné frameworky a umožňují mimo jiné akceleraci pomocí GPU. Keras slouží pouze jako alternativní vnější rozhraní, dnes zejména pro TensorFlow. Primárním programovacím jazykem je pro všechny tři Python. [18]

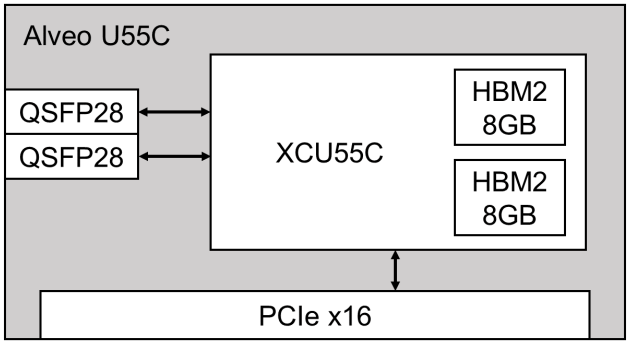
1.1.5 Optimalizace neuronových sítí pro akceleraci

1.1.5.1 Kvantizace

Obvykle jsou modely neuronových sítí vyvíjeny nad datovými typy v plovoucí řádové čárce, běžně ve velikosti 32 bitů. To ovšem stojí relativně velké množství zdrojů. Operace nad celými čísly a čísly s pevnou řádovou čárkou jsou oproti tomu rychlejší a nevyžadují tak složitý hardware.

Proto je dnes na akcelerátorech (nejen FPGA) často možné neuronové sítě provozovat i v reprezentaci s pevnou řádovou čárkou. Konverze modelu z plovoucí řádové čárky do té fixní se pak nazývá jako tzv. *kvantizace* (anglicky quantization). Ta často lehce snižuje přesnost modelu, ale zároveň zvyšuje výkon a umožňuje tak nasazení modelů větších.

Proces kvantizace lze provést dvěma způsoby. Prvním je tzv. Post Training Quantization (PTQ). To znamená, že model se nejprve natrénuje v plovoucí řádové čárce a kvantizace se provede až poté a bez dalšího trénování. Druhým způsobem je tzv. Quantization Aware Training (QAT). Při něm se kvantizace provádí během trénování. Model se může trénovat od začátku nebo může jít jen o přeučení modelu už naučeného v plovoucí řádové čárce. PTQ má výhodu v tom, že ho lze jednoduše a rychle použít na existující natrénované modely. QAT naproti tomu obvykle dosahuje lepších výsledků. Cenou za to je vyšší náročnost tohoto procesu. [19]



Obrázek 1.1 Diagram karty Alveo U55C. Vytvořeno dle informací z [20].

1.1.5.2 Prořezávání

Další možnou optimalizací je *prořezávání* neuronové sítě (anglicky pruning). To znamená, že se v síti vyberou parametry, které mají zanedbatelný vliv na výsledek a odstraní se. Tím lze výslednou síť výrazně zmenšit. Prořezávání, stejně jako kvantizace, často lehce snižuje výslednou přesnost, ovšem vykoupeno to je opět vyšší propustností. [19]

1.2 FPGA akcelerátor

Pro tuto práci byla použita deska Alveo U55C od firmy AMD. Tato sekce shrnuje specifikaci této karty a popisuje softwarové nástroje nutné pro její použití.

1.2.1 Hardware - Alveo U55C

Alveo U55C je serverová FPGA karta s rozhraním PCIe. Její diagram je zobrazen na obrázku 1.1. Je vybavena 16 GB HBM2 pamětí s propustností 460 GB/s. Dále poskytuje dva síťové porty QSFP28 Ethernet o rychlosti 100 Gb/s. Maximální odběr je 150 W. Karta je chlazená pasivně, předpokládá se umístění v serverové skříni s aktivním prouděním vzduchu.

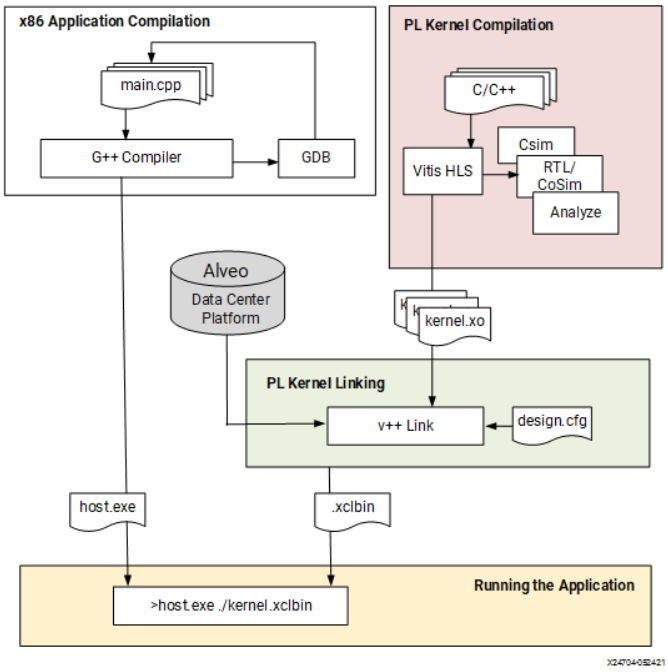
Srdcem karty je FPGA XCU55C postavené na architektuře UltraScale+. To obsahuje 1304000 LUT (Lookup Table) jednotek, 2607000 registrů a 9024 DSP (Digital Signal Processor) jednotek. Čip poskytuje 270 Mb paměti v UltraRAM blocích (po 288 kb), 70,9 Mb v block RAM (po 36 kb) a až 36,7 Mb v distribuované RAM. Uvnitř XCU55C jsou zároveň umístěné i zmíněné HBM2 paměti o kapacitě 2×8 GB. [20]

1.2.2 Software - Vitis

Většina nástrojů od AMD sloužících pro akceleraci (nejen) pomocí karet Alveo je dnes sjednocená pod názvem Vitis. Vitis projekt má zpravidla dvě části. Hostitelskou aplikaci běžící na CPU a pak tzv. PL kernely běžící v FPGA na Alveo kartách.

1.2.2.1 PL kernely

PL (Programmable Logic) kernely mají podobu souborů s příponou xo. Lze je vytvořit přímo na RTL úrovni (tedy ve VHDL nebo Verilogu). Nejprve se z Vivado projektu exportuje Vivado IP, které musí dodržet standardizované rozhraní pro kernely. To se potom zabalí pomocí příkazu *package_xo*, což je Tcl příkaz z nástroje Vivado. Tím vznikne xo soubor kernelu. Alternativní



Obrázek 1.2 Struktura Vitis HLS projektu určeného pro Alveo kartu. Převzato z [21].

možností je vysoko-úrovňová syntéza, anglicky High-Level Synthesis (HLS) ve Vitis IDE a její build pomocí Vitis Kernel Flow, který rovnou vytvoří xo soubor. To potom znamená, že kernel je napsaný v C++.

Oba přístupy je možné i kombinovat, z HLS je možné vytvořit Vivado IP (pomocí Vivado IP Flow namísto Vitis Kernel Flow). To se potom vloží do Vivado projektu. Další části se doprogramují na úrovni RTL, výsledek se vyexportuje a poté zabalí pomocí příkazu *package_xo*. Tento postup používá například Vivado Accelerator backend z HLS4ML (viz sekce 1.3.1.4).

Nicméně xo soubory jsou stále jen IP (Intellectual Property) se standardizovaným vnějším rozhraním. Tyto soubory je dále nutné syntetizovat pomocí nástroje v++. V tomto kroku je také samozřejmě nutné vybrat konkrétní cílovou platformu. Pokud všechno proběhne v pořádku, tak v++ provede syntézu, optimalizaci, placement, routing a vygenerování bitstreamu. Tento proces může zabrat od jednotek hodin až po pár dní. Výsledkem je soubor s příponou xclbin, tzv. device binary file. Ten obsahuje výsledný bitstream jednoho nebo i více kernelů a lze ho nahrát do FPGA.

PL kernely běží v tzv. dynamických regionech FPGA. To kromě dynamických regionů obsahuje ještě region statický, který je naprogramován vždy hned po startu karty a pak už nemůže být modifikován. Ten zajišťuje základní funkcionalitu jako je např. správa kernelů nebo komunikace přes PCIe rozhraní.

Kernely také mohou komunikovat s hardwarovými AI akcelerátory (tzv. AI Engines), pokud je dané zařízení obsahuje. Karty Alveo U55C ale tyto AI akcelerátory neobsahují. [21, 22]

1.2.2.2 Hostitelský software

Hostitelská aplikace je běžná softwarová aplikace napsaná v C++ běžící na CPU. Pro komunikaci s Alveo kartami se používá Xilinx Runtime (XRT) API. To pomocí ovladačů umožňuje aplikaci nahrát do FPGA příslušný device binary file (xclbin) a poté s ním komunikovat pomocí paměťově mapovaných registrů a bufferů. [21]

Jako nadstavbu XRT lze ještě použít Xilinx FPGA Resource Manager (XRM). Ten umožňuje

snadněji spravovat větší množství Alveo karet a rozdělovat dostupné zdroje aplikacím. Pro běžnou akceleraci není nutné XRM použít, ale Vitis AI (probíraný v dalších sekcích) ho vyžaduje [23]. [24]

Xilinx je jméno původní společnosti zabývající se vývojem FPGA. Dnes je ale tato firma součástí AMD.

1.3 Nástroje pro akceleraci neuronových sítí na FPGA

Nástroje pro implementaci neuronových sítí na FPGA lze na základě jejich architektury rozdělit do dvou kategorií: proudové architektury (streaming architectures) a samostatné výpočetní jednotky (single computation engines).

Proudové architektury jsou specializované pro každý model neuronové sítě. Pro jednotlivé vrstvy modelu jsou typicky vygenerovány jednotlivé speciální bloky hardwaru. Bloky se pak spojí za sebe a vytvoří pipeline. Vstupní data tak mohou být zpracovávána proudově ve všech blocích najednou. Proudové architektury umožňují potenciálně větší optimalizaci, nicméně vyžadují syntézu nového bitstreamu pro každý model neuronové sítě.

Architektury postavené na specializovaných výpočetních jednotkách naopak dávají přednost vysoké flexibilitě. Základem je fixní výpočetní jádro, které sekvenčně zpracovává jednotlivé vrstvy neuronové sítě. Operace jsou vykonávány na základě dodaných instrukcí odpovídajících dané neuronové síti. Stejné jádro tak může být použito pro více různých neuronových sítí. Cenou za to je potenciální nižší efektivita podobně jako u běžného procesoru. Podle konkrétního typu sítě například nemusí být využíván všechen implementovaný hardware a podobně. [25]

Příkladem proudových architektur jsou projekty fpgaConvNet [26, 27], DeepBurning [28], HADDOC2 [29], AutoCodeGen [30], HPIPE [31], FINN [32, 33, 34] nebo HLS4ML [35, 36, 37, 38]. Většina práce na tomto poli pochází čistě z akademické sféry. fpgaConvNet je práce Intelligent Digital Systems Lab, Imperial College London. Tento projekt je dnes stále rozvíjen, nicméně podle aktivity ve veřejném repozitáři se zatím nezdá, že by se těšil větší oblibě. Nástroj FINN je dnes udržován (jako open-source) společností AMD. Framework HLS4ML vznikl původně v rámci výzkumu v CERN a dnes je stále aktivně rozvíjen lidmi sdruženými v Fast Machine Learning Lab. Ze třech posledně jmenovaných je HLS4ML nejvíce univerzální a nejrozšířenější (dle statistik jeho veřejného repozitáře). Věnuje se mu i tato práce. Většina dalších jmenovaných projektů už není dnes aktivně rozvíjena.

Z architektur využívajících specializované výpočetní jádro lze jmenovat ALAMO [39], Angel-Eye [40], DnnWeaver [41], Caffeine [42], FP-DNN [43], Snowflake [44] nebo framework Vitis AI [23]. Až na Vitis AI jsou to opět většinou akademické projekty a v jejich základní podobě dnes nejsou prakticky využívány. Vitis AI je hotová sada nástrojů společnosti AMD určená pro akceleraci inference neuronových sítí na jejich zařízeních a je dále zkoumána v této práci.

1.3.1 HLS4ML

HLS4ML je Python balíček pro akceleraci neuronových sítí v FPGA. Název je zkratkou z High Level Synthesis for Machine Learning. Balíček umožňuje načíst model neuronové sítě z frameworku pro strojové učení (např. PyTorch) a zkompilovat ho do C++ kódu vhodného pro vysoko-úrovňovou syntézu.

Nástroj vznikl díky vědecké komunitě okolo urychlovače částic v CERN. Původní motivací bylo rozdělení naměřených dat ze srážek částic na ty, která se mají dále zpracovat, a na ty, která už zpracovávat netřeba. Strojové učení bylo na tento problém vhodné, avšak aby ho bylo možné použít živě za běhu, bylo třeba dosáhnout velmi nízkých latencí zpracování. Použití FPGA toto umožnilo. Původní článek o HLS4ML z roku 2018, v rámci kterého byl tento nástroj vytvořen, lze nalézt zde [35].

Dnes je projekt ve verzi 0.8.1. Je rozvíjen lidmi z Fast Machine Learning Lab. Celý je plně open-source a lze ho najít v repozitáři zde [37]. Jeho cílem je především dosáhnout co nejnižších latencí při zpracování neuronových sítí. [38]

1.3.1.1 Kvantizace v HLS4ML

HLS4ML používá kvantizovanou reprezentaci modelu. Kvantizace je nativně provedena manuálně, je nutné ručně zvolit datový typ. Lze přesně nastavit kolik bitů bude mít celočíselná část a kolik zlomková část. Datový typ lze nastavit pro celý model jednotně nebo pro každou vrstvu zvlášť. V rámci jednotlivých vrstev je poté možné odlišit i datový typ pro vstupy, váhy a bias. Alternativně lze použít QAT pomocí knihovny QKeras. HLS4ML má totiž implementované rozhraní pro načtení modelů z QKeras. [35, 36]

Co se týče prořezávání, tak to může být provedeno předem pomocí externího nástroje. Nicméně HLS4ML nabízí i vlastní *Hardware-aware Optimization API*. To umožňuje například cílit prořezávání na co nejnižší využití BRAM a DSP jednotek v FPGA. [38]

1.3.1.2 Další konfigurace modelu v HLS4ML

Jak už bylo zmíněno v předchozí kapitole, HLS4ML umožňuje nastavit datový typ použitý pro výpočty. Kromě toho lze ale nastavit i řadu dalších parametrů a vyrobit si tak akcelerátor vhodný pro potřeby konkrétní neuronové sítě a konkrétního typu FPGA. Nejdůležitějšími nastaveními jsou:

- precision,
- reuse factor,
- strategy,
- io type.

Pomocí *precision* se nastavuje přesnost použitého datového typu, konkrétně celková velikost a potom počet bitů určených pro celočíselnou část.

Reuse factor říká, kolikrát je násobička použita pro výpočet v dané vrstvě. Reuse factor roven jedné znamená, že výpočet je plně paralelní. Naopak reuse factor roven počtu násobení na dané vrstvě vede k plně sériovému výpočtu. Nižší reuse factor tedy znamená rychlejší výpočet, ale za cenu více použitých hardwarových prostředků FPGA, zejména DSP jednotek použitých pro násobení. [35]

U *strategy* lze volit mezi *Latency* a *Resource*. Obě používají lehce odlišnou HLS implementaci. První je vhodná zejména pro malé sítě, kde není třeba šetřit použitými prostředky. Druhá je zase vhodnější pro použití ve větších sítích a používá se obvykle s vysokým *reuse faktorem*.

Potom lze nastavit způsob předávání dat mezi vrstvami pomocí parametru *io_type*. Zde jsou na výběr možnosti *io_parallel* a *io_stream*. První způsob předává data plně paralelně. U větších sítí takové řešení ale nemusí být syntetizovatelné. Druhý způsob používá k předání dat mezi dvěma vrstvami FIFO (first-in first-out) buffer.

Pokud je součástí daného backendu i logická syntéza a vygenerování bitstreamu, tak lze ovlivnit i s tím související parametry. Například frekvenci na které poběží výsledný bitstream v FPGA. [38]

1.3.1.3 Podporované vstupní formáty

Pro načtení existujícího modelu má HLS4ML moduly pojmenované jako konvertory (anglicky converters). Někdy jsou též nazvány jako frontend. Plně implementovány jsou konvertory pro Keras a QKeras. Konvertor pro PyTorch existuje v limitované podobě. Ve vývoji jsou potom ještě konvertory pro ONNX a QONNX.

1.3.1.4 Podporované výstupní formáty

Výstupem z HLS4ML je obvykle projekt obsahující vygenerovaný kód pro HLS. Výstup se ale samozřejmě liší pro různé platformy. V HLS4ML se jednotlivé moduly pro generování HLS kódu nazývají backendy. Implementovány jsou backendy pro Intel HLS (Quartus) pro FPGA od Intelu a pro Vivado HLS a Vitis HLS pro FPGA od AMD. Podporované verze nástrojů jsou:

- Intel HLS - verze od 20.1 do 21.4,
- Vivado HLS - verze od 2018.2 do 2020.1,
- Vitis HLS - experimentální, verze od 2020.2 do 2022.2.

