ČESKÉ VYSOKÉ UČENÍ TECHNICKÉ V PRAZE

FAKULTA STROJNÍ

DIPLOMOVÁ PRÁCE

STUDIE NÍZKO-NÁKLADOVÝCH EMBEDDED SYSTÉMŮ PRO REÁLNĚ ČASOVÉ APLIKACE STROJOVÉHO UČENÍ

2021

ONDŘEJ BUDÍK

ZADÁNÍ DIPLOMOVÉ PRÁCE

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Budík** Jméno: **Ondřej** Osobní číslo: **458430**

Fakulta/ústav: **Fakulta strojní**

Zadávající katedra/ústav: **Ústav mechaniky, biomechaniky a mechatroniky**

Studijní program: **Průmysl 4.0**

Studijní obor: **bez oboru**

II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

Studie nízkonákladových embedded systémů pro reálně časové aplikace strojového učení

Název diplomové práce anglicky:

Study of low-cost embedded systems for real-time machine learning applications

Pokyny pro vypracování:

Proveďte rešerši nízkonákladových HW řešení s open-source SW nástroji pro výkonné výpočty se strojovým učením v reálném čase.
Pro vybranou platformu, která představuje vhodný poměr výpočetního výkonu vůči náročnosti na HW a SW realizaci (pro širokou skupinu uživatelů) implementujte reálně-časovou aplikaci se strojovým učení (predikce nebo detekce nebo adaptivní řízení).
Aplikace vybrané HW platformy a strojového učení může být na reálném nebo externě simulovaném systému.
Experimentálně ověřte a vyhodnoťte vlastnosti vámi implementovaného algoritmu a výkonnostních parametrů HW platformy.
Řešení řádně zdokumentujte.
Rozsah práce min. 50 stran + přílohy.
Grafický obsah max 50 %.

Seznam doporučené literatury:

[1] OKAFOR, Kennedy, Genevra CHINWE a Ogungbenro AKINYELE. Hardware description language (HDL): An efficient approach to device independent designs for VLSI market segments [online]. 2011. ISBN 978-1-4673-0758-1. Dostupné z: doi:10.1109/ICASTech.2011.6145181
[2] BUKOVSKY, Ivo a Noriyasu HOMMA. An Approach to Stable Gradient-Descent Adaptation of Higher Order Neural Units. IEEE Transactions on Neural Networks and Learning Systems [online]. 2016, 1–13. ISSN 2162-237X, 2162-2388. Dostupné z: doi:10.1109/TNNLS.
[3] Tensor Processing Unit (TPU). Semiconductor Engineering [online]. [vid. 2020-06-24]. Dostupné z: https://semiengineering.com/knowledge_centers/integrated-circuit/ic-types/processors/tensor-processing-unit-tpu

Jméno a pracoviště vedoucí(ho) diplomové práce:

doc. Ing. Ivo Bukovský, Ph.D., U12110.3

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **28.04.2021** Termín odevzdání diplomové práce: **13.08.2021**

Platnost zadání diplomové práce: _____

doc. Ing. Ivo Bukovský, Ph.D. podpis vedoucí(ho) práce	doc. Ing. Miroslav Španiel, CSc. podpis vedoucí(ho) ústavu/katedry	prof. Ing. Michael Valášek, DrSc. podpis děkana(ky)

III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____ Datum převzetí zadání

_____ Podpis studenta

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně s tím, že její výsledky mohou být dále použity podle uvážení vedoucího diplomové práce jako jejího spoluautora. Souhlasím také s případnou publikací výsledků diplomové práce nebo její podstatné části, pokud budu uveden jako její spoluautor.

V Praze, dne _____ *Podpis* _____

Poděkování

Velmi děkuji své rodině a přátelům, kteří mě po celou dobu studia podporovali, kteří mi vždy pomohli jak mohli a kterým vděčím za veškeré své možnosti. Bez nich by to zcela jistě nebylo možné.

Tímto bych rád poděkoval vedoucímu této práce doc. Ing. Ivu Bukovskému za všestrannou pomoc, množství cenných a inspirativních rad, podnětů doporučení, připomínek a zároveň velkou trpělivost s obdivuhodnou ochotou při konzultacích poskytnutých ke zpracování této práce a za vše, co jsem se díky němu naučil.

Souhrn: Cílem a výstupem této práce je studie dostupných nízkonákladových HW řešení s open-source SW nástroji pro výkonné výpočty se strojovým učením v reálném čase a vlastní implementace řešení problematiky detekce anomálií na jedné zvolené platformě. Zvolenou platformou je pro účely této práce vývojová deska PYNQ-Z1 s čipem FPGA. Tato deska poskytuje široké možnosti budoucích aplikací a díky dnes již podstatně usnadněnému HW programování jí bylo vhodné otestovat i z pohledu náročnosti implementace pro strojní inženýry. Práce se tak zabývá návrhem a vývojem vlastního HW návrhu výpočetního akcelerátoru pro čip FPGA, jehož výkon je zde poté otestován ve zde též vyvinutém detekčním SW v jazyce Python. V závěru práce je vyhodnocena vhodnost užití zvolené platformy a jsou zde shrnuty možnosti vhodných využití pro reálné aplikace.