Vivado backend ale ještě obsahuje rozšíření, tzv. Vivado Accelerator backend. Ten staví na výstupním projektu z normálního Vivado backendu. Automaticky v něm spustí C-syntézu a vygeneruje příslušný kód na úrovni RTL. Ten potom obalí připravenými moduly napsanými ve Verilogu a vytvoří vnější AXI rozhraní kompatibilní s Python knihovnou PYNQ. Celý Vivado projekt se poté automaticky sesyntetizuje a výstupem je bitstream pro FPGA (v podobě xclbin souboru). Výsledná aplikace pak může komunikovat s akcelerátorem prostřednictvím volání knihovny PYNQ, pro což HLS4ML poskytuje i hotový skript. HLS4ML oficiálně podporuje jen karty: pynq-z2, zcu102, alveo-u50, alveo-u200, alveo-u250 a alveo-u280. Nicméně podporu lze snadno rozšířit na všechny karty podporované knihovnou PYNQ. [38]

PYNQ na kartách Alveo na pozadí komunikuje s akcelerátorem skrze XRT API. [45]

1.3.2 Vitis AI

Vitis AI je vývojové prostředí pro akceleraci inference neuronových sítí na FPGA platformách od firmy AMD. Jeho základem jsou tzv. Deep-learning Processor Units (DPU). Jsou to IP jádra vytvořená společností AMD, sloužící pro akceleraci neuronových sítí. K nim patří nástroje pro optimalizaci, kvantizaci a kompilaci neuronových sítí. Ty umožňují vzít naučený model z daného populárního frameworku a převést ho do instrukcí vhodných pro konkrétní DPU. Při samotném procesu akcelerace pak přichází na řadu Vitis AI Runtime. Ten se stará o nahrání příslušných DPU jednotek do karet Alveo a jejich použití pomocí zkompilovaných modelů. Nakonec je součástí frameworku i Model Zoo a Vitis AI Library. Ty obsahují příklady zkompilovaných modelů a kódů pro jejich spuštění. [23, 46]

Vitis AI vznikl okolo roku 2019 přejmenováním z Xilinx AI SDK. [47] Ten vznikl nedlouho předtím. Dnes je aktuální verze Vitis AI 3.5.

1.3.2.1 DPU

Deep-learning Processor Units (DPU) jsou programovatelné jednotky pro akceleraci neuronových sítí. AMD poskytuje řadu různých DPU, jak pro vestavěná zařízení jako Zynq nebo Versal, tak pro cloudové karty Alveo.

Pro cloudovou kartu Alveo U55C, použitou v této práci, je doporučeno DPUCAHX8H. To je optimalizované pro konvoluční neuronové sítě a zejména pro vysokou propustnost.

V tomto případě je DPU implementované plně v dynamickém regionu FPGA čipu. V situaci, kdy cílová platforma obsahuje hardwarové AI Engines jako například karta Alveo V70, bývá potom použita varianta DPU, která je v programovatelné logice FPGA implementována jen částečně. Část činnosti v takovém případě vykonávají právě AI Engines. [23, 46]

DPU jsou obvykle poskytovány ve formě hotového *device binary file* (xclbin). Ale ve Vitis AI repozitáři ve verzi 2.5 [48] ve složce *reference_design* lze najít i odkazy pro stažení původních kernelů ve formě xo souborů. Z nich si lze potom syntetizovat vlastní xclbin soubory.

1.3.2.2 Vitis AI Optimizer

Vitis AI Optimizer poskytuje nástroje pro prořezávání (pruning) neuronových sítí. V současné verzi 3.5 jsou podporované frameworky Tensorflow a PyTorch a příslušné nástroje jsou pojmenovány *vai_p_tensorflow*, *vai_p_tensorflow2* a *vai_p_pytorch*. Ve verzi 2.5 byly podporované i Caffe a Darknet. [23]

1.3.2.3 Vitis AI Quantizer

DPU z Vitis AI používají pro výpočty datový typ INT8 (8-bitový typ s fixní řádovou čárkou a znaménkem). Modely neuronových sítí je proto nutné před akcelerací kvantizovat.

Vitis AI podporuje jak post-training quantization (PTQ), tak quantization-aware training (QAT). V případě PTQ se provede jen analýza na malé sadě vstupních dat. Ta nemusí být ani anotovaná. Na základě toho se provede kvantizace. Nicméně to může pro některé modely vést k velké ztrátě přesnosti. QAT oproti tomu vyžaduje mít původní datovou sadu použitou pro trénování. Pomocí ní se provede přetrénování modelu do kvantizované podoby. Ztráta přesnosti je potom nižší, ale proces trvá oproti PTQ déle.

Ve verzi 3.5 jsou opět podporované frameworky TensorFlow a PyTorch a příslušné nástroje jsou tentokrát pojmenované *vai_q_tensorflow*, *vai_q_tensorflow2* a *vai_q_pytorch*. Pro *vai_q_pytorch* je v dokumentaci a příkladech také občas použito označení NNDCT (Neural Network Deep Compression Toolkit). [23]

DPU a potažmo Vitis AI Quantizer a Vitis AI Compiler také nepodporují všechny možné modely vytvořené v TensorFlow nebo PyTorch. Naopak podporované funkcionality jsou relativně dost omezené. Například DPUCAHX8H, použité v této práci, podporuje jako aktivační funkce pouze: ReLU, ReLU6 a Leaky ReLU [49]. Pokud model vyžaduje jinou aktivační funkci, je třeba ji implementovat jiným způsobem. Například výpočtem v CPU.

1.3.2.4 Vitis AI Compiler

Nakonec je třeba kvantizovaný model zkompilovat do instrukcí pro dané DPU. Pro Tensorflow se o toto starají *vai_c_tensorflow*, *vai_c_tensorflow2* a pro PyTorch *vai_c_xir*.

Výstupem je soubor s příponou xmodel. Ten obsahuje serializovaný model kódovaný v tzv. Xilinx Intermediate Representation (XIR), což je grafová reprezentace výpočtů pro neuronovou síť. Části grafu, které lze spustit v DPU, potom obsahují příslušné instrukce pro DPU. [23]

1.3.2.5 Vitis AI Runtime

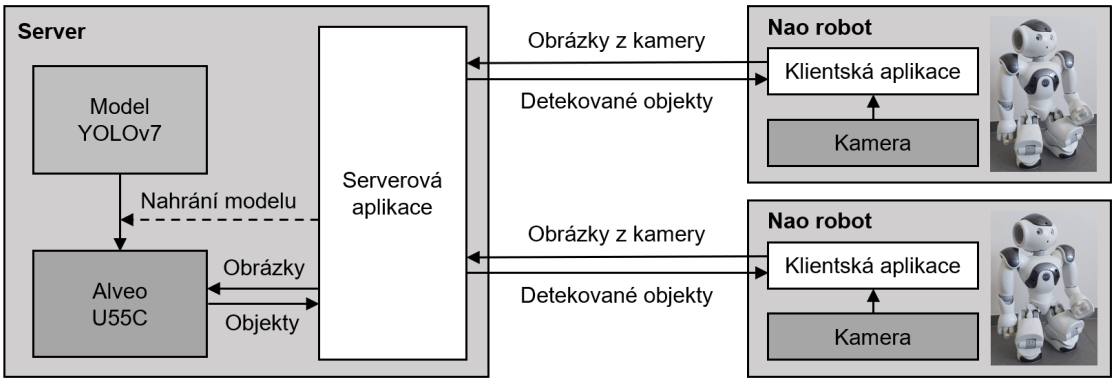
Vitis AI Runtime (VART) je vysoko-úrovňové API, které umožňuje spouštění zkompilovaných modelů v DPU na FPGA kartách. Jsou dostupné implementace v jazycích C++ a Python. VART poskytuje rozhraní pro asynchronní spouštění úloh na akcelerátorech. Je podporováno více-vláknové i více-procesové zpracování. Na pozadí je pro ovládání karet použito XRM. [23]

1.3.2.6 Vitis AI Docker a Vitis AI repozitář

Většina výše zmíněných nástrojů má zveřejněné zdrojové kódy v rámci veřejného Vitis AI Github repozitáře [50]. Z nich je lze samostatně nainstalovat. Doporučené je však použití Vitis AI Docker kontejneru. Ten obsahuje všechny zmíněné nástroje už zkompilované a nainstalované. Lze ho jednoduše stáhnout skrze Docker Hub. [23]

Kontejner se potom používá společně se zmíněným repozitářem. Ten totiž obsahuje skripty pro spuštění a nastavení kontejneru. Kromě toho také obsahuje třeba ukázkové kódy z Vitis AI Library nebo skripty pro stahování hotových zkompilovaných modelů z Model Zoo.

Kapitola 2
Návrh



Obrázek 2.1 Diagram prototypové aplikace pro zpracování obrazu akcelerované na Alveo U55C

Praktická část této práce se zabývá testováním akcelerace neuronových sítí na serverové kartě Alveo U55C a to zejména s důrazem na konvoluční neuronové sítě pro zpracování obrazu.

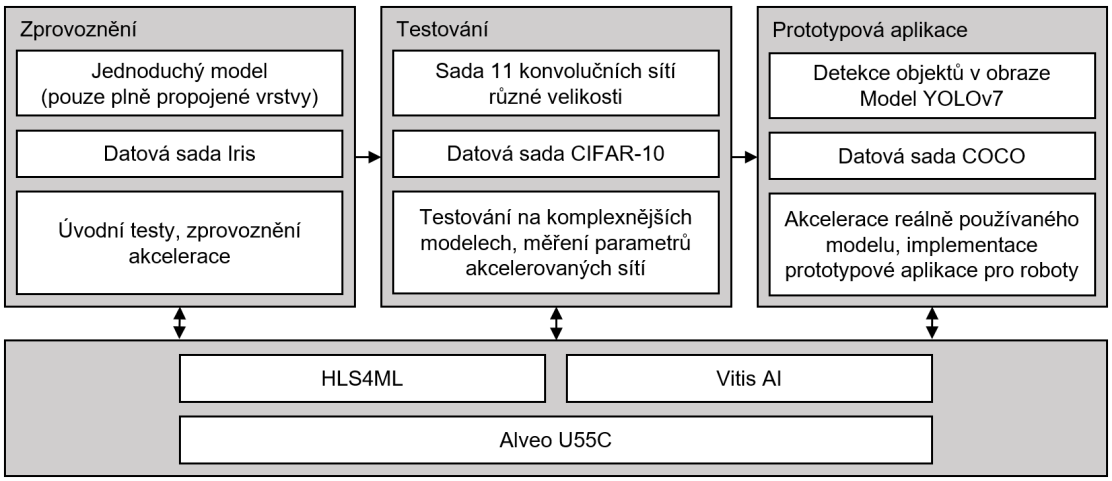
Nejprve jsou vyzkoušeny testovací modely neuronových sítí vytvořené v rámci této práce. Poté je demonstrováno praktické použití akcelerace vytvořením prototypové aplikace. Ta používá akceleraci existujícího modelu pro detekci objektů v obrázcích z kamer humanoidních robotů Nao. Diagram aplikace lze vidět na obrázku 2.1.

Praktická část práce je rozdělena konkrétně do třech větších celků, které jsou zobrazeny na diagramu 2.2.

Prvním je zprovoznění nástrojů HLS4ML a Vitis AI na maličké neuronové síti pro datovou sadu Iris. Cílem této části je otestovat oba frameworky na co nejjednodušším modelu, pro který například syntéza v HLS4ML nebude trvat příliš dlouho, a vyřešit tak potenciální problémy s v HLS4ML oficiálně nepodporovanou použitou kartou a odlišnými verzemi nástrojů od AMD.

V druhé fázi je provedeno další testování obou frameworků na jedenácti modelech vlastních konvolučních neuronových sítí definovaných pomocí rozhraní Keras v TensorFlow 2 a trénovaných pro datovou sadu CIFAR-10. Zde je cílem nalézt omezení obou frameworků a změřit parametry výsledných akcelerovaných modelů.

Nakonec je provedena akcelerace modelu YOLOv7 pro detekci a klasifikaci objektů v obraze. Pro tento model je poté vytvořen prototyp serverové aplikace a příslušný klient pro humanoidní roboty Nao.

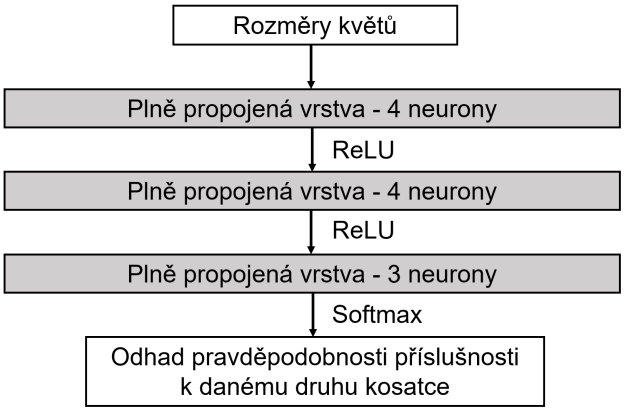


Obrázek 2.2 Diagram praktické části této práce

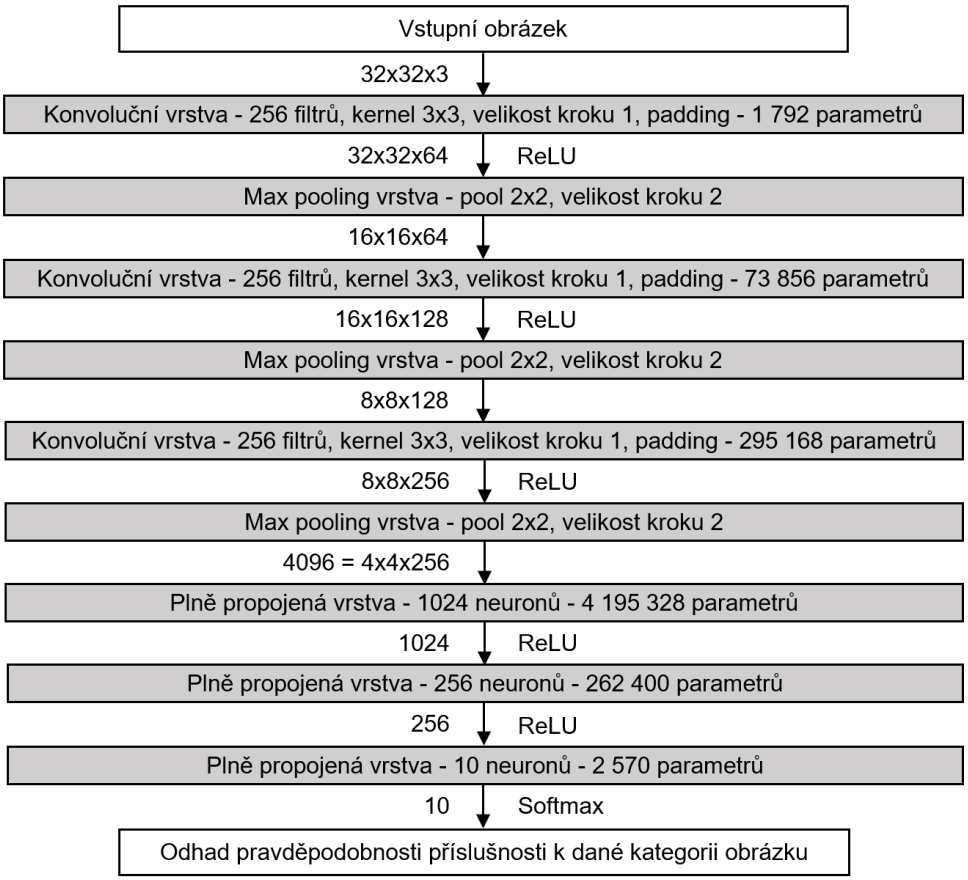
2.1 Malý model pro datovou sadu Iris

V první části je použit upravený ukázkový model neuronové sítě z předmětu NI-ESW napsaný pomocí Keras. Upravený model má tři plně propojené vrstvy o čtyřech, čtyřech a třech neuronech. První dvě vrstvy používají aktivační funkci ReLU, aby bylo možné model implementovat pomocí Vitis AI na DPUCAHX8H. Poslední vrstva využívá aktivační funkci softmax, aby výstupem byly pravděpodobnosti příslušnosti ke každému ze tří druhů kosatce. Softmax musí být v rámci akcelerace pomocí Vitis AI implementována na CPU, neboť DPUCAHX8H ji nepodporuje. Vstupem modelu jsou čtyři rozměry květů kosatce z datové sady Iris. Strukturu modelu lze vidět na obrázku 2.3.

Na tomto modelu je ověřována základní funkčnost akcelerace pomocí Vitis AI a HLS4ML. Je vyzkoušena kvantizace a kompilace modelu pro DPU a spuštění zkompilovaného modelu. Poté je testována syntéza kernelu pomocí Vivado Accelerator backendu z HLS4ML a jeho spuštění pomocí skriptu z HLS4ML používajícího knihovnu PYNQ.



Obrázek 2.3 Diagram modelu neuronové sítě použitého pro datovou sadu Iris



Obrázek 2.4 Diagram modelu použitého pro datovou sadu CIFAR-10 (ve verzi big_4M8).

2.2 Modely pro datovou sadu CIFAR-10

Pro datovou sadu CIFAR-10 bylo v rámci této práce vytvořeno jedenáct konvolučních neuronových sítí. Modely sdílí téměř stejnou strukturu a liší se v podstatě jen svou velikostí. Architektura je inspirována sítí Alexnet [5].

Všechny obsahují nejprve tři konvoluční vrstvy. Ty jsou proložené třemi max pooling vrstvami. Za nimi pak následují tři plně propojené vrstvy. Konvoluční vrstvy používají kernel velikosti 3×3, krok (stride) o velikosti 1 a rovnoměrný padding ze všech stran. Max pooling vsrtvy používají pool o velikosti 2×2, velikost kroku (stride) je 2 a padding není použit (velikost kroku i poolu je zvolena stejně kvůli kompatibilitě s HLS4ML). Všechny konvoluční i plně propojené vrstvy až na tu poslední používají jako aktivační funkci ReLU. Poslední plně propojená vrstva používá aktivační funkci softmax. Model bere jako vstup barevné obrázky o velikosti 32×32 pixelů. To je velikost obrázků v datové sadě CIFAR-10. Výstupů je deset. Každý pro jednu kategorii obrázků. Udávají odhad pravděpodobnosti, že obrázek je z dané kategorie.

Pro jednotlivé modely se potom liší počet neuronů v plně propojených vrstvách a počet filtrů v konvolučních vrstvách. S trochou abstrakce se dá říct, že jednotlivé modely se liší svou šířkou, ale jejich hloubka zůstává stejná. Nicméně u všech platí, že v konvolučních vrstvách směrem od vstupu počet filtrů roste a v plně propojených vrstvách počet neuronů směrem k výstupu klesá. Modely jsou tedy nejširší uprostřed mezi poslední konvoluční a první plně propojenou vrstvou.