Klíčová slova: Vestavěné systémy, PYNQ, HDL, Vivado, Vitis, neuronová síť, HONU, Learning Entropy, detekce anomálií, neobvyklé stavy

Summary: The aim and output of this thesis is a research of available low-cost HW solutions with open-source SW tools for real-time machine learning computations and the actual implementation of the solution to the anomaly detection problem on one selected platform. The chosen platform for purposes of this work is a PYNQ-Z1 development board with a FPGA chip. This board provides a wide range of possibilities for future applications and due to the now substantially easier HW programming, it was also suitable to test it in terms of implementation complexity for mechanical engineers. The thesis thus deals with the development of a custom HW design of a computational accelerator for the FPGA chip, whose performance is then tested in the original detection software written in Python. The thesis concludes with an evaluation of possible use-cases of the chosen platform and summarizes the possibilities of suitable uses for real applications.

Key words: Embedded systems, PYNQ, HDL, Vivado, Vitis, neuron network, HONU, Learning Entropy, anomaly detection, unusual states

Obsah

Úvod

V dnešní době je dostupné široké spektrum nízko-nákladových HW prostředků, jednodeskových počítačů a embedded zařízení, které lze buďto přímo nasadit nebo na nich rychle vyvíjet prototypy pro reálně časové aplikace a přitom na nich implementovat i klasické algoritmy, či vyvíjet, zkoumat a testovat další algoritmy včetně strojového učení pro úlohy zpracování reálně časových dat jako jsou predikce, detekce a řízení.

Kromě různého výpočetního výkonu jsou tyto zařízení také různé z hlediska náročnosti jejich implementace a tedy nutných programátorských znalostí a dovedností, což ovlivňuje jejich využitelnost a nasaditelnost v praxi nebo třeba i v mechatronických experimentech a projektech i na naší Fakultě.

Tématem zpracovaným v této diplomové práci je rešerše nízko-nákladových hardwarových řešení s využitím softwarových nástrojů typu open-source pro výkonné výpočty se strojovým učením v reálném čase, výběr platformy, která by byla vhodná jak pro vývoj a výzkum, tak i pro výuku a musí na ní být možné navrhovat, implementovat a testovat právě algoritmy strojového učení. Dále je zde vyhodnocena využitelnost a perspektivnost zvolené HW platformy a je zde diskutována náročnost vývoje aplikací pro zvolenou platformu.

Algoritmy strojového učení využité v této práci jsou architektury HONU [1, 2]. Tyto algoritmy jsou v této práci využity především k detekci neobvyklých stavů sledovaného systému na principu nové míry novosti se strojovým učením [3, 4]. Ostatní možnosti aplikace jako jsou predikce či adaptivní řízení jsou zde pouze diskutovány. Hlavní logika algoritmů je napsána v jazyce Python s akcelerací vybraných výpočetních operací pomocí výkonného výpočetního čipu zvoleného hardwarového řešení.

Neuronové sítě jsou vzhledem k jejich relativně vysoké výpočetní náročnosti obvykle jen obtížně aplikovatelné na problematiky vyžadující zpětnou vazbu v téměř reálném čase. Nejčastěji se dnes setkáme s aplikací neuronových sítí na výkonných výpočetních serverech kam běžně přes internet propojené embedded zařízení zasílá sbíraná data ze senzorů a až na základě odpovědi daného serveru provede požadovaný úkon. Tento způsob přenosu a výpočtu již ovšem může trvat příliš dlouho či být příliš nespolehlivý k relevantní predikci chování systému či ke včasnému zásahu v případě detekce anomálního chování. Obdobně problematické je toto i v případě adaptivního řízení, kde kombinace embedded zařízení s výpočetním serverem nemusí být pro řízený systém dostatečně rychlá. Objevují se zde ovšem

stále výkonnější a dostupnější čipy a jimi osazená dostupná embedded zařízení, která již dokáží provádět i velmi obtížné výpočty v téměř reálném čase. Tato zařízení tak umožňují aplikaci neuronových sítí i bez využití výpočetních serverů a internetového připojení a je tedy možné pomocí těchto zařízení dosáhnout detekce, predikce či adaptivního řízení v téměř reálném čase přímo na místě sběru dat.