Jedenáct natrénovaných modelů je ve skriptech pojmenovaných podle počtu jejich parametrů

jako: mega_633M, biggest_156M, bigger_39M, big_4M8, medium_701k, small_191k, tiny_57k, tinier_24k, tinier_15k, tinier_8k a tiniest_2k5. Původní je model označený big_4M8, který má 4 831 114 parametrů. Jeho konvoluční vrstvy mají 64, 128 a 256 filtrů a plně propojené vrstvy mají 1024, 256 a 10 neuronů. Model je zobrazen na obrázku 2.4. Další modely jsou od něj odvozené a jejich konkrétní rozměry lze nalézt ve skriptu *cifar-10_train.py*.

Na těchto modelech jsou otestovány kromě plně propojených vrstev i konvoluční a max pooling vrstvy. Na modelech jsou změřeny časy zpracování a spotřeba vytížených karet. Je také postupně vyzkoušeno více verzí DPUCAHX8H pro různé frekvence. Spotřeby a časy zpracování jsou přibližně porovnány s původním modelem v plovoucí řádové čárce spuštěném na GPU.

2.3 Návrh prototypové aplikace pro detekci objektů v obraze

Cílem prototypové aplikace je demonstrovat praktické použití akcelerace neuronových sítí pomocí karty Alveo U55C.

Aplikace bude číst obrázky z kamer humanoidních robotů Nao, posílat je na server a tam v nich detekovat objekty pomocí neuronové sítě akcelerované na kartě Alveo. Zároveň bude možné změřit propustnost, latenci a přesnost akcelerované neuronové sítě.

Aplikace bude sloužit zejména jako prototyp a při výběru technologií je zvolena víceméně cesta vedoucí k co nejjednodušší implementaci. Aplikace je naprogramována v jazyce Python. Komunikace mezi serverem a roboty je realizována pomocí protokolu TCP. Jako neuronová síť pro detekci objektů je zvolen model YOLOv7. Ten je relativně nový, z roku 2022. Je tak reprezentativní vzhledem k technologiím dnes používaným k detekci objektů. Kromě toho Vitis AI obsahuje v Model Zoo a Vitis AI Library příklady akcelerace YOLOv7. Ty jsou sice cílené na novější hardwarové platformy obsahující AI Engines, ale jejich základ lze použít i v rámci této aplikace používající Alveo U55C.

Akcelerace je provedena pouze pomocí frameworku Vitis AI. HLS4ML se během experimentů na modelech pro CIFAR-10 ukázalo jako nevhodné pro takto velké modely.

2.3.1 Kvantizace a kompilace modelu

Před vývojem samotné aplikace je samozřejmě opět nejprve třeba kvantizovat a zkompilovat původní YOLOv7 model. Jako základ této části práce je použit repozitář od AMD obsahující Copyleft Model Zoo. Lze ho najít zde [51]. Konkrétně je použita část o modelu YOLOv7. Ta obsahuje původní PyTorch implementaci YOLOv7 modelu, upravené skripty pro kvantizaci tohoto modelu i hotový kvantizovaný a zkompilovaný model.

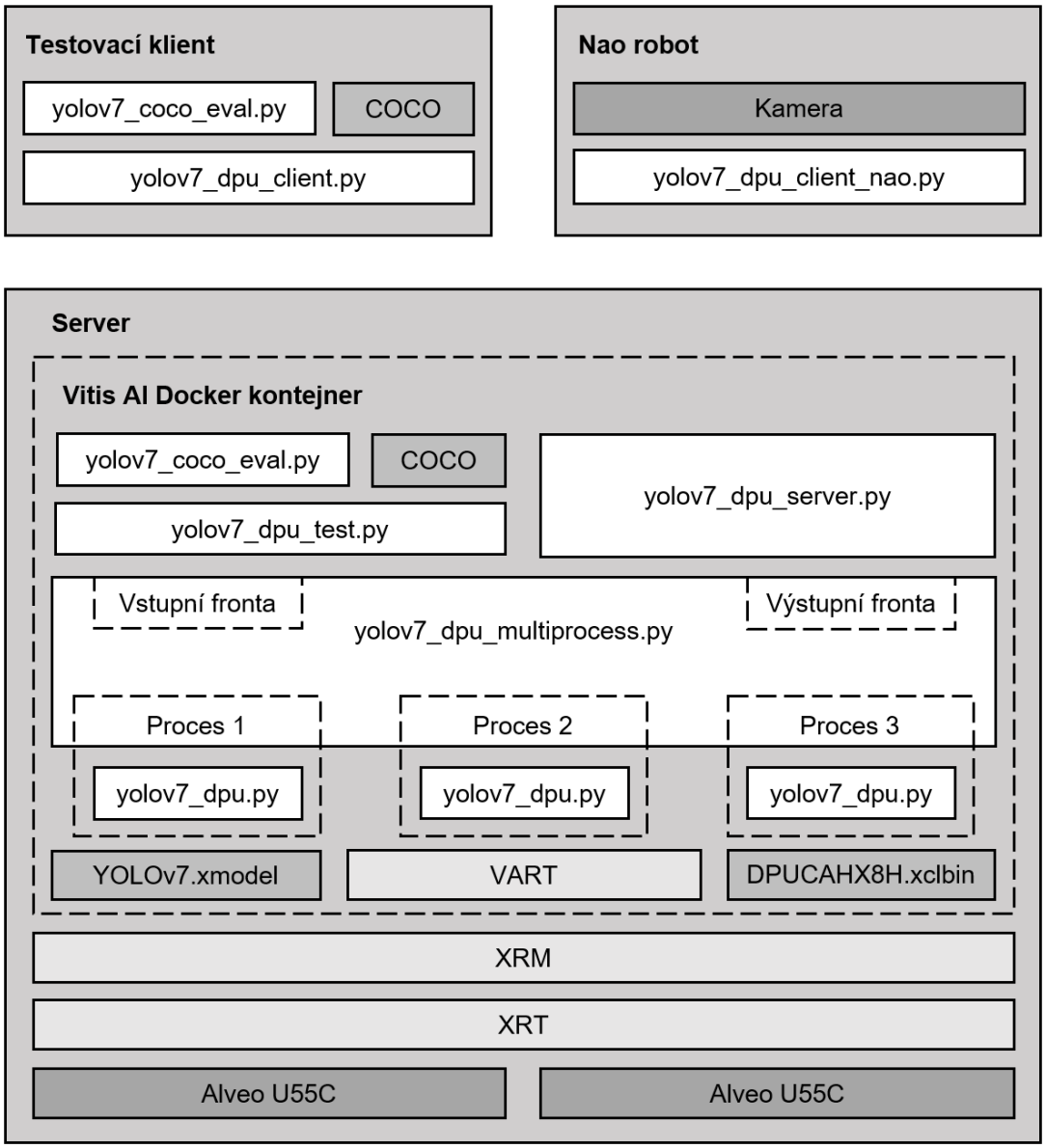
Kvantizace provedená AMD je ale cílená na novější DPU, které zvládají akcelerovat aktivační funkci swish (pomocí její aproximace hard-swish). A to DPUCAHX8H určené pro Alveo U55C neumí. To znamená, že je nutné provést úpravy původního modelu, model s těmito úpravami přetrénovat a poté provést kvantizaci a kompilaci modelu tentokrát pro DPUCAHX8H.

2.3.2 Struktura prototypové aplikace

Struktura celé aplikace je zobrazena na diagramu 2.5.

Základem serverové části je modul *yolov7_dpu.py* obsahující třídu *Yolov7Dpu*. Ta skrze volání VART provádí samotnou akceleraci na DPU. Kromě toho také vykonává nutný pre-processing a post-processing dat. Jejím vstupem tak jsou obrázky libovolné velikosti v RGB formátu. Výstupem je pak pole detekovaných objektů, obsahující souřadnice bounding boxů, třídy objektů a konfidence.

Na tom poté staví *Yolov7DpuMultiprocess*. Ten vnitřně spouští několik procesů, z čehož každý má svou isntanci objektu *Yolov7Dpu*. Jako vnější rozhraní objektu *Yolov7DpuMultiprocess* jsou



Obrázek 2.5 Diagram prototypové aplikace

použity dvě fronty. Jedna vstupní, do ní vstupují obrázky a jejich přiřazené id, a jedna výstupní, kam pracovní procesy vkládají výsledné pole detekovaných objektů a id obrázků, ke kterým výsledky patří. Důvod, proč je zvoleno použití procesů a ne vláken, je ten, že v CPythonu (referenční implementaci Pythonu) není možné vykonávat vlákna skutečně paralelně. Důvodem je Global Interpreter Lock, který umožňuje vykonávat jen jedno vlákno najednou. Pro skutečný paralelismus je nutné právě použití více procesů.

Nad tímto je poté modul *yolov7_dpu_test.py*, který implementuje CLI aplikaci pro načítání obrázků ze souborového systému, jejich zpracování pomocí *Yolov7DpuMultiprocess* a zobrazení výstupu v zadaném formátu. Podporované výstupní formáty jsou: textový výpis na standardní výstup, JSON soubor kompatibilní s pycocotools nebo obrázky s nakreslenými bounding boxy.

Modul *yolov7_dpu_test.py* je určen pro lokální použití. Vytvoření serveru je potom implementováno v modulu *yolov7_dpu_server.py*. Server přijímá spojení skrze TCP. Přes ty pak přijímá zprávy obsahující obrázky. Konec zprávy je rozpoznán dle velikosti obrázku uvedené v hlavičce zprávy. Zpět pak jsou posílána pole detekovaných objektů.

K serveru jsou vytvořeni i příslušní klienti. Modul *yolov7_dpu_client.py* je obdobou modulu *yolov7_dpu_test.py*, jen namísto přímého použití *Yolov7DpuMultiprocess* je implementována komunikace se serverem. Modul *yolov7_dpu_client_nao.py* je potom určen pro použití na robotech Nao. Čte obrázky z kamery, posílá je na server a přijímá data o detekovaných objektech.

Kapitola 3
Implementace

3.1 Akcelerace modelu pro datovou sadu Iris

3.1.1 Vitis AI

3.1.1.1 Příprava XRT a XRM

Před přípravou samotného Vitis AI je nutné mít zprovozněné nástroje XRT a XRM a v kartách Alveo nahrán základní firmware. XRT a XRM nástroje musí být v PATH daného uživatele, toho lze dosáhnout příslušnými skripty, viz výpis 3.1.

```
source /opt/xilinx/xrt/setup.sh
source /opt/xilinx/xrm/setup.sh
```

Výpis kódu 3.1 Přidání XRT a XRM do PATH

Na našem serveru také není centrálně, jako služba, spuštěn XRM démon. Je tedy třeba si ho před použitím DPU ručně spustit. To lze udělat jednoduše pomocí: `xrmd &`.

Funkčnost XRT lze ověřit příkazem `xbutil examine`. Ten vypíše základní informace o XRT a přítomných Alveo kartách. Pomocí přepínačů `--report` a `--device` lze vypsat podrobné informace o konkrétní kartě. Kartu lze také prověřit pomocí `xbutil validate`. Další informace o příkazu xbutil lze nalézt v jeho nápovědě (`--help`).

Funkčnost XRM lze ověřit pomocí utility xrmadm. Ta bere jako vstup JSON soubory s konkrétními příkazy. Pár takových souborů je už ale připravených od AMD. Použít lze například `xrmadm /opt/xilinx/xrm/test/list_cmd.json`. Ten vypíše informace o dostupných kartách a v nich nahraných xclbin souborech. Případně se může hodit příkaz pro vymazání aktuálně nahraných xclbin souborů: `xrmadm /opt/xilinx/xrm/test/unload_devices_cmd.json`.

3.1.1.2 Příprava Vitis AI nástrojů

Pro kvantizaci, kompilaci i následné spuštění modelu neuronové sítě pomocí Vitis AI je potřeba několik různých nástrojů a knihoven (viz sekce 1.3.2). Nejjednodušším způsobem jejich instalace je použití připraveného Vitis AI Docker kontejneru. Ten lze stáhnout skrze Docker Hub. Na našem serveru už je stažený a lze ho zobrazit pomocí příkazu: `docker image ls`.

Kontejner se potom používá společně s Vitis AI repozitářem, který obsahuje skript pro jeho spuštění, *docker_run.sh*. Je tedy vhodné si naklonovat i Vitis AI repozitář v příslušné verzi. Stažení kontejneru a repozitáře a spuštění kontejneru lze vidět ve výpisu 3.2.

```
docker pull xilinx/vitis-ai-cpu:2.5
git clone --depth 1 --tag "v2.5"  https://github.com/Xilinx/Vitis-AI.git
cd Vitis-AI
./docker_run.sh xilinx/vitis-ai-cpu:2.5
```

Výpis kódu 3.2 Stažení a spuštění Vitis AI kontejneru

Skript *docker_run.sh* do kontejneru během spouštění mimo jiné připojí některé složky z hlavního operačního systému (přes *volume mount*). Připojena je například složka samotného Vitis AI repozitáře. Ten je použit jako hlavní pracovní prostor. Pro testování skriptů z této práce je tedy vhodné si vše vložit do kořenové složky Vitis AI repozitáře.

Další důležitou připojenou složkou je /opt/xilinx/overlaybins. Tam se vkládají xclbin soubory s DPU. Ty lze buď už hotové stáhnout pomocí skriptů umístěných ve Vitis AI repozitáři ve složce setup nebo board_setup (dle verze Vitis AI), anebo syntetizovat z xo souborů. V této práci je použito DPUCAHX8H syntetizované vedoucím práce v několika verzích pro různé frekvence.

Pro použití konkrétního DPU souboru skrze VART je potom nutné v kontejneru nastavit systémové proměnné XLNX_VART_FIRMWARE a XCLBIN_PATH. V případě standardních stažených DPU souborů je toto možné udělat automaticky pomocí příslušných skriptů. V našem případě je to třeba provést ručně. Na výpisu 3.3, lze vidět nastavení pro verzi DPU na frekvenci 300 MHz.

```
export XCLBIN_PATH=\
/opt/xilinx/overlaybins/dpu_u55c_base_3_202210_1_3-4-4-300.xclbin
export XLNX_VART_FIRMWARE=\
/opt/xilinx/overlaybins/dpu_u55c_base_3_202210_1_3-4-4-300.xclbin
```

Výpis kódu 3.3 Nastavení DPU pro VART

Zároveň je vhodné uvnitř kontejneru také nastavit systémové proměnné pro použití XRT a XRM. To se opět provede pomocí příslušných skriptů, viz výpis 3.1. Spouštění modelů funguje i bez toho, avšak nefungovaly by utility jako xbutil nebo xrmadm.

3.1.1.3 Verze Vitis AI

Zde je nutné udělat ještě jednu odbočku. Aktuální verze Vitis AI je 3.5. Vitis AI Library ve verzi 3.5 dle dokumentace (viz [52]) oficiálně podporuje akceleraci jen pomocí zařízení Versal VEK280 a Alveo V70. V hlavní dokumentaci Vitis AI toto explicitně zmíněno není. Nicméně ve verzi 3.5 v Model Zoo jsou ukázky modelů připravené pouze pro VEK280 a V70 a Vitis AI repozitář obsahuje skripty pouze pro stažení DPU pro VEK280 a V70.

Alveo U55C má plnou podporu naposledy ve verzi 2.5. Ta obsahuje jak kompatibilní modely v Model Zoo, tak xclbin soubory s DPU. Jenže tyto xclbin soubory jsou ještě pro firmware ve verzi *base_2*. Na našem serveru je na kartách instalován firmware ve verzi *base_3*, který se ukázal jako nekompatibilní s DPU pro *base_2*. To je hlavní důvod, proč jsou na našem serveru použité vlastní xclbin soubory syntetizované z poskytnutých xo souborů. Syntéza totiž byla provedena právě už pro firmware *base_3*.

Vitis AI kompilátor nicméně dle testování zvládne zkompilovat model pro Alveo U55C v obou verzích 2.5 i 3.5. Takže pro většinu této práce je možné použít libovolnou z těchto dvou verzí. Verze 3.5 nebude obsahovat v Model Zoo modely přímo zkompilované pro Alveo U55C. Verze 2.5 zase nebude mít v Model Zoo ukázky kvantizace s novějšími modely neuronových sítí a Vitis AI Library nebude obsahovat kódy pro spuštění těchto novějších modelů.

Během testování se také ukázalo, že v kontejneru ve verzi 3.5 nefunguje knihovna PyTorch. Po importu Python vždy spadne s chybou „Illegal instruction (core dumped)“.

V této práci je během testování modelů pro datové sady Iris a CIFAR-10 použit kontejner i repozitář ve verzi 2.5. Pro vývoj prototypové aplikace s modelem YOLOv7 je použit kontejner i repozitář ve verzi 3.5. Nicméně aplikace bez problémů běží i ve verzi 2.5.

Příprava Vitis AI kontejneru a repozitáře ve verzi 2.5 je vidět na výpisu 3.2. Při použití verze 3.5 je potom třeba zvolit konkrétní kontejner, buď pro PyTorch nebo TensorFlow. Například xilinx/vitis-ai-pytorch-cpu:latest.

3.1.1.4 Naučení modelu

Učení modelu bylo provedeno ve skriptu *iris_learn.py*. Ten vychází z ukázkového skriptu z předmětu NI-ESW a je v něm pomocí Keras rozhraní implementována jednoduchá neuronová síť dle struktury popsané v sekci 2.1. Pro učení je použit Adam s *learning rate* 0,05. Provedeno je 1000 epoch. Síť po učení obvykle dosáhne 148 správných předpovědí ze 150 prvků v datové sadě. Výsledný model je uložen do souboru *iris.keras*.

Díky malé velikosti modelu je možné učení bez problému provést na CPU. Bylo tedy provedeno přímo ve Vitis AI kontejneru, který už obsahuje potřebné nástroje (jen je třeba aktivovat conda prostředí pro TensorFlow 2).

3.1.1.5 Kvantizace a kompilace modelu

Kvantizace je provedena pomocí PTQ. PTQ neuronové sítě z frameworku Keras (TensorFlow 2) ve Vitis AI probíhá pomocí rozhraní přímo v Pythonu. Je třeba předat model a trochu vstupních dat, na kterých se provede kalibrace. Správné výstupy třeba nejsou. V tomto případě je předána celá datová sada. Rozhraní pro kvantizaci lze vidět ve výpisu 3.4. Celý kód včetně načtení modelu a datové sady je ve skriptu *quantize.py*. Kvantizovaný model je uložen do souboru *iris_quantizied.h5*.

```
from tensorflow_model_optimization.quantization.keras import vitis_quantize
quantizer = vitis_quantize.VitisQuantizer(model)
quantized_model = quantizer.quantize_model(calib_dataset=x_train_norm, \
    calib_batch_size=len(x_train_norm))
```

Výpis kódu 3.4 Kvantizace modelu z TensorFlow 2 pomocí Vitis AI

Kompilátor modelu už je samostatná utilita. Její použití lze vidět ve výpisu 3.5. Je nutné specifikovat vstupní kvantizovaný model, architekturu, pro kterou se provádí kompilace, a výstupní umístění a jméno zkompilovaného modelu. Výstupem je soubor s příponou xmodel. Ten už lze pomocí VART spustit na Alveo kartě.

```
vai_c_tensorflow2 -m ./iris_quantizied.h5 \
    -a /opt/vitis_ai/compiler/arch/DPUCAHX8H/U55C-DWC/arch.json \
    -o ./iris_compiled -n iris
```

Výpis kódu 3.5 Kompilace kvantizovaného modelu z TensorFlow 2 pomocí Vitis AI

3.1.1.6 Spuštění modelu na Alveo U55C

Spuštění modelu na kartě probíhá pomocí VART. Ten má rozhraní v C++ a v Pythonu. Obě jsou velice podobná. To v C++ je o něco lépe dokumentované. Nicméně v této práci je používán VART skrze Python.

Hlavní komponentou VART je tzv. DPU Runner. To je objekt, skrze který probíhá inference v DPU na kartě. Při vytváření je mu předán model, respektive jeho podgraf spustitelný na DPU. VART se během toho postará o automatické nahrání xclbin souboru s DPU do karty (pokud se to ještě nestalo) a poté jednu dostupnou jednotku DPU přidělí danému DPU Runner objektu. Vytvoření DPU objektu lze vidět ve výpisu 3.6. Je načten XIR graf z xmodel souboru. Poté jsou vybrány subgrafy spustitelné na DPU. V tomto případě se předpokládá, že část spustitelná na DPU je souvislá a podgraf by tak měl být jen jeden. Poté je vytvořen samotný DPU Runner.

```
graph = xir.Graph.deserialize(xmodel_filepath)
subgraphs = get_child_subgraph_dpu(graph)
assert len(subgraphs) == 1
dpu_runner = vart.Runner.create_runner(subgraphs[0], "run")
```

Výpis kódu 3.6 Vytvoření objektu DPU Runner

Podoba rozhraní objektu DPU Runner pro vstupní a výstupní data je ovlivněna jak původním modelem, tak přidělenou DPU jednotkou. Vstupní a výstupní data bývají vždy osmibitová čísla v pevné řádové čárce. Nicméně pozice řádové čárky se může lišit v závislosti na tom, jak byl původní model kvantizován. Konkrétní DPU jednotka zase ovlivňuje velikost vstupních a výstupních bufferů. Například naše syntetizované xclbin soubory s DPUCAHX8H obsahují 3 DPU jednotky (každá v jedné ze tří oblastí FPGA). Jednu s velikostí dávky 3 a dvě s velikostí dávky 4. Velikost dávky je dána počtem enginů vytvořených v dané oblasti. Pokud je vytvořen pouze jeden DPU Runner je mu obvykle přidělena první jednotka s dávkou o velikosti 3. Tyto informace o formátu vstupů a výstupů je možné získat pomocí metod objektu DPU Runner, `get_input_tensors()` a `get_output_tensors()`. Podle těchto informací je potom nutné alokovat vstupní a výstupní buffery a provést případný preprocessing a postprocessing, například převedení dat z plovoucí řádové čárky do fixní a zpět. V našem případě je také ještě nutné nad výstupními daty provést softmax funkci, neboť tu DPU spočítat neumí.

Samotné zpracování dat pomocí DPU je potom spuštěno voláním funkce `execute_async()`. Ukázku lze vidět ve výpisu 3.7.

```
job_id = dpu_runner.execute_async(batch_input_data, batch_output_data)
dpu_runner.wait(job_id)
```

Výpis kódu 3.7 Zpracování dat pomocí objektu DPU Runner

Celá akcelerace modelu pro Iris byla implementována ve skriptu *deploy.py*. Skript bere z příkazové řádky jeden argument, a to cestu k xmodel souboru se zkompilovným modelem. Nejdříve dojde k vytvoření objektu DPU Runner a vypsání informací o něm. Poté je pomocí něj zpracována datová sada Iris. Nakonec je to samé provedeno původním modelem spuštěným na CPU. Výsledky jsou poté porovnány a vypsány.

Naměřené výsledky lze nalézt v souboru *results.txt*. Akcelerovaná verze modelu dělá o jednu chybu více než původní model. Má správně 147 ze 150 vstupů. Přesnost se tedy dle očekávání kvůli kvantizaci lehce snížila.

3.1.2 HLS4ML

3.1.2.1 Příprava

V této práci byl pro syntézu použitý školní server livsgpu01, který má nainstalovány nástroje Vitis a Vivado nutné pro syntézu. Na serveru s Alveo kartami pak bylo testováno pouze spouštění hotových kernelů.

Před syntézou pomocí HLS4ML je třeba mít všechny nástroje v PATH. To lze opět provést pomocí příslušných skriptů od AMD, viz výpis 3.8.

```
source /opt/xilinx-2022/Vivado/2022.2/settings64.sh
source /opt/xilinx-2022/Vitis/2022.2/settings64.sh
```

Výpis kódu 3.8 Přidání nástrojů Vivado a Vitis do PATH

Dále je třeba nainstalovat potřebné Python balíčky. Zejména TensorFlow a samotné HLS4ML. Co se týče HLS4ML, tak je nutné instalovat upravenou verzi vytvořenou pro tuto práci. Před instalací je vhodné odstranit předchozí instalované verze HLS4ML. Všechny Python nástroje je vhodné instalovat do vlastního Python Virtual Environment (zkráceně venv). Pro instalaci stejných verzí Python nástrojů, jaké byly použity v této práci, je možné použít soubor *requirements_training_and_hls4ml_synth.txt* z části o modelech pro CIFAR-10. Jsou tam zahrnuty nástroje pro učení a inferenci pomocí TensorFlow 2 (včetně knihoven pro akceleraci na GPU) a veškeré potřebné závislosti pro HLS4ML. Samotný balíček HLS4ML tam vložen není, je potřeba lokálně nainstalovat jeho upravenou kopii z této práce.

Pro tuto část práce o akceleraci modelu pro Iris pomocí HLS4ML byla zvolena forma Jupyter notebooku, který obsahuje jak samotný kód, tak jeho zaznamenaný výstup. Nicméně dále už jsou opět používány normální Python skripty. Pro případné spuštění tohoto Jupyter notebooku je vhodné se připojit na server přes VNC a spustit ho třeba skrze Visual Studio Code. Pro využití venv v Jupyteru je nutné z něj vytvořit kernel pomocí příkazu `ipython kernel install`.

Co se týče serveru s Alveo kartami, tak je nutné mít nainstalovanou knihovnu PYNQ (v práci je použita verze 3.0.1). Pro správné fungování PYNQ je třeba ručně instalovat ještě knihovnu IPython. Stejně jako u Vitis AI je také třeba mít nainstalované XRT, které musí být v PATH (viz sekce 3.1.1.1). Kromě toho je nutný také skript *axi_stream_driver.py* z HLS4ML, takže je vhodné si nainstalovat i HLS4ML. Všechno je opět ideální instalovat do samostatného venv.

3.1.2.2 Úprava HLS4ML

Pro testování akcelerace pomocí HLS4ML byl zvolen VivadoAccelerator backend, jakožto nejjednodušší způsob pro akceleraci pomocí HLS4ML. Měl by automaticky provést vše od vysokoúrovňové syntézy po vytvoření xclbin souboru. Zároveň obsahuje i hotový Python skript pro ovládání vytvořeného kernelu. Jenže ve výchozím stavu se tento backend pro potřeby této práce ukázal jako nefunkční. Hlavní důvody jsou dva. Za prvé backend ve výběru zařízení nenabízí přímo kartu Alveo U55C. A za druhé je backend připraven pro HLS pomocí Vivado HLS (do verze 2020.2). Jenže v této práci bylo nutné použít nástroje Vitis a Vivado ve verzi 2022.2, což znamená, že byla automaticky použita Vitis HLS (verze 2022.2) a ta není s původní Vivado HLS plně zpětně kompatibilní.

To znamená, že bylo třeba v HLS4ML provést úpravy. Upravená verze vychází z originálního HLS4ML ve verzi 0.8.1.

Prvně bylo třeba přidat desku Alveo U55C mezi podporované zařízení do souboru *supported_boards.json*. Byla použita stejná konfigurace jako pro ostatní podporované Alveo desky, pouze bylo zvoleno správné označení FPGA (pro Alveo U55C je to xcu55c-fsvh2892-2L-e).

Poté se ukázal právě problém s chybějící zpětnou kompatibilitou mezi Vitis HLS a Vivado HLS. Vnější rozhraní kolem implementované neuronové sítě nešlo pomocí Vitis HLS syntetizovat. Bylo nutné jeho šablony ve Vivado Accelerator backendu přepsat za použití hls::axis a hls::stream.

To pro změnu rozbilo kódy pro C-simulaci. Tento problém byl vyřešen jednoduše simulováním pouze vnitřní sítě bez tohoto vnějšího rozhraní.

Poté už proběhla HLS v pořádku. Nicméně se objevily problémy v dalších krocích. Vivado-Accelerator backend po provedení HLS, vytvoří Vivado IP. To je potom importováno do Vivado a obaleno Verilog kódem pro celkové vnější rozhraní kernelu a celý výsledek je exportován jako

xo soubor. V tomto kroku se ale objevil problém se jmény entit ve Verilogu. V připraveném Verilog kódu bylo na několika místech použito označení myproject. Jenže skutečné entity měly jméno shodné se skutečným jménem projektu. Na tato místa bylo třeba přidat automatické přejmenování.

Další problematickou částí bylo rozhraní hotového kernelu. Kernel automaticky používal datový typ ap_uint<32>. Jenže knihovna PYNQ, tento datový typ nebyla schopna rozeznat. Bylo třeba přidat nový argument pro příkaz package_xo, který donutil kernel použít navenek datový typ unsigned int. Úprava byla provedena v souboru *axi_stream_design.tcl*. S typem unsigned int už si knihovna PYNQ poradila.

Po těchto úpravách už se podařilo syntetizovat funkční kernel. Nicméně během testů modelů pro CIFAR-10 byla provedena ještě jedna změna. Třída VivadoAcceleratorWriter byla pozměněna tak, aby dědila ze třídy VitisWriter namísto VivadoWriter. Tím bylo docíleno použití šablon pro neuronové sítě z backendu Vitis. Ty jsou oproti backendu Vivado modifikovány pro kompatibilitu s Vitis HLS. Toto pomohlo lehce redukovat problémy s *routing congestion*.

Veškeré změny v knihovně HLS4ML byly provedeny v samostatných commitech v rámci repozitáře pro tuto práci. Tyto commity mají v popisu vždy prefix hls4ml_xilinx_2022.2 nebo hls4ml_xilinx_2022.2_fix. Konkrétní provedené změny by tedy nemělo být příliš těžké dohledat.

Zprovoznění HLS4ML byla jedna z více časově náročných částí této práce. Zvolená cesta přes úpravu Vivado Accelerator backendu a spouštění skrze PYNQ určitě nebyla jediná možnost. Alternativně by mělo být možné třeba automaticky vygenerovat jen kód pro HLS, pak z něj ručně pomocí Vitis a jeho Vitis Kernel Flow přímo syntetizovat kernel a ten ovládat jen skrze XRT. To by mohlo být potenciálně elegantnější. Nicméně jako první se podařilo zprovoznit upravený Vivado Accelerator backend a další postupy už potom nebyly testovány.

3.1.2.3 Syntéza modelu pomocí HLS4ML

Učení, simulace HLS4ML modelu i jeho syntéza pomocí Vivado Accelerator backendu byly provedeny v Jupyter notebooku *iris_hls4ml.ipynb*.

Učení modelu pro Iris bylo provedeno v podstatě stejně, jako bylo popsáno v sekci 3.1.1.4.

Vytvoření akceleračního kernelu v HLS4ML poté probíhá v několika krocích. Nejdříve je třeba naimportovat naučený model. Během toho se také nastaví parametry HLS4ML, například reuse factor. Celý import a nastavení parametrů lze vidět ve výpisu 3.9.

Ve chvíli, kdy máme v Pythonu inicializovaný HLS4ML model, tak je možné pomocí metody `hls_model.write()` vytvořit HLS projekt pro Vivado/Vitis. Do něj se zapíšou veškeré vygenerované kódy v jazyce C určené pro vysoko-úrovňovou syntézu. Projekt je zapsán do složky specifikované během importu.

Poté je možné provést softwarovou simulaci. Model se zkompiluje pomocí volání metody `hls_model.compile()`. Metoda `hls_model.predict()` potom funguje totožně jako ta z nástroje Keras.

Nakonec přichází na řadu metoda `hls_model.build()`. Ta dle nastavených parametrů spouští vestavěnou C-simulaci, HLS i plnou syntézu a generování bitstreamu. Simulace pomocí `build()` je alternativou k simulaci pomocí `compile()` a `predict()`. Před jejím použitím je vhodné do HLS projektu umístit data pro testování (do složky tb_data). Volání `build()` s parametrem `bitfile` nastaveným na `True` postupně provede celý proces vygenerování xclbin souboru. Funkce po dokončení vrátí parametry vygenerovaného kernelu, zejména počty použitých komponent FPGA.

Je třeba mít na paměti, že plné generování xclbin souboru je značně časově náročná operace. Na našem serveru livsgpu01 to pro maličkou neuronovou síť pro Iris trvalo přibližně 2 hodiny. I pro menší varianty modelů pro CIFAR-10 to ale zabralo už obvykle i kolem 8 hodin.

3.1.2.4 Spuštění syntetizovaného kernelu

Spuštění syntetizovaného modelu na Alveo kartě je realizováno pomocí modulu *axi_stream_driver* z HLS4ML knihovny. Ten vnitřně používá knihovnu PYNQ. Kromě toho zajišťuje alokaci bufferů

```
model = tf.keras.models.load_model('iris.keras')
config = hls4ml.utils.config_from_keras_model(
        model=model,
        granularity="model",
        )
config["Model"]["ReuseFactor"] = 1
config["Model"]["Strategy"] = "Latency"
config["Model"]["Precision"] = "ap_fixed<16,6>"
hls_model = hls4ml.converters.convert_from_keras_model(model,
        hls_config=config,
        project_name="iris_hls4ml_prj",
        output_dir="iris_hls4ml_prj",
        io_type="io_stream",
        board="alveo-u55c",
        backend="VivadoAccelerator",
        part="xcu55c-fsvh2892-2L-e"
        )
hls_model.config.config["ClockPeriod"] = 10
hls_model.config.config["AcceleratorConfig"]["Platform"] =
        "xilinx_u55c_gen3x16_xdma_3_202210_1"
```

Výpis kódu 3.9 Import a nastavení parametrů HLS4ML modelu

pro komunikaci skrze PYNQ a navenek vytváří rozhraní v podobě funkce `predict()`, která je opět inspirovaná tou z nástroje Keras.

Pro použití tohoto modulu je jednoduchou cestou prosté zkopírování modulu do složky se skriptem, který ho volá, a pak jednoduchý import.

Ukázku použití lze vidět ve výpisu 3.10. Celý kód včetně načtení datové sady a vypsání výsledků je ve skriptu *iris_hls4ml_deploy.py*. Jeho výstup lze vidět v *deployment_output.txt*.

```
from axi_stream_driver import NeuralNetworkOverlay
nn = NeuralNetworkOverlay(xclbin_name = 'iris_hls4ml_prj_kernel.xclbin')
y_predict = nn.predict(X = x_train_norm, y_shape=y_train.shape)
nn.free_overlay()
```

Výpis kódu 3.10 Spuštění syntetizovaného HLS4ML kernelu

3.2 Akcelerace modelů pro datovou sadu CIFAR-10

3.2.1 Příprava

Před trénováním a akcelerací modelů je nutné si stáhnout samotnou datovou sadu CIFAR-10, jak lze vidět ve výpisu 3.11. Skripty v repozitáři této práce počítají s tím, že se datová sada nachází ve složce *cifar-10/cifar-10-batches-py*.

```
wget https://www.cs.toronto.edu/~kriz/cifar-10-python.tar.gz
tar -xzvf cifar-10-python.tar.gz
```

Výpis kódu 3.11 Stažení datové sady CIFAR-10

Příprava frameworků HLS4ML a Vitis-AI už byla popsána v sekci o akceleraci modelů pro Iris.

3.2.2 Trénování modelů

Pro CIFAR-10 bylo vytvořeno a natrénováno 11 modelů konvolučních sítí. Jejich struktura a pojmenování už bylo popsáno v sekci 2.2.

Trénování, tentokrát už na rozdíl od malého modelu pro Iris, bylo provedeno pomocí dvou GPU A100 40GB na serveru livsgpu01. Potřebné nástroje (a jejich verze) pro učení na GPU jsou uvedeny v už zmíněném souboru *requirements_training_and_hls4ml_synth.txt*.

Trénování bylo vyladěno pro model big_4M8. Byla využita optimalizace Adam s *learning rate* 0,001. Provedeno bylo 250 epoch. Pro lepší výsledky byly použity dropout vrstvy i augmentace trénovacích dat pomocí zrcadlení, změny jasu a prohození barevných kanálů. Model big_4M8 nakonec dosáhl na validační datové sadě CIFAR-10 přesnosti 87,92 %. Pro dosažení dalšího výrazného zvýšení přesnosti by už bylo nejspíš třeba mít větší množství trénovacích dat.

Pro další odvozené velikosti modelů už nebyla výsledné přesnosti věnována tak velká pozornost. Pro učení byly použity téměř totožné parametry jako pro model big_4M8, pozměněny byly jen míry dropout na jednotlivých vrstvách. Odvozené modely už tedy dosahovaly přesnosti o něco nižší. Z hlediska akcelerace není ale tak důležitá výsledná přesnost jako spíš míra jejího snížení v akcelerované verzi modelu.