V teoretické části jsou prozkoumány možnosti jednotlivých výpočetních čipů, které je dnes možné nalézt na embedded zařízení. V této části je stručně vysvětlen rozdíl mezi jednotlivými čipy a jsou představeny jak jejich hlavní výhody, tak jejich nevýhody či jejich případná omezení plynoucí z jejich hardwarové architektury. Dále jsou zde ve zkratce představeny zvažované embedded zařízení s ohledem na cíle této práce a možnostech jejich případného budoucího rozšíření. Ze zde představených zařízení jsou vybrána dvě možná řešení, která nejlépe splňují dané požadavky. Konkrétně se jedná o vývojovou desku Coral edge 2.1.6 [5] a vývojovou desku PYNQ-Z1 2.1.3 [6]. Vývojová deska Coral edge je osazena výkoným čipem TPU 1.4 [7] a poskytuje širokou podporu frameworku Tensorflow [8] s minimálními latencemi mezi CPU architektury ARM 1.1.2 [9] a dedikovaným čipem TPU, který je energeticky velmi efektivní. Druhá zvažovaná vývojová deska, PYNQ-Z1, je osazená čipem architektury Zynq-7000, který obsahuje jak dvoujádrový CPU architektury ARM, tak čip FPGA 1.3 [10]. Ačkoliv tento FPGA čip není moc velký, pro účely této práce je naprosto postačující. Čip FPGA je obdobně jako TPU velice energeticky efektivní a poskytuje velmi vysoké výpočetní výkony. Kromě toho může být v budoucí aplikaci téměř celá aplikace napsána na hardwarové úrovni, čímž je možné dosáhnout až na výpočetní rychlost v téměř reálném čase. Pro účely této práce byla z těchto dvou zvažovaných možností zvolena vývojová deska PYNQ-Z1 k detailnějšímu zpracování a testování. Důvody k této volbě jsou zde 2.2 podrobně diskutovány.

Jelikož byla zvolena vývojová deska osazená čipem FPGA, je v této práci stručně vysvětlena metodika hardwarového programování 3 včetně vysvětlení principu programování pomocí nejužívanějších hardwarově popisných jazyků [11–13]. Dále je zde představen software Vitis a Vivado HLS 3.3, který je určen pro vývoj hardwarového návrhu čipu FPGA. Tyto softwarové nástroje zásadním způsobem usnadňují celkový vývoj návrhu designu FPGA a snižují nároky na odborné znalosti i investovaný čas. Tento software dokáže vysokoúrovňové jazyky C a C++ přeložit na úroveň hardwarově popisných jazyků a následně je přeložit až na úroveň registrů.

Závěrem teoretické části je detailně vysvětlen přístup neuronových jednotek HONU 5 včetně vybraných metodik jejich adaptace jako jsou například gradient descent 5.2.1, Levenberg-Marquardt 5.2.2 nebo Adam 5.2.4. Je zde také vysvětlen princip detekce anomálií pomocí Learning Entropy 6.1 a to jak pro online, tak dávkové metody učení a nastíněn princip detekce dlouhodobých anomálií 6.2.

V praktické části jsou nejdříve popsány vstupní testovací data pro dvě možné aplikace zvoleného embedded zařízení s vyvinutým detekčním softwarem, kde jedním ze zvažovaných scénářů je monitoring chování dynamického systému a druhým je dávkové vyhodnocování dat z vícero měřících stanic.

Pro účely monitoringu dynamického systému 7.1 byl vyvinut simulátor, jehož simulace může probíhat čistě na úrovni vývojové desky PYNQ-Z1 či může být simulována externě například pomocí zařízení Raspberry. Možnost externí simulace byla zvolena proto, aby se zde testovaná aplikace co nejvíce podobala možné reálné aplikaci. Ke snadné obsluze bylo pro tento simulátor a online detektor vyvinuto grafické rozhraní v prostředí Jupyter notebook, které je na platformě zcela nezávislé a je tak možné simulační zařízení zvolit zcela libovolně.

Dávkové vyhodnocení je aktuálně prováděno pouze lokálně bez grafického rozhraní na anonymních datech z testovacích a měřících stanic 7.2 automotive průmyslového partnera Robert Bosch, s.r.o. Cílem je otestovat výkonnost zde zvoleného embedded zařízení pro dávkově zpracovávaná data pro procesy, kde není možné provádět online monitoring systému a jedno toto zařízení by tak dokázalo obstarávat větší množství stanic.

Dále je zde věnován velký prostor postupu při návrhu a vývoji bitstreamu pro FPGA 8. Kromě detailního popisu postupu vývoje a porovnání jak teoretické tak reálné výpočetní rychlosti 8.2 jsou v této části také diskutovány možnosti optimalizace kódu pomocí direktiv překladu, kde nejužívanější z nich jsou zde detailně popsány 8.3.