3.2.3 Vitis AI

3.2.3.1 Kvantizace a kompilace modelů

Postup pro kvantizaci i kompilaci je prakticky totožný s tím použitým u modelu pro Iris (viz sekce 3.1.1.5). Kvantizace je opět provedena formou PTQ. Naprogramována je ve skriptu *cifar-10_quantize.py*. Kompilaci lze provést příkazem vai_c_tensorflow2.

Oproti modelu pro Iris, kde byl trénink proveden přímo ve Vitis AI kontejneru, se zde ale projevil problém s přenosem Keras modelů mezi různými verzemi TensorFlow 2. Na serveru livsgpu01 byly modely učeny v TensorFlow 2.15. Ve Vitis AI kontejneru ve verzi 2.5 je ale instalován TensorFlow 2.8 a v něm nebylo možné Keras modely z TensorFlow 2.15 importovat. Bylo třeba místo formátu Keras použít formát h5. Ani to nefungovalo bez problému. Nicméně když se ve skriptu *cifar-10_quantize.py* znovu naprogramovala struktura modelu a ze souboru h5 se importovaly čistě jen hodnoty parametrů, tak se nakonec podařilo modely mezi různými verzemi přenést.

Kromě tohoto problému se podařilo všechny modely až na dva největší (mega_633M a biggest_156M) úspěšně kvantizovat a zkompilovat. Dva největší se zkompilovat nepodařilo, neboť DPUCAHX8H nepodporuje plně propojené vrstvy o více než 4096 neuronech.

Přestože byla použita jen PTQ, tak snížení přesnosti bylo minimální, vždy nižší než půl procenta. QAT proto pro tyto modely nebyl testován.

3.2.3.2 Spuštění modelů na Alveo U55C

Spouštění modelu pomocí VART je naprogramováno ve skriptu *cifar-10_dpu_deploy.py*. Kód vychází z implementace pro Iris (viz sekce 3.1.1.6). Nicméně aby bylo možné smysluplně měřit časy zpracování, bylo nutné provést několik změn, kvůli nízké efektivitě původního kódu.

Kód je přepsán tak, že jsou předem alokovány vstupní a výstupní buffery pro všechna připravená data. Jednotlivé dávky jsou spouštěny ve větším množství hned po sobě. Nečeká se na dokončení předchozí dávky. Je nastaveno pouze omezení na počet najednou spuštěných dávek. O postupné zpracování dat na kartě a přenosy dat mezi kartou a hlavní pamětí serveru se postará VART.

Další změnou je optimalizace výpočtu softmax funkce, který je prováděn na CPU. Kód je přepsán, tak aby zpracovával všechna data najednou pomocí operací knihovny Numpy namísto původně použitých Python for cyklů. To zrychlilo post-processing asi 10 krát. Čas nutný na post-processing se tak stal výrazně nižší než čas samotného zpracování na DPU a neovlivňuje tak příliš moc výsledné časy.

Kód je také upraven pro použití všech tří DPU jednotek dostupných na jedné Alveo kartě. Použitý xclbin soubor pro DPUCAHX8H totiž obsahuje tři DPU jednotky. Jednu s velikostí dávky 3 a dvě s velikostí dávky 4.

Skript automaticky měří čas zpracování. Naměřen je čas nutný pouze pro zpracování pomocí DPU, poté čas zpracování i s výpočtem softmax funkce a nakonec i celkový čas včetně incializace DPU.

Kromě toho je během testu také průběžně měřen elektrický odběr karty. Před zpracováním dat pomocí DPU je spuštěn skript *measure_power_alveo.sh*. Ten každou sekundu přečte aktuální spotřebu pomocí nástroje xbutil (viz sekce 3.1.1.1). Po dokončení zpracování dat jsou naměřená data vypsána na standardní výstup skriptu *cifar-10_dpu_deploy.py*.

Pro souhrnné měření všech modelů byl vytvořen skript *cifar-10_dpu_deploy_all_models.sh*. Ten spustí všechny modely pro CIFAR-10 pro velikosti vstupů 1, 10000, 50000 a 100000 obrázků. Výsledky jsou uloženy do souborů ve složce *deployment_dpu_output*. Testy jsou prováděny vždy na validační datové sadě z CIFAR-10. Pro větší velikosti vstupu je sada jen několikrát zopakována.

Naměřené spotřeby, přesnosti a časy zpracování jsou uvedeny v kapitole 4.

3.2.3.3 Různé frekvence DPUCAHX8H a externí napájení

V této fázi práce byly také postupně otestovány všechny naše verze DPUCAHX8H syntetizované pro různé frekvence od 100 MHz po 300 MHz. Originální xclbin soubor od AMD (syntetizovaný ale jen pro starší firmware verze base_2) je vytvořen pro 300 MHz.

Karta byla původně připojena v serveru pouze pomocí PCIe bez externího napájecího konektoru. To fungovalo pouze pro DPU na frekvenci 100 MHz a pouze při vytížení maximálně dvou DPU jednotek. Při vytížení třetí jednotky (a tím vytížení třetí oblasti FPGA) se karta dostala do problémového stavu, kdy ji nebylo možné ovládat. Řešením byl až restart serveru.

Karta před přechodem do problémového stavu většinou ukazovala spotřebu lehce přes 40 W. Napájení skrze PCIe by teoreticky mělo být schopné dodat až 75 W. Nicméně po připojení externího napájecího konektoru (až 225 W) problém zmizel.

Poté byly postupně otestovány všechny frekvence DPU až po 300 MHz. S žádnou verzí už nenastaly další problémy. I při plném vytížení verze pro 300 MHz se odběr držel kolem 53 W a teplota karty pod 60 °C.

3.2.4 HLS4ML

Během prvních pokusů s konvolučními vrstvami se ukázalo, že HLS4ML nepodporuje max pooling vrstvy s velikostí kroku jinou než je velikost poolu (minimálně pro Keras modely). To ovlivnilo zvolenou strukturu modelů pro CIFAR-10, kde na základě toho byla zvolena velikost kroku i poolu 2.

Kód pro generování kernelů skrze HLS4ML, jejich simulaci i jejich spouštění opět vychází z toho použitého u modelu pro Iris (viz sekce 3.1.2.3 a 3.1.2.4). Generování kernelu a jeho simulaci lze nalézt ve skriptu *cifar-10_hls4ml.py*. Skript obsahuje spouštění simulace skrze metody compile a predict i přípravu dat pro C-simulaci skrze Vitis. Spuštění syntetizovaného kernelu na kartě je pak ve skriptu *cifar-10_hls4ml_deploy.py*. Zde byly provedeny pouze změny související s testováním různě velkých vstupů a měřením časů zpracování.

Prvně byla testována simulace modelů na CPU. Malé modely fungovaly bez problému, u větších (například big_4M8) bylo nutné zvýšit maximální velikost stacku. Pro zvýšení na 65 MiB

lze použít příkaz `ulimit -s 65536`. Simulace větších modelů se ukázala také jako značně časově náročná.

Na základě simulace byl určen datový typ kernelu, který nepovede k výrazné ztrátě přesnosti. Jako takový byl vybrán typ ap_fixed<24,8>, 24 bitový typ s 8 bitovou celočíselnou částí a 16 bitovou zlomkovou částí.

V rámci této části práce byl vytvořen i skript *cifar-10_intermediate_output.py*. Ten ukazuje minimální a maximální hodnotu pro váhy na jednotlivých vrstvách a data procházející mezi vrstvami modelu. Pomocí něho pak proběhly pokusy o nastavení datových typů zvlášť pro jednotlivé vrstvy. Pro model big_4M8 například nebyl problém potom ručně snížit velikost většiny vah jen na 12 bitů bez zásadního snížení přesnosti simulace. Alternativně by bylo možné použít knihovnu QKeras. Nicméně pro další testy byl nakonec ponechán pro všechny vrstvy jednotný typ ap_fixed<24,8>.

Poté byla otestována syntéza kernelů. Vzhledem k větší velikosti modelů byla použita strategie *resource* a io_type *io_stream*. Syntéza byla prováděna pro cílovou frekvenci 100 MHz.

Jako první proběhl pokus o syntézu modelu big_4M8. Jenže ten se ukázal jako příliš velký. Už během HLS došlo k problému s nedostatkem paměti serveru. Vysoko-úrovňová syntéza zabrala celých 128 GB RAM, které byly na serveru k dispozici a poté pro nedostatek další paměti spadla.

Poté byly tedy provedeny pokusy s menšími modely (od tiniest_2k5 po tiny_57k). U nich už HLS proběhla v pořádku. Nicméně většina pokusů bohužel skončila neúspěchem během fáze logic routing. Objevil se totiž problém s *routing congestion* (nedostatkem cest v FPGA).

Během těchto testů byla provedena ještě jedna změna v rámci Vivado Accelerator backendu, a to použití HLS4ML šablon z Vitis backendu namísto Vivado backendu. Změna už byla zmíněna v sekci 3.1.2.2. To lehce zmírnilo problémy s *routing congestion*, ale nevyřešilo je to.

Úspěšně syntetizovat se tak podařilo pouze modely tiniest_2k5, tinier_8k a tinier_15k. Větší modely se syntetizovat nepodařilo.

Podrobnosti o všech provedených experimentech jsou uvedeny v kapitole 4.

3.2.5 GPU akcelerace

Pro přibližné porovnání akcelerace na FPGA s tou na GPU byly implementovány i skripty pro spuštění modelů na GPU. Skripty jsou pojmenovány *cifar-10_deploy.py*, *measure_power_gpu_0.sh* a *cifar-10_deploy_all.sh*. Plní obdobnou funkci jako skripty připravené pro akceleraci pomocí Vitis AI (viz sekce 3.2.3.2). Akcelerace je provedena jednoduše pomocí funkce predict z rozhraní Keras. Pokud jsou instalovány potřebné nástroje, síť by měla být automaticky spuštěna na GPU. Pro měření spotřeby je použita utilita nvidia-smi.

3.3 Prototypová aplikace pro detekci objektů v obraze

3.3.1 Příprava GPU serveru pro kvantizaci

Jak už bylo popsáno v sekci 2.3.1, jako základ pro kvantizaci byla použita část Copyleft Model Zoo repozitáře od AMD. Jeho použitou a upravenou část lze najít v repozitáři této práce ve složce *yolov7_train_qaunt*.

Jenže operace nutné pro kvantizaci tohoto modelu se ukázaly být velmi výpočetně náročné a bylo nutné kvantizaci provést pomocí GPU na serveru livsgpu01.

Hotový Vitis AI kontejner, který lze stáhnout přes Docker Hub, bohužel neobsahuje podporu pro použití GPU. Proto byl zvolen postup samostatné instalace vai_q_pytorch a jeho závislostí (xir, unilog a pybind11). Tu je nutné udělat kompilací ze zdrojových kódů ve Vitis AI repozitáři. Kromě toho bylo také třeba instalovat závislosti pro původní YOLOv7, stáhnout datovou sadu COCO a stáhnout váhy originálního YOLOv7 modelu.

Příkazy nutné pro vykonání všech těchto kroků lze nalézt v *README* souboru ve složce *yolov7* (v repozitáři této práce).

3.3.2 Příprava Alveo serveru

Příprava Alveo serveru je opět stejná jako při testování modelů pro Iris a CIFAR-10. Viz sekce 3.1.1.1 a 3.1.1.2. Navíc je třeba si jen stáhnout datovou sadu COCO pro testování.

3.3.3 Kvantizace modelu

Původní repozitář od AMD už obsahuje skripty pro kvantizaci, *train_qat.py* and *test_nndct.py*. Ty lze použít. Ovšem jak už bylo zmíněno, takto kvantizovaný YOLOv7 model, který lze od AMD i stáhnout už hotový, nelze zkompilovat pro DPUCAHX8H. To totiž nepodporuje aktivační funkci hard-swish, která zde aproximuje swish. Swish je použita ve všech konvolučních vrstvách originálního modelu YOLOv7.

Pro použití v DPUCAHX8H bylo rozhodnuto aproximovat swish pomocí funkce Leaky ReLU. Tuto aproximaci lze vidět i u některých jiných modelů z Model Zoo. Leaky ReLU je také použita v malé verzi YOLOv7 modelu pojmenované YOLOv7-Tiny. Kromě toho jsou možnosti výběru aktivační funkce DPUCAHX8H značně omezené a jiný výběr než Leaky ReLU v podstatě nepřipadá v úvahu. Záporný sklon Leaky ReLU byl zvolen jako 0.1015625 dle hodnoty 0.1 použité v YOLOv7-Tiny a toho, co podporuje DPU.

Změna byla provedena v konfiguracích ve složce cfg. Úpravou originálního *yolov7.yaml* byl vytvořen nový konfigurační soubor *yolov7-leaky.yaml*, kde byly nahrazeny aktivační funkce swish funkcí Leaky ReLU. Jako problematická se ukázala komponenta SPPCSPC, ve které nebylo možné změnit aktivační funkce přímo z konfiguračního souboru. Bylo nutné upravit moduly *common.py* a *yolo.py* ve složce models a přidat do nich komponentu SPPCSPC_for_DPU s aktivační funkcí změněnou na Leaky ReLU. V *yolov7-leaky.yaml*, poté byla místo SPPCSPC použita komponenta SPPCSPC_for_DPU.

Kromě aktivační funkce se při kompilaci ukázalo také, že DPUCAHX8H nepodporuje v max-pooling vrstvách pool větší než 8. Originální komponenta SPPCSPC používala pooly o velikosti 5, 9 a 13. V SPPCSPC_for_DPU to tedy bylo upraveno na 3, 5 a 7.

V úvodní fázi testování kvantizace byly testy prováděny následujícím postupem. Se změněnou konfigurací byla vždy provedena jedna epocha přeučení původního modelu (pomocí *train.py*). Tím byl vytvořen změněný PyTorch model. Na něm byla poté pomocí skriptu *test_nndct.py* provedena PTQ (pro konktrétní parametry příkazu viz původní README z Copyleft Model Zoo). Tím byl ve složce nndct vytvořen kvantizovaný model. Ten pak byl přesunut na Alveo server, kde byla ve Vitis AI kontejneru otestována kompilace. Tento postup ovšem vedl k velmi nízké výsledné přesnosti. mAP[0.5:0.95] byla jen kolem 36 %.

Proto byla poté raději použita metoda QAT. Zvolený postup byl nedělat žádné přeučení předem, ale provést ho až v rámci QAT. Pomocí skriptu *train_qat.py* byl proveden QAT a výsledný model byl potom exportován pomocí *test_nndct.py*, viz výpis 3.12. Tento postup vedl k lepším výsledkům, nicméně je nutno dodat, že trval velmi dlouho. QAT zabral až čtyřnásobek paměti GPU oproti klasickému trénování, bylo tedy třeba trénovat v dávkách menší velikosti. Byl také celkově více výpočetně náročný. Jedna epocha zabrala na jedné GPU A100 40GB přes dvě hodiny. Celkově bylo provedeno 100 epoch, což trvalo něco přes týden. Nicméně nejlepší výsledek byl dosažen už kolem 60. epochy. Logy z provedené kvantizace lze najít ve složce *quantization_logs*.

Při zpětném pohledu jsem si také všiml, že během prvních epoch byla mAP velice nízká. To nejspíš ukazuje, že se přenos „znalostí“ z originálního modelu příliš nepovedl. Pro příště by tedy stálo za pokus nejprve přeučit model normálně v plovoucí řádové čárce a až poté na něm spustit QAT.

Výsledný model dosáhl mAP[0.5:0.95] 46,3 %. Zde je třeba ještě zmínit, že ve skriptech *train.py*, *train_qat.py* a *test_nndct.py* jsou na několika místech vypsány různé mAP, které se

```
# QAT
python3 -u train_qat.py --workers 8 --device 0 --batch-size 16 \
    --data data/coco.yaml --img 640 640 --cfg cfg/training/yolov7-leaky.yaml \
    --weights pt_yolov7_3.5/float/yolov7.pt --name yolov7-leaky_qat \
    --hyp data/hyp.scratch.p5_qat.yaml
# export modelu
python3 -u test_nndct.py --data data/coco.yaml --img 640 --batch 1 --conf 0.001 \
    --iou 0.65 --device 0 --weights runs/train/yolov7-leaky_qat/weights/best.pt \
    --name yolov7-leaky_qat_test_best --quant_mode test --nndct_qat
```

Výpis kódu 3.12 Spuštění QAT pro YOLOv7 model

navzájem liší. Je to způsobeno pravděpodobně různými použitými parametry během měření. V této práci byla brána jako určující mAP změřená nástrojem pycocotools na konci exportu pomocí *test_nndct.py*. Tak tomu bylo i v původním Copyleft Model Zoo repozitáři. Toto číslo zároveň je opravdu téměř stejné jako mAP naměřená pomocí pycocotools nad skutečnými výsledky z Alveo U55C.

Pro originální YOLOv7 model v plovoucí řádové čárce jsem naměřil mAP[0.5:0.95] 51 %. Copyleft Model Zoo repozitář pak říká, že jejich kvantizovaný model (QAT) s hard-swish funkcemi dosáhl 47,9 %. Naměřených 46,3 % je tedy vzhledem k provedeným změnám a aproximacím docela dobrý výsledek.

3.3.4 Kompilace modelu

Kompilace modelu byla provedena ve Vitis AI kontejneru na Alveo serveru. Proces je obdobný jako při kompilaci modelů z TensorFlow v předchozích částech práce. Za zmínku ale stojí, že výstupem kvantizace PyTorch modelu je už soubor s příponou xmodel. Ten tedy už obsahuje model ve formátu XIR. Nicméně tento soubor ještě není připraven pro spuštění na DPU. Stále musí projít obvyklým procesem kompilace. Tentokrát je k tomu ale použit nástroj vai_c_xir. Ten má však prakticky totožné rozhraní jako vai_c_tensorflow2. Kompilací vznikne opět xmodel soubor, tentokrát už ale spustitelný na DPU.

3.3.5 Aplikace využívající akcelerovaný model

Jakmile je k dispozici zkompilovaný model YOLOv7 je možné přistoupit k implementaci samotné aplikace využívající tento model.

Návrh a obecné rozdělení aplikace do modulů bylo stručně popsáno v sekci 2.3.2 a v diagramu na obrázku 2.5. Zdrojové kódy aplikace lze v repozitáři najít ve složce *yolov7_dpu_deployment*.

3.3.5.1 yolov7_dpu.py a yolov7_dpu_multiprocess.py

Základem celé aplikace je třída *Yolov7Dpu*. Instance této třídy si během inicializace vytvoří vlastní DPU Runner objekt a alokuje pro něj vstupní a výstupní buffery. Potom lze objekt ovládat pomocí třech metod: `preprocess_and_add_image_to_input_buffer()`, `run_batch()` a `postprocess_and_get_results()`.

První metoda přijme RGB obrázek jako Numpy pole a provede změnu jeho velikosti, tak aby odpovídal vstupnímu formátu 640×640 px. Poměr stran je zachován, zbytek je doplněn nulami. Poté jsou všechny hodnoty přeškálovány, aby odpovídaly vstupnímu formátu modelu, a vloženy do vstupního bufferu.

Druhá metoda spustí akceleraci na DPU. Volání této metody je blokující a čeká na dokončení zpracování dat.

Třetí metoda z této třídy je o něco složitější. Provádí post-processing YOLOv7 modelu, který byl stručně popsaný v sekci 1.1.2.1. Jde o to, že na výstupu modelu je sada *anchor boxů*, rozmístěných v mřížce po obrázku. Každý reprezentuje potenciální objekt. Síť pro každý anchor box vygeneruje pravděpodobnost (konfidenci), že v něm je skutečně objekt, pravděpodobnosti příslušnosti ke kategorii objektu a korekce pozice a rozměrů bounding boxu detekovaného objektu vůči původnímu anchor boxu. Souřadnice nejsou v pixelech ale v bodech původní mřížky s anchor boxy. Všechna data je navíc třeba ještě prvně zpracovat funkcí sigmoida. V rámci post-processingu je tedy nutné všechna data převést na bounding boxy skutečně detekovaných objektů a ponechat jen ty s konfidencí vyšší než zadaný konfidenční práh. Na výsledných bounding boxech je poté ještě provedena non-maximum suppresion. Kód byl inspirován Python kódem z originálního YOLOv7 repozitáře (používajícím PyTorch) i C++ kódem z Vitis AI repozitáře. Nicméně výsledná implementace je od obou lehce odlišná, napsaná v Pythonu, ale využívající pouze knihovnu Numpy. Byl kladen důraz na relativně vysokou rychlost zpracování. Post-processing byl měl být tedy relativně svižný, ale stále jde jen o kód v Pythonu s co největším využitím Numpy. Dobře napsaná čistá C++ implementace by byla určitě rychlejší.

Nad třídou *Yolov7Dpu* potom staví *Yolov7DpuMultiprocess*. Ta používá Python knihovnu multiprocessing. Během inicializace vytvoří daný počet procesů, každý se svou instancí *Yolov7Dpu* a tím pádem svým objektem DPU Runner. Pokud jsou tedy na serveru 3 karty Alveo U55C a do každé se nahrají 3 DPU jednotky, tak pro jejich plné vytížení je třeba v rámci *Yolov7DpuMultiprocess* vytvořit minimálně 9 pracovních procesů. Data do procesů a z nich jsou předávány pomocí front z knihovny multiprocessing.

S tím se pojí ještě jedno zajímavé pozorování ohledně paralelního zpracování v Pythonu. Pro vybírání dat z front jsou použita blokující volání. Ta by v případě prázdné fronty teoreticky téměř vůbec nemusela zatěžovat procesor. Ovšem během testování se ukázalo, že to není pravda. V dokumentaci není proces čekání přesně popsaný. Nicméně i v případě, že jsou fronty prázdné a jen se čeká na data, tak všechny procesy relativně výrazně vytěžují procesor. Připomíná to aktivní čekání.

3.3.5.2 yolov7_dpu_test.py a yolov7_coco_eval.py

Modul *yolov7_dpu_test.py* slouží jako lokální uživatelské rozhraní pro *Yolov7DpuMultiprocess*. Rozhraní je implementováno skrze příkazovou řádku pomocí Python knihovny argparse. Lze skrze něj specifikovat cestu ke zkompilovanému modelu, vstupní obrázky pro inferenci, formát výstupu, počet pracovních procesů nebo konfidenční a IoU práh pro post-processing. Popis všech možností lze získat pomocí přepínače `--help`.

Modul nejprve provede načtení všech obrázků z disku (pomocí OpenCV) a jejich převedení na Numpy pole v RGB formátu. Poté je provedena samotná inference skrze *Yolov7DpuMultiprocess*. Pro zobrazení jejího průběhu je použit progress bar z knihovny tqdm. Je změřena propustnost i průměrná doba pre-processingu, zpracování na DPU i post-postprocessingu. Výsledná data jsou poté dle specifikace uživatelem vypsána na standardní výstup, zakreslena jako bounding boxy do původních obrázků, nebo uložena v JSON souboru kompatibilním s pycocotools. Skript *yolov7_coco_eval.py* poté slouží pro změření mAP výsledných dat pomocí pycocotools.

Ukázku použití *yolov7_dpu_test.py* a *yolov7_coco_eval.py* lze vidět ve výpisu 3.13.

```
python yolov7_dpu_test.py \
    -x ../yolov7_train_quant/compiled_yolov7-leaky_qat/yolov7-leaky.xmodel \
    -l ../yolov7_train_quant/coco/val2017.txt --prefix-image-list -c 9 \
    --conf-thres 0.001 --iou-thres 0.65 --results-to-json
python yolov7_coco_eval.py ../yolov7_train_quant/coco/ predictions.json
```

Výpis kódu 3.13 Použití *yolov7_dpu_test.py* a *yolov7_coco_eval.py*

3.3.5.3 yolov7_dpu_server.py

Modul *yolov7_dpu_server.py* implementuje TCP server kolem *Yolov7DpuMultiprocess*. Uživatelské rozhraní je opět vytvořeno pomocí knihovny argparse. Jeho dokumentaci vypíše přepínač `--help`.

Síťová komunikace je napsána pomocí knihovny socket. Server obsahuje jedno vlákno pro přijímání nových spojení, poté jedno vlákno pro každé otevřené spojení sloužící pro příjem dat, jedno společné vlákno pro odesílání zpracovaných dat a jedno vlákno pro výpis aktuálních informací o otevřených spojeních a čekajících datech.

Formát přijímaných zpráv je: id obrázku (2 byty), výška obrázku (2 byty), šířka obrázku (2 byty), data obrázku (výška × šířka × 3 bytů). Formát odesílaných zpráv je: id obrázku (2 byty), počet detekovaných objektů (2 byty) a data o detekovaných objektech (počet objektů × 6 × 4 bytů). Výsledky jsou posílány v 32-bitovém datovém typu float.

3.3.5.4 yolov7_dpu_client.py

Modul *yolov7_dpu_client.py* je obdobou *yolov7_dpu_test.py*. Jejich uživatelské rozhraní je téměř totožné. *yolov7_dpu_client.py* ale nepoužívá *Yolov7DpuMultiprocess* přímo, nýbrž s ním komunikuje po síti skrze *yolov7_dpu_server.py*. Klient nejprve zase v hlavním vlákně načte všechny obrázky, poté vytvoří nové vlákno, které obrázky postupně posílá na server. Hlavní vlákno mezitím přijímá výsledky a pomocí tqdm zobrazuje aktuální stav. Rychlost posílání obrázků na server lze nastavit přepínačem `--images-per-second`.

Některé společné části modulů *yolov7_dpu_client.py* a *yolov7_dpu_test.py* týkající se načítání vstupů a ukládání výsledků byly implementovány v modulu *yolov7_io_utils.py*, aby se předešlo zbytečným duplicitám. Pro použití *yolov7_dpu_client.py* je tedy třeba mít dostupný i tento modul.

3.3.5.5 yolov7_dpu_client_nao.py

Modul *yolov7_dpu_client_nao.py* je připraven pro použití na robotech NAO pomocí python3-qi. Neobsahuje argparse rozhraní jako ostatní moduly. Konfiguraci lze provést nastavením konstant na začátku funkce main.

Modul v hlavním vlákně čte obrázky z kamery robota a pravidelně je posílá na server. Hlavní vlákno také kreslí naposledy přijaté bounding boxy do aktuálního obrázku a zobrazuje je pomocí funkce `imshow()` z knihovny OpenCV. Vedlejší vlákno mezitím přijímá aktuální výsledky ze serveru.

K robotům je třeba se připojovat skrze server livsgpu01 pomocí programu python3-qi. Aby fungovalo zobrazování obrazu pomocí `imshow()` je nutné mít na livsgpu01 samozřejmě nějaké GUI. Je tedy vhodné se k livsgpu01 připojit skrze VNC.

Příklady použití *yolov7_dpu_server.py*, *yolov7_dpu_client.py* a *yolov7_dpu_client_nao.py* lze najít v README ve složce yolov7 v repozitáři této práce.

Kapitola 4
Výsledky experimentů

4.1 Modely pro datovou sadu CIFAR-10

Aby bylo možné výsledky akcelerace pomocí karet Alveo U55C s něčím alespoň přibližně srovnat, tak jsou nejdříve prezentovány výsledky původních modelů v plovoucí řádové čárce akcelerovaných pomocí GPU. Poté jsou uvedeny výsledky akcelerace pomocí Vitis AI a HLS4ML.

4.1.1 Akcelerace modelů na GPU

Testování na GPU proběhlo pomocí skriptů ze sekce 3.2.5. Celé výstupy skriptů lze najít v repozitáři ve složce *deployment_gpu_output*. Souhrn výsledků je v tabulce 4.1.

Testování proběhlo na jedné kartě GPU A100 40GB. Modely byly testovány v jejich původní podobě v plovoucí řádové čárce.

Přesnost byla měřena na validační datové sadě z CIFAR-10. Nejvyšší přesnost má model big_4M8, pro který byly speciálně vyladěny třeba míry dropout na jednotlivých vrstvách. Ostatním modelům taková pozornost věnována nebyla.

Spotřeba byla měřena při zpracování 100000 obrázků, jakožto průměr z měření prováděných co jednu sekundu.

Model	Přesnost	Spotřeba	Čas zpracování		
			1 obrázek	10000 obr.	100000 obr.
mega_633M	65,11 %	226,62 W	0,75 s	3,55 s	25,42 s (254 μs/obr.)
biggest_156M	84,36 %	165,25 W	0,71 s	2,23 s	12,23 s (122 μs/obr.)
bigger_39M	87,92 %	71,59 W	0,71 s	2,06 s	10,17 s (102 μs/obr.)
big_4M8	88,53 %	40,10 W	0,69 s	2,04 s	9,29 s (93 μs/obr.)
medium_701k	84,55 %	37,14 W	0,75 s	1,85 s	9,38 s (94 μs/obr.)
small_191k	81,36 %	35,25 W	0,68 s	1,99 s	9,47 s (95 μs/obr.)
tiny_57k	77,05 %	36,03 W	0,68 s	1,90 s	9,48 s (95 μs/obr.)
tinier_24k	69,27 %	35,06 W	0,72 s	1,97 s	9,57 s (96 μs/obr.)
tinier_15k	67,18 %	35,83 W	0,67 s	1,81 s	9,11 s (91 μs/obr.)
tinier_8k	62,31 %	35,25 W	0,68 s	1,94 s	9,25 s (93 μs/obr.)
tiniest_2k5	43,79 %	34,22 W	0,83 s	1,87 s	9,57 s (96 μs/obr.)

Tabulka 4.1 Výsledky GPU akcelerace modelů pro CIFAR-10

Model	Přesnost	Spotřeba	Čas zpracování		
			1 obrázek	10000 obr.	100000 obr.
mega_633M	DPUCAHX8H nepodporuje dost široké plně propojené vrstvy				
biggest_156M	DPUCAHX8H nepodporuje dost široké plně propojené vrstvy				
bigger_39M	88,20 %	52,87 W	0,0046 s	2,78 s	27,82 s (278 μs/obr.)
big_4M8	88,56 %	53,39 W	0,0021 s	0,55 s	5,23 s (52 μs/obr.)
medium_701k	84,48 %	30,43 W	0,0025 s	0,54 s	5,16 s (52 μs/obr.)
small_191k	81,04 %	28,46 W	0,0022 s	0,54 s	5,19 s (52 μs/obr.)
tiny_57k	76,87 %	27,27 W	0,0029 s	0,55 s	5,18 s (52 μs/obr.)
tinier_24k	68,99 %	26,08 W	0,0024 s	0,55 s	5,18 s (52 μs/obr.)
tinier_15k	66,73 %	26,05 W	0,0023 s	0,55 s	5,14 s (51 μs/obr.)
tinier_8k	62,32 %	26,03 W	0,0020 s	0,55 s	5,17 s (52 μs/obr.)
tiniest_2k5	43,62 %	25,93 W	0,0020 s	0,54 s	5,18 s (52 μs/obr.)

Tabulka 4.2 Výsledky Vitis AI (DPU) akcelerace modelů pro CIFAR-10

Čas zpracování byl změřen jako čas vykonávání funkce predict (z Keras). Do času tedy není zahrnuto třeba načítání modelu z disku, které pro velké modely není uplně zanedbatelné. Pro model big_4M8 trvalo 2,6 sekundy, pro mega_633M už přibližně 210 sekund (viz *deployment_gpu_output*). Na druhou stranu toto stačí provést pouze jednou před zahájením akcelerace.

Zpracováním jednoho obrázku se myslí zavolání funkce predict s pouze jedním obrázkem. Představuje to tedy minimální režii nutnou pro rozběhnutí akcelerace.

Z časů zpracování a spotřeb je vidět, že modely menší než bigger_39M kartu v podstatě nevytížily. V čase zpracování pravděpodobně převažovala režie, například přenos obrázků do karty. Kartu více zatížily pouze dva až tři největší modely.

4.1.2 Vitis AI

Vitis AI akcelerace byla provedena pomocí skriptů popsaných v sekci 3.2.3.2. Podrobné výsledky lze dohledat ve výstupech skriptů, které jsou v repozitáři ve složce *deployment_dpu_output*. Souhrn naměřených dat je v tabulce 4.2.

Měření byla provedena na jedné kartě Alveo U55C. Použity byly tři DPU Runner objekty pro vytížení všech dostupných DPU jednotek. Jako DPU byla použita naše varianta DPUCAHX8H pro 300 MHz.

Pro testované modely byla pro kvantizaci použita pouze metoda PTQ. Při porovnání tabulek 4.1 a 4.2 je ale vidět, že pokles přesnosti byl na těchto modelech i tak nízký. Nikdy nepřekonal půl procenta. V některých případech se přesnost dokonce lehce zvýšila. Lze tak říct, že pro tyto stále relativně mělké modely funguje Vitis AI Quantizer velice dobře.

Do časů zpracování uvedených v tabulce 4.2 je započítáno zpracování na DPU a výpočet funkce softmax na CPU. V časech není zahrnuto načítání modelu z disku ani inicializace objektu DPU Runner (a s tím spojené ověření/nahrání daného xclbin souboru). Čas včetně inicializace objektu DPU Runner, čas bez výpočtu softmax funkce či čas načítání modelu z disku lze najít ve výstupech ve složce *deployment_dpu_output*. Inicializace třech objektů DPU Runner obvykle zabrala kolem 6 sekund. Pro menší počet méně. Výpočet funkce softmax na CPU zabral pro 100000 obrázků většinou kolem půl sekundy.

V naměřených časech zpracování je vidět stejný jev jako u akcelerace na GPU. Modely menší než bigger_39M kartu příliš nezatížily. Časy spojené s minimální režií nutnou při zpracování dat jsou zde ale o něco nižší (za předpokladu, že nepočítáme prvotní inicializaci objektu DPU Runner). To může být způsobeno mnoha faktory od architektury celého systému, přes to, že zde je použit kvantizovaný model a pracuje se jen s kvantizovanými daty, která jsou objemově menší,

po to, že funkce predict v Keras může ještě provádět nějaký typ inicializace, který je zde už zahrnut v rámci inicializace objektu DPU Runner.

Nicméně podíváme-li se na časy zpracování pro modely big_4M8 a větší, kde už dochází k vyššímu zatížení karet, tak naměřená data ve prospěch Vitis AI už příliš nehovoří. Dle spotřeby dochází k vysokému zatížení karty Alveo U55C už u modelu big_4M8 a s modelem bigger_39M začínají růst časy zpracování. Zatímco u GPU A100 40GB dojde k plnému vytížení až s modelem mega_633M. Grafická karta zvládne i tímto největším modelem zpracovat 100000 obrázků rychleji než to DPU zvládne s modelem bigger_39M. DPUCAHX8H navíc nepodporuje plně propojené vrstvy o dostatečné šířce pro implementaci modelů větších než bigger_39M. Pokud provedeme porovnání pro model bigger_39M, který se podařilo implementovat i na Alveo kartě, tak lze vidět, že grafická karta vychází lepší v porovnání času zpracování i v porovnání celkové spotřebované elektrické energie. Ve prospěch grafické karty dále hovoří i to, že implementované modely na ní byly spouštěny v původní verzi v 32-bitové plovoucí řádové čárce, která je výpočetně náročnější. Přitom grafické karty dnes už podporují i akceleraci kvantizovaných modelů.

Alveo karta je ale zase o něco levnější. Rychlým pohledem do internetových obchodů se ukazuje, že Alveo U55C lze v dnešní době pravděpodobně pořídit za dvakrát až třikrát nižší cenu než GPU A100 40GB.

Kromě toho jsou zde implementované modely navíc specifické svým škálováním do šířky. Takže větší testované modely jsou především hodně široké (mají hodně parametrů v rámci jedné vrstvy) a nejsou příliš hluboké (nemají příliš mnoho vrstev). Naměřená data tedy nemusí být ve všech ohledech vypovídající.

Celé toto srovnání mezi Alveo U55C a GPU A100 40GB je provedeno spíše jen jako orientační a není příliš důsledné. To ale nebylo ani cílem této práce.