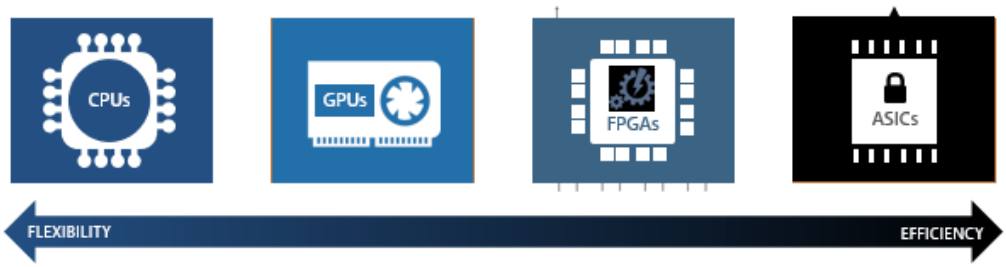
V závěru praktické části je vyhodnocen výpočetní výkon pouze na procesoru ARM a poté s akcelerací výpočtů pomocí FPGA na desce PYNQ-Z1. Jednotlivé vyvinuté implementace výpočetních akcelerátorů jsou porovnány a jejich výpočetní výkony a omezení jsou diskutovány. V neposlední řadě je zde otestována i přesnost implementovaných detekčních metodik AISLE a ABSLE 6.1 pro detekci náhlých změn a detekce dlouhodobých chyb pomocí vyhodnocení dle 6.2.

1 DOSTUPNÉ HARDWAROVÉ MOŽNOSTI

Ať se již se jedná o implementaci adaptivního řízení, predikci nebo detekci neobvyklých stavů systému pomocí HONU, jejichž princip je vysvětlen v 5, může být tato aplikace jak výpočetně málo náročná, pro případ LNU s malým počtem vstupů, tak výpočetně velmi komplexní, pro případ CNU s vícero vstupy. Kromě toho některé aplikace vyžadují zpětnou vazbu v prakticky reálném čase, aby byly výsledky relevantní a použitelné ke správným akčním zásahům či včasným varováním obsluhy. Z dnešních hardwarových řešení použitelných k těmto specifickým výpočtům se nabízí hned několik následujících možností implementace, a to:

1. Central Processing Unit (CPU) architektury x86-64
2. Centra Processing Unit (CPU) architektury ARM
3. Graphic Processing Unit (GPU)
4. Field Programmable Gate Array (FPGA)
5. Tensor Processing Unit (TPU)

každé z těchto řešení má své výhody, ale bohužel i nevýhody a je tedy nutné provést optimální kompromis vzhledem k požadované rychlosti, ceně, velikosti, ale i obtížnosti implementace či spolehlivosti cílového řešení [14]. První tři zde zvažovaná hardwarová řešení patří k těm běžněji využívaným především díky jejich snadné implementaci zatím co poslední dvě řešení byly pro účely aplikace neuronových sítí až do nedávna využívány jen velmi okrajově kvůli jejich obtížnější implementaci. S novými softwarovými nástroji jsou ovšem i tato řešení relativně snadno implementovatelná a jejich využití velmi rychle roste.



Obr. 1: Vizuální porovnání náročnosti implementace, flexibility a efektivity jednotlivých čipu. (Převzato a upraveno dle [14]).

1.1 CENTRAL PROCESSING UNIT (CPU)

Centrální procesorová jednotka je základem téměř všech dnešních počítačů a embedded zařízení. Tato jednotka se vyznačuje tím, že pomocí její instrukční sady dokáže vykonávat vysoké množství nejrůznějších operací jako jsou například výpočetní úkony, logické operace, řízení chodu systému, anebo správa vstupně výstupních operací, čímž se zásadně liší od čipů GPU či ASIC, které dokáží vykonávat pouze velmi specifické operace. Tato univerzálnost se ovšem podepisuje na dosahovaném výpočetním výkonu, který téměř nikdy nedosahuje rychlostí čipů k tomu určených. V případě jednotek CPU je také zcela zásadní jejich architektura, která je definována jejich instrukční sadou. V dnešní době jsou využívány především dvě hlavní architektury, a to x86-64 a ARM, jejichž rozdíly jsou popsány níže.

1.1.1 ARCHITEKTURA X86-64

Výkonné procesory architektury x86-64 jsou v dnešní době poskytovány výhradně společnostmi Intel a AMD (Advanced Micro Devices) [15]. Procesory této architektury se vyznačují 64-bitovou instrukční sadou vytvořenou především společností AMD s podporou 32-bitové instrukční sady, která byla vytvořena především společností Intel. Hlavní charakter této architektury je podpora 64-bitových registrů, 64-bitových aritmeticko-logických operací a 64-bitových virtuálních adres [16]. Procesory této architektury mohou mít dnes od 1 až po 128 jader, které zpracovávají dané požadavky sekvenčně