4.1.3 HLS4ML

Syntéza kernelů pomocí HLS4ML i jejich spouštění bylo testováno pomocí skriptů popsaných v sekci 3.2.4. Podrobný výstup ze skriptů pro všechny provedené pokusy lze najít v repozitáři ve složkách *deployment_hls4ml_output*, *synthesis_outputs* a *simulation_outputs*. Přehled provedených pokusů o syntézu a jejich výsledků lze vidět v tabulce 4.3.

Jak už bylo popsáno, tak během pokusů o syntézu se objevilo několik problémů. Ty vedly k tomu, že se nakonec podařilo syntetizovat jen velmi malé modely. Což je potažmo hlavní důvod, proč v sadě testovaných konvolučních sítí vytvořených pro CIFAR-10 je tolik malých modelů.

U modelu big_4M8 se nepodařilo provést ani HLS. Byly provedeny dva pokusy. Jeden s datovým typem ap_fixed<24,8> a jeden kombinací různých datových typů na různých vrstvách modelu. Oba selhaly kvůli příliš velkému množství potřebné RAM.

Další pokusy už byly prováděny na podstatně menších modelech (od tiniest_2k5 po tiny_57k). U nich proběhla HLS v pořádku, ale objevil se problém během fáze logic routing. Konkrétně *routing congestion* (nedostatek potřebných cest v FPGA). Byla provedena řada pokusů o syntézu s různými parametry HLS4ML modelu. Změna parametru *strategy* z *resource* na *latency* vedla pouze k jiné chybě. To není příliš překvapující. *latency* není pro větší modely doporučená. Snížení cílové frekvence kernelu také nepomohlo. Problém s routing congestion to spíše zhoršilo. Změnou, která měla pozitivní vliv, bylo výrazné zvýšení parametru *reuse factor*. Jenže to vede samozřejmě ke snížení rychlosti kernelu. Kernely, které se implementovat podařilo proto nebyly příliš rychlé.

Pomohlo také použití šablon z Vitis backendu (dědením třídy VivadoAcceleratorWriter z VitisWriter namísto VivadoWriter) modifikovaných pro Vitis HLS. Podařilo se díky tomu kromě modelů tiniest_2k5 a tinier_8k syntetizovat i model tinier_15k.

U modelů, které se podařilo syntetizovat byl změřen čas zpracování obrázků. V tomto případě byla měřena celá funkce predict implementovaná ve skriptu axi_stream_driver.py z HLS4ML. To znamená, že naměřené časy zahrnují i inicializaci kernelu. Nicméně před měřením byla akcelerace vždy provedena ještě jednou, aby se neměřil celý čas nového nahrání kernelu do karty.

Model (a označení pokusu)	Vitis Writer	Parametry HLS4ML modelu				Výsledky / Chyba syntézy			
		Precision	Reuse Factor	Strategy	Frekvence	Přesnost	Čas zpracování		
							1 obr.	10000 o.	100000 o.
big_4M8_1	Ne	ap_fixed<24,8>	1024	Resource	100 MHz	Nedostatek paměti během HLS			
big_4M8_2	Ne	ap_fixed<24,8> /ap_fixed<12,2>	1024	Resource	100 MHz	Nedostatek paměti během HLS			
tiny_57k_1	Ne	ap_fixed<24,8>	1024	Resource	100 MHz	Routing congestion (effective congestion level 7)			
tiny_57k_2	Ne	ap_fixed<24,8>	64	Resource	100 MHz	Routing congestion (effective congestion level 6)			
tinier_24k_1	Ne	ap_fixed<24,8>	1024	Resource	100 MHz	Routing congestion (effective congestion level 5)			
tinier_24k_2	Ne	ap_fixed<24,8>	1024	Resource	10 MHz	Routing congestion (effective congestion level 6)			
tinier_24k_3	Ne	ap_fixed<24,8>	1024	Latency	100 MHz	Jiná chyba routing (SLL Assignment failed.)			
tinier_15k_1	Ne	ap_fixed<24,8>	1024	Resource	100 MHz	Routing congestion (effective congestion level 6)			
tinier_8k_1	Ne	ap_fixed<24,8>	64	Resource	100 MHz	Routing congestion (effective congestion level 5)			
tinier_8k_2	Ne	ap_fixed<24,8>	128	Resource	100 MHz	Routing congestion (effective congestion level 5)			
tinier_8k_3	Ne	ap_fixed<24,8>	256	Resource	100 MHz	Routing congestion (effective congestion level 5)			
tinier_8k_4	Ne	ap_fixed<24,8>	1024	Resource	100 MHz	62,29 %	0,51 s	10,48 s	100,11 s
tiniest_2k5_1	Ne	ap_fixed<24,8>	256	Resource	100 MHz	43,81 %	0,51 s	4,25 s	38,13 s
tiniest_2k5_2	Ne	ap_fixed<24,8>	16	Resource	100 MHz	43,81 %	0,54 s	1,58 s	12,02 s
tiny_57k_vitis_1	Ano	ap_fixed<24,8>	128	Resource	100 MHz	Routing congestion (effective congestion level 6)			
tinier_24k_vitis_1	Ano	ap_fixed<24,8>	1024	Resource	100 MHz	Routing congestion (effective congestion level 6)			
tinier_15k_vitis_1	Ano	ap_fixed<24,8>	128	Resource	100 MHz	Routing congestion (effective congestion level 5)			
tinier_15k_vitis_2	Ano	ap_fixed<24,8>	1024	Resource	100 MHz	67,12 %	0,83 s	14,33 s	137,20 s
tinier_8k_vitis_1	Ano	ap_fixed<24,8>	128	Resource	100 MHz	62,29 %	0,78 s	7,37 s	68,50 s

Tabulka 4.3 Výsledky HLS4ML akcelerace modelů pro CIFAR-10

Úspěšně se podařilo implementovat jen několik malých modelů. A to s vysokým reuse factor, potažmo tedy i vysokým časem zpracování.

Hlavním problémem byla už zmíněná *routing congestion*. V tabulce 4.3 lze vidět, že největší model, který se nakonec podařilo syntetizovat je tinier_15k a to jen za cenu vysokého reuse factor. Nicméně, i pokud by se podařilo tento problém například nějakou změnou architektury vyřešit, tak u větších modelů je stále problém s příliš velkou pamětí potřebnou během HLS a nakonec by pravděpodobně nastal i problém s umístěním všech vah do FPGA. HLS4ML totiž umožňuje váhy umístit jen přímo do FPGA čipu. Jenže tam se jich vejde jen omezené množství. Když sečteme distribuovanou RAM, BRAM i URAM v XCU55C, tak dostaneme jen 377,6 Mb. I to už může být pro větší modely potenciálně omezující. HLS4ML se tak v současné podobě nejeví jako příliš vhodný nástroj pro akceleraci větších modelů.

Jako cíl HLS4ML je i oficiálně prezentována hlavně akcelerace malých modelů pro extrémně nízké latence v řádu mikrosekund. Jenže tohle využití na PCIe kartě nepřipadá v úvahu. Nehledě na to, že karta bude nejčastěji umístěná někde v cloudovém serveru a komunikovat po síti. V takovém systému už z principu nelze dosáhnout latencí pro které bylo HLS4ML původně navrženo.

4.2 Prototypová aplikace pro detekci objektů v obraze

S prototypovou aplikací bylo provedeno několik testů pro měření její propustnosti, přesnosti a latence. Všechny testy byly prováděny na validační datové sadě z COCO obsahující 5000 obrázků. Seznam těchto obrázků je v COCO pojmenován jako *val2017.txt*.

4.2.1 Lokální testy

Nejprve byly provedeny lokální testy pomocí *yolov7_dpu_test.py*. Celé výstupy lze nalézt v repozitáři ve složce *deployment_outputs*. Shrnuté výsledky lze vidět v tabulce 4.4.

Nastavení				Naměřená data		
Počet karet	Počet procesů	Konfidenční práh	IoU práh	Propustnost	Průměrný čas zpracování dávky	
					Celkem	Pouze DPU
1	1	0,25	0,65	16,01 fps	0,18534 s	0,12957 s
1	2	0,25	0,65	26,21 fps	0,22656 s	0,17117 s
1	2	0,001	0,65	19,82 fps	0,29802 s	0,14814 s
1	3	0,25	0,65	26,21 fps	0,34025 s	0,28658 s
1	3	0,001	0,65	25,17 fps	0,35017 s	0,20214 s
1	6	0,25	0,65	26,23 fps	0,68154 s	0,62912 s
3	3	0,25	0,65	48,84 fps	0,17860 s	0,12731 s
3	6	0,25	0,65	74,62 fps	0,22641 s	0,17598 s
3	9	0,25	0,65	72,55 fps	0,33992 s	0,28870 s

Tabulka 4.4 Výsledky testů pomocí *yolov7_dpu_test.py*

Vyzkoušeny byly konfigurace s různým počtem karet a různým počtem pracovních procesů (tedy i různým počtem objektů DPU Runner). Počet dostupných karet lze omezit pomocí proměnné XLNX_ENABLE_DEVICES. Počet vytvořených pracovních procesů lze v *yolov7_dpu_test.py* nastavit přepínačem `--processes-count` (`-c`).

Je měřena propustnost a poté průměrný čas zpracování jedné dávky (toto měření je provedeno v rámci pracovního procesu ve třídě *Yolov7DpuMultiprocess*). Celkovým časem zpracování dávky se myslí pre-processing obrázku, zpracování obrázku na DPU skrze VART a post-processing

výsledných dat. Čas zpracování na DPU je měřen od spuštění zpracování skrze VART do jeho dokončení. Tento čas tedy nemusí nutně vyjadřovat pouze zpracování na DPU. Naopak může zahrnovat například čekání na dostupnou DPU jednotku (avšak od toho nás VART odstíní).

Nejvyšší dosažená propustnost s jednou kartou byla kolem 26 obrázků za sekundu a s třemi kartami kolem 75 obrázků za sekundu. Zajímavé zjištění je, že s počtem DPU Runner objektů se propustnost neškáluje lineárně. Při použití jedné karty a jednoho objektu DPU Runner je propustnost 16 obrázků za sekundu, při použití dvou objektů stoupne na 26 snímků za sekundu. Při dalším zvýšení počtu DPU Runner objektů už zůstává konstantní a to i přesto, že na jedné kartě jsou dostupné ne dvě, ale tři DPU jednotky. Při použití 3 karet se rychlost zvedá téměř na trojnásobek, pro počet procesů zde ale platí podobné pravidlo jako u jedné karty. Nejlépe se chovalo 6 procesů (přestože bylo 9 dostupných DPU jednotek). Nejspíš se úzkým hrdlem stává nějaká režie spojená s komunikací s kartami.

Kromě toho lze v tabulce také vidět, že na propustnost může mít vliv i zvolený konfidenční práh. Při jeho nastavení na nízkou hodnotu je ve výsledcích více bounding boxů a post-processing díky tomu může trvat podstatně déle. V případě 2 procesů a konfidenčního prahu 0,001 doba nutná pro pre-processing a post-processing přesáhla dobu samotného zpracování na DPU.

Měření přesnosti je provedeno skriptem *yolov7_coco_eval.py*, který interně používá nástroj pycocotools. Tomu je třeba předat všechny výsledky včetně těch s nízkou konfidencí. Proto je třeba nastavit nízký konfideční práh, například 0,001. S takto nastaveným konfidenčním prahem byla naměřena mAP[0,5:0,95] 46,4 %. Celý výstup z pycocotools lze také najít v repozitáři ve složce *deployment_outputs*.

Elektrický odběr při použití jedné karty a třech procesů byl kolem 44 W a nepřesáhl 46 W.

4.2.2 Testy celkové latence na straně klienta

Kromě lokálních testů byla provedena i měření včetně síťové komunikace. Na Alveo serveru byl ve Vitis AI kontejneru spuštěn *yolov7_dpu_server.py*. Byly použity všechny 3 dostupné Alveo karty a 6 pracovních procesů. Konfidenční práh byl nastaven na 0,25 a IoU práh na 0,65. Samotné měření skriptem *yolov7_dpu_client.py* bylo provedeno ze serveru livsgpu01. Shrnuté výsledky lze najít v tabulce 4.5. Celé výstupy jsou vloženy v repozitáři.

Přibližné nastavení rychlsoti	**Skutečná rychlost zpracování**	**Celková latence**		
		Průměr	**Min**	**Max**
30 fps	28,26 fps	0,2442 s	0,1590 s	0,3226 s
50 fps	45,04 fps	0,2588 s	0,1732 s	0,3339 s
70 fps	65,03 fps	0,2617 s	0,1774 s	0,3220 s
80 fps	73,43 fps	0,2738 s	0,1672 s	0,3732 s
90 fps	72,84 fps	1,5943 s	0,1716 s	3,2397 s

Tabulka 4.5 Celkové latence naměřené při použití *yolov7_client_test.py* na serveru livsgpu01

Testy byly provedeny pro různé rychlosti odesílání dat. Rychlost lze v *yolov7_dpu_client.py* nastavit přepínačem `--images-per-second`. Ovšem je to nastavení přibližné. Výpočet nebere v úvahu to, že samotné posílání dat také trvá nějaký čas. Nicméně pro toto měření je to dostačující. Změřená latence zahrnuje přípravu zprávy s obrázkem, její odeslání, příjem odpovědi a zpracování přijaté zprávy. V naměřených datech lze vidět, že i při použití síťové komunikace se celková latence drží kolem 0,25 sekundy v průměrném případě a 0,33 sekundy ve špatném případě. Takto lze dosáhnout v podstatě plných 70 snímků za sekundu, teprve při překročení této hranice dojde k zahlcení serveru, začne se plnit vstupní fronta a latence začne růst. Ovšem je třeba mít na paměti, že toto testování bylo provedeno jen ze serveru livsgpu01. Síťová komunikace je v tomto případě rychlá a spolehlivá a celkovou latenci tak zvedá minimálně.

Diskuse

Experimenty prokázaly, že jak HLS4ML, tak Vitis AI lze použít pro spuštění neuronových sítí na Alveo U55C. Nicméně oba nástroje mají velkou řadu omezení.

V rámci testování HLS4ML bylo zjištěno, že nástroj například nepodporuje max pooling vrstvy s kroky jiné velikosti než je velikost poolu. Avšak jako hlavní překážka jeho použití se ukázaly být problémy s *routing congestion* během vytváření kernelu. U větších modelů se tyto problémy nepodařilo překonat. Naopak, u velkých modelů se ještě objevily problémy s velmi velkou vyžadovanou pamětí během HLS. Na Alveo kartě se kvůli *routing congestion* nakonec podařilo spustit jen velmi malé modely. Režie spojená s komunikací mezi kartou a CPU serveru tak pravděpodobně neguje jakékoliv zrychlení získané oproti implementaci přímo v CPU. Modely jsou na to prostě příliš malé. Původní cíl HLS4ML dosáhnout extrémně nízkých latencí pro malé modely v tomto typu systému nelze pořádně uplatnit. Výjimkou by mohl být pouze systém, kde by HLS4ML kernel byl součástí nějakého většího designu implementovaného celého přímo v FPGA. Každopádně HLS4ML se v současné podobě nejeví jako vhodná volba pro serverovou akceleraci neuronových sítí.

Vitis AI oproti tomu dokázal spustit i velké modely. Co se týče velikosti, tak se problémové ukázaly jen velmi široké plně propojené vrstvy. DPUCAHX8H určené pro Alveo U55C podporuje plně propojené vrstvy jen do počtu 4096 neuronů. Kromě toho má však DPUCAHX8H i další omezení na podobu akcelerovaného modelu. Nejsou například podporovány max pooling vrstvy, které mají *pool* větší než 8. Nejvíce striktní je potom výběr aktivačních funkcí. Jsou podporovány pouze ReLU, ReLU6 a Leaky ReLU.

Nicméně i přes tato omezení se podařilo vytvořit modely akcelerovatelné na DPUCAHX8H a použít je pro klasifikaci obrázků z datové sady CIFAR-10. Kromě toho se pro karty Alveo U55C podařilo upravit i model YOLOv7 a úspěšně ho použít v prototypové aplikaci. Vitis AI také umožňuje jednoduše akceleraci škálovat i na více Alveo karet nejednou. Jeho použití v praxi si lze tedy docela snadno představit.

DPU je ale v podstatě fixní procesorová jednotka. Nevyužívá tak možnosti flexibility FPGA úplně naplno. Naopak, jeho výkon je do velké míry omezen právě možnostmi a rychlostí FPGA čipu. Dle provedených testů na konvolučních sítích pro CIFAR-10 je výkon na Alveo U55C nižší než akcelerace původního modelu pomocí GPU A100 40GB. Proti použití čistých FPGA karet hovoří i to, že AMD v posledních verzích Vitis AI již karty jako Alveo U55C nepodporuje. Místo toho přišlo v nových platformám s tzv. AI Engines, což je dedikovaný hardware pro akceleraci některých operací z DPU. Nové DPU tak jsou v dynamickém FPGA regionu implementovány už jen částečně. Vitis AI se dnes tedy zřejmě od použití čistých FPGA karet spíše odklání a přechází k hybridu mezi dedikovaným hardwarem a FPGA.

Tato práce ukázala, že cloudová karta Alveo U55C za použití Vitis AI určitě je možnou alternativou pro akceleraci neuronových sítí. Nicméně příliš výhodným řešením dnes nejspíš není.

Závěr

Hlavním cílem práce bylo otestovat akceleraci neuronových sítí na cloudové kartě Alveo U55C pomocí frameworků HLS4ML a Vitis AI. Dalším cílem bylo vyvinout prototypovou aplikaci pro detekci objektů v obrázcích z kamer robotů.

Nejprve byla zprovozněna akcelerace malé sítě pro datovou sadu Iris pomocí frameworku Vitis AI. Při tom byla vyzkoušena základní funkcionalita jeho nástrojů pro kvantizaci a kompilaci sítě. Na základě příkladů z Vitis AI repozitáře pak byl vytvořen testovací skript pro spuštění modelu.

Poté byla otestována akcelerace této malé neuronové sítě také pomocí frameworku HLS4ML, konkrétně jeho backendu Vivado Accelerator. Při tom se během syntézy objevila řada problémů způsobených zejména použitou novější verzí nástrojů od AMD a chybějící oficiální podporou pro Alveo U55C. Ty se ale zásahem do kódu HLS4ML podařilo vyřešit a model úspěšně spustit.

Dále byl vytvořen a natrénován soubor jedenácti různě velkých CNN pro datovou sadu CIFAR-10. Na nich byla otestována akcelerace pomocí Vitis AI a HLS4ML. Pomocí Vitis AI se podařilo akcelerovat všechny modely až na dva. Ty obsahovaly velmi širokou plně propojenou vrstvu s větším počtem neuronů než podporuje DPUCAHX8H. Pomocí HLS4ML se podařilo syntetizovat jen malé modely. U středně velkých modelů se během syntézy objevily problémy s routing congestion, u těch větších potom i s příliš velkou pamětí nutnou pro C-syntézu. U největších modelů by byla omezující i velikost vah, které by se nevešly do vnitřní paměti FPGA čipu. HLS4ML se tak ukázalo jako nevhodné pro velké neuronové sítě. Pro implementované modely potom byl změřen čas nutný pro zpracování různého počtu obrázků a průměrná spotřeba karty během výpočtu. Obdobná měření byla provedena i pro akceleraci pomocí GPU skrze Keras.

Nakonec byl pomocí frameworku Vitis AI zprovozněn model YOLOv7. Ten musel být upraven, aby neobsahoval funkcionality nepodporované v DPUCAHX8H. V Pythonu byl potom naprogramován nutný preprocessing a postprocessing. Byla změřena propustnost, latence a přesnost modelu při akceleraci pomocí karty Alveo U55C. Kromě toho byl napsán prototypový TCP server. Ten je schopný přijímat obrázky, zpracovat je pomocí modelu na kartě a odeslat zpět informace o detekovaných objektech. K němu byl naprogramován i příslušný testovací klient pro roboty Nao. Ten pravidelně čte data z kamery, odesílá je na server a přijímá zpět odpovědi.

Práce tak ukázala použití cloudových karet Alveo U55C v oblasti strojového učení. Odhalila také některé problémy, na které při něm lze narazit.

Na práci by v budoucnu mohlo být navázáno vývojem aplikace pro roboty NAO, která by spolupracovala s implementovaným serverem a dále využívala získaná data o objektech z obrázků. Při dalším použití by také stálo za zvážení, zda server nepřepsat raději do jazyka C++. Python se totiž v několika ohledech neukázal jako ideální volba pro dlouhodobější použití (čekající procesy zatěžující CPU, nutnost použít procesy namísto vláken a nižší efektivita kódu pro post-processing dat z YOLO modelu).

Bibliografie

1. MISHRA, Mayank. Convolutional Neural Networks, Explained. *Towards Data Science* [online]. 2020 [cit. 2024-04-18]. Dostupné z: https://towardsdatascience.com/convolutional-neural-networks-explained-9cc5188c4939.
2. PANCHAL, Shubham. No, Kernels & Filters Are Not The Same. *Towards Data Science* [online]. 2021 [cit. 2024-04-18]. Dostupné z: https://towardsdatascience.com/no-kernels-filters-are-not-the-same-b230ec192ac9.
3. JAIN, Vandit. Everything you need to know about "Activation Functions" in Deep learning models. *Towards Data Science* [online]. 2019 [cit. 2024-04-18]. Dostupné z: https://towardsdatascience.com/everything-you-need-to-know-about-activation-functions-in-deep-learning-models-84ba9f82c253.
4. LECUN, Yann; BOTTOU, Léon; BENGIO, Yoshua; HAFFNER, Patrick. Gradient-Based Learning Applied to Document Recognition [online]. 1998 [cit. 2024-04-18]. Dostupné z: http://yann.lecun.com/exdb/publis/pdf/lecun-01a.pdf.
5. KRIZHEVSKY, Alex; SUTSKEVER, Ilya; HINTON, Geoffrey E. ImageNet Classification with Deep Convolutional Neural Networks [online]. 2012 [cit. 2024-04-18]. Dostupné z: https://proceedings.neurips.cc/paper_files/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.
6. WANG, Chien-Yao; BOCHKOVSKIY, Alexey; LIAO, Hong-Yuan Mark. YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors [online]. 2022 [cit. 2024-04-18]. Dostupné z: https://arxiv.org/abs/2207.02696.
7. RAHAMAN, Md. Faishal. The Current Trends of Object Detection Algorithms: A Review [online]. 2023 [cit. 2024-04-18]. Dostupné z: https://www.researchgate.net/publication/373392107_The_Current_Trends_of_Object_Detection_Algorithms_A_Review.
8. SOLAWETZ, Jacob. What is YOLOv7? A Complete Guide. [online]. 2024 [cit. 2024-04-18]. Dostupné z: https://blog.roboflow.com/yolov7-breakdown/.
9. HUGHES, Chris; CAMPS, Bernat Puig. YOLOv7: A Deep Dive into the Current State-of-the-Art for Object Detection. *Towards Data Science* [online]. 2022 [cit. 2024-04-18]. Dostupné z: https://towardsdatascience.com/yolov7-a-deep-dive-into-the-current-state-of-the-art-for-object-detection-ce3ffedeeaeb.
10. VEDOVELI, Henrique. Metrics Matter: A Deep Dive into Object Detection Evaluation [online]. 2023 [cit. 2024-04-19]. Dostupné z: https://medium.com/henriquevedoveli/metrics-matter-a-deep-dive-into-object-detection-evaluation-ef01385ec62.
11. GLENN-JOCHER; RIZWANMUNAWAR; ABIRAMI-VINA. Performance Metrics Deep Dive [online]. 2023 [cit. 2024-04-19]. Dostupné z: https://docs.ultralytics.com/guides/yolo-performance-metrics/.

12. FISHER, R. A. The use of multiple measurements in taxonomic problems. *Annals of Eugenics*. 1936, s. 179–188.

13. UNWIN, Antony; KLEINMAN, Kim. The Iris Data Set: In Search of the Source of Virginica. *Significance*. 2021, roč. 18, č. 6, s. 26–29. ISSN 1740-9705. Dostupné z DOI: 10.1111/1740-9713.01589.

14. KRIZHEVSKY, Alex. Learning Multiple Layers of Features from Tiny Images [online]. 2009, s. 32 [cit. 2024-04-18]. Dostupné z: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf.

15. KRIZHEVSKY, Alex; NAIR, Vinod; HINTON, Geoffrey. The CIFAR-10 dataset [online]. 2009 [cit. 2024-04-18]. Dostupné z: https://www.cs.toronto.edu/~kriz/cifar.html.

16. LIN, Tsung-Yi; MAIRE, Michael; BELONGIE, Serge; BOURDEV, Lubomir; GIRSHICK, Ross; HAYS, James; PERONA, Pietro; RAMANAN, Deva; DOLLÁR, Piotr; ZITNICK, C. Lawrence. *Microsoft COCO: Common Objects in Context* [online]. 2015. [cit. 2024-04-19]. Dostupné z arXiv: 1405.0312 [cs.CV].

17. GLENN-JOCHER; RIZWANMUNAWAR; LAUGHING-Q. COCO Dataset [online]. 2023 [cit. 2024-04-19]. Dostupné z: https://docs.ultralytics.com/datasets/detect/coco/.

18. MADHAVAN, Samaya; AHMED, Sidra; RAO, Vinay; JOHN, Anto. Compare deep learning frameworks [online]. 2021 [cit. 2024-04-18]. Dostupné z: https://developer.ibm.com/articles/compare-deep-learning-frameworks/.

19. LIANG, Tailin; GLOSSNER, John; WANG, Lei; SHI, Shaobo; ZHANG, Xiaotong. Pruning and Quantization for Deep Neural Network Acceleration: A Survey [online]. 2021 [cit. 2024-04-22]. Dostupné z arXiv: 2101.09671 [cs.CV].

20. ADVANCED MICRO DEVICES, Inc. Alveo U55C Data Center Accelerator Cards Data Sheet (DS978) [online]. 1.3. vyd. 2023 [cit. 2024-04-19]. Dostupné z: https://docs.amd.com/r/en-US/ds978-u55c.

21. ADVANCED MICRO DEVICES, Inc. Vitis Unified Software Platform Documentation: Application Acceleration Development (UG1393) [online]. 2023.2. vyd. 2023 [cit. 2024-04-19]. Dostupné z: https://docs.amd.com/r/en-US/ug1393-vitis-application-acceleration.

22. ADVANCED MICRO DEVICES, Inc. Vitis High-Level Synthesis User Guide (UG1399) [online]. 2023.2. vyd. 2023 [cit. 2024-04-19]. Dostupné z: https://docs.amd.com/r/en-US/ug1399-vitis-hls.

23. ADVANCED MICRO DEVICES, Inc. Vitis AI User Guide (UG1414) [online]. 3.5. vyd. 2023 [cit. 2024-04-19]. Dostupné z: https://docs.amd.com/r/en-US/ug1414-vitis-ai.

24. ADVANCED MICRO DEVICES, Inc. Xilinx FPGA Resource Manager (XRM) [online]. 2023.2. vyd. [B.r.] [cit. 2024-04-19]. Dostupné z: https://xilinx.github.io/XRM/.

25. VENIERIS, Stylianos I.; KOURIS, Alexandros; BOUGANIS, Christos-Savvas. Toolflows for Mapping Convolutional Neural Networks on FPGAs: A Survey and Future Directions [online]. 2018 [cit. 2024-04-22]. Dostupné z arXiv: 1803.05900 [cs.CV].

26. VENIERIS, Stylianos I.; BOUGANIS, Christos-Savvas. fpgaConvNet: A Framework for Mapping Convolutional Neural Networks on FPGAs [online]. 2016, s. 40–47 [cit. 2024-04-22]. Dostupné z DOI: 10.1109/FCCM.2016.22.

27. INTELLIGENT DIGITAL SYSTEMS LAB, IMPERIAL COLLEGE LONDON. fpgaConvNet [online]. 2024 [cit. 2024-04-22]. Dostupné z: https://fpgaconvnet.com.

28. WANG, Ying; XU, Jie; HAN, Yinhe; LI, Huawei; LI, Xiao-Wei. DeepBurning: Automatic Generation of FPGA-based Learning Accelerators for the Neural Network Family [online]. 2016 [cit. 2024-04-22]. Dostupné z DOI: 10.1145/2897937.2898003.

29. ABDELOUAHAB, Kamel; PELCAT, Maxime; SEROT, Jocelyn; BOURRASSET, Cedric; QUINTON, Jean-Charles; BERRY, François. Hardware Automated Dataflow Deployment of CNNs [online]. 2017 [cit. 2024-04-22]. Dostupné z arXiv: 1705.04543 [cs.OH].

30. LIU, Zhiqiang; DOU, Yong; JIANG, Jingfei; XU, Jinwei. Automatic code generation of convolutional neural networks in FPGA implementation [online]. 2016, s. 61–68 [cit. 2024-04-22]. Dostupné z DOI: 10.1109/FPT.2016.7929190.

31. HALL, Mathew; BETZ, Vaughn. HPIPE: Heterogeneous Layer-Pipelined and Sparse-Aware CNN Inference for FPGAs [online]. 2020 [cit. 2024-04-22]. Dostupné z arXiv: 2007.10451 [cs.AR].

32. UMUROGLU, Yaman; FRASER, Nicholas J.; GAMBARDELLA, Giulio; BLOTT, Michaela; LEONG, Philip; JAHRE, Magnus; VISSERS, Kees. FINN: A Framework for Fast, Scalable Binarized Neural Network Inference. In: *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays* [online]. ACM, 2017, s. 65–74 [cit. 2024-04-22]. FPGA '17.

33. BLOTT, Michaela; PREUSSER, Thomas B; FRASER, Nicholas J; GAMBARDELLA, Giulio; O'BRIEN, Kenneth; UMUROGLU, Yaman; LEESER, Miriam; VISSERS, Kees. FINN-R: An end-to-end deep-learning framework for fast exploration of quantized neural networks. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)* [online]. 2018, roč. 11, č. 3, s. 1–23 [cit. 2024-04-22].

34. XILINX. FINN [online]. [B.r.] [cit. 2024-04-22]. Dostupné z: https://xilinx.github.io/finn/.

35. DUARTE, Javier et al. Fast inference of deep neural networks in FPGAs for particle physics. *JINST* [online]. 2018, roč. 13, č. 07, P07027 [cit. 2024-04-22]. Dostupné z DOI: 10.1088/1748-0221/13/07/P07027.

36. AARRESTAD, Thea et al. Fast convolutional neural networks on FPGAs with hls4ml. *Mach. Learn. Sci. Tech.* [online]. 2021, roč. 2, č. 4, s. 045015 [cit. 2024-04-22]. Dostupné z DOI: 10.1088/2632-2153/ac0ea1.

37. FASTML TEAM. *fastmachinelearning/hls4ml* [online]. Zenodo, 2023. Ver. v0.8.1 [cit. 2024-04-22]. Dostupné z DOI: 10.5281/zenodo.1201549.

38. FASTML TEAM. *HLS4ML Documentation* [online]. 2024. [cit. 2024-04-22]. Dostupné z: https://fastmachinelearning.org/hls4ml/.

39. MA, Yufei; SUDA, Naveen; CAO, Yu; VRUDHULA, Sarma; SEO, Jae-sun. ALAMO: FPGA acceleration of deep learning algorithms with a modularized RTL compiler. *Integration* [online]. 2018, roč. 62 [cit. 2024-04-22]. Dostupné z DOI: 10.1016/j.vlsi.2017.12.009.

40. GUO, Kaiyuan; SUI, Lingzhi; QIU, Jiantao; YU, Jincheng; WANG, Junbin; YAO, Song; HAN, Song; WANG, Yu; YANG, Huazhong. Angel-Eye: A Complete Design Flow for Mapping CNN Onto Embedded FPGA. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* [online]. 2018, roč. 37, č. 1, s. 35–47 [cit. 2024-04-22]. Dostupné z DOI: 10.1109/TCAD.2017.2705069.

41. SHARMA, Hardik; PARK, Jongse; MAHAJAN, Divya; AMARO, Emmanuel; KIM, Joon Kyung; SHAO, Chenkai; MISHRA, Asit; ESMAEILZADEH, Hadi. From high-level deep neural models to FPGAs. In: *Microarchitecture (MICRO), 2016 49th Annual IEEE/ACM International Symposium on* [online]. IEEE, 2016, s. 1–12 [cit. 2024-04-22].

42. ZHANG, Chen; SUN, Guangyu; FANG, Zhenman; ZHOU, Peipei; PAN, Peichen; CONG, Jason. Caffeine: Toward Uniformed Representation and Acceleration for Deep Convolutional Neural Networks. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* [online]. 2019, roč. 38, č. 11, s. 2072–2085 [cit. 2024-04-22]. Dostupné z DOI: 10.1109/TCAD.2017.2785257.

43. GUAN, Yijin; LIANG, Hao; XU, Ningyi; WANG, Wenqiang; SHI, Shaoshuai; CHEN, Xi; SUN, Guangyu; ZHANG, Wei; CONG, Jason. FP-DNN: An Automated Framework for Mapping Deep Neural Networks onto FPGAs with RTL-HLS Hybrid Templates. In: [online]. 2017, s. 152–159 [cit. 2024-04-22]. Dostupné z DOI: 10.1109/FCCM.2017.25.
44. GOKHALE, Vinayak; ZAIDY, Aliasger; CHANG, Andre Xian Ming; CULURCIELLO, Eugenio. Snowflake: A Model Agnostic Accelerator for Deep Convolutional Neural Networks [online]. 2017 [cit. 2024-04-22]. Dostupné z arXiv: 1708.02579 [cs.AR].
45. ADVANCED MICRO DEVICES, Inc. *PYNQ Docs* [online]. 2024. [cit. 2024-04-23]. Dostupné z: https://pynq.readthedocs.io/en/latest.
46. ADVANCED MICRO DEVICES, Inc. Vitis AI User Guide (UG1414) [online]. 2.5. vyd. 2022 [cit. 2024-04-23]. Dostupné z: https://docs.amd.com/r/2.5-English/ug1414-vitis-ai.
47. ADVANCED MICRO DEVICES, Inc. Vitis AI Library User Guide (UG1354) [online]. 1.1. vyd. 2020 [cit. 2024-04-23]. Dostupné z: https://docs.amd.com/r/1.1-English/ug1354-xilinx-ai-sdk.
48. ADVANCED MICRO DEVICES, Inc. Xilinx/Vitis-AI [online]. 2022 [cit. 2024-04-23]. Dostupné z: https://github.com/Xilinx/Vitis-AI/tree/v2.5.
49. ADVANCED MICRO DEVICES, Inc. DPUCAHX8H for Convolutional Neural Networks Product Guide (PG367) [online]. 1.2. vyd. 2024 [cit. 2024-04-23]. Dostupné z: https://docs.amd.com/r/en-US/pg367-dpucahx8h.
50. ADVANCED MICRO DEVICES, Inc. Xilinx/Vitis-AI [online]. 2023 [cit. 2024-04-23]. Dostupné z: https://github.com/Xilinx/Vitis-AI/tree/v3.5.
51. ADVANCED MICRO DEVICES, Inc. Xilinx/Vitis-AI-Copyleft-Model-Zoo [online]. 2023 [cit. 2024-05-07]. Dostupné z: https://github.com/Xilinx/Vitis-AI-Copyleft-Model-Zoo.
52. ADVANCED MICRO DEVICES, Inc. Vitis AI Library User Guide (UG1354) [online]. 3.5. vyd. 2023 [cit. 2024-04-29]. Dostupné z: https://docs.amd.com/r/en-US/ug1354-xilinx-ai-sdk.

Obsah příloh

Jako příloha je přiložen jen soubor zp-2023-akcelerace-neuronove-site-na-cloudove-fpga-karte.zip. Ten obsahuje archiv s kopií posledního stavu repozitáře (na posledním commitu) této práce použitého pro vytvořené zdrojové kódy a výstupy testů. Součástí jsou i příslušné readme soubory.

Plná verze repozitáře je dostupná z: `https://gitlab.fit.cvut.cz/skrbek/zp-2023-akcelerace-neuronove-site-na-cloudove-fpga-karte`. Pokud je to možné, je doporučeno použít tuto plnou verzi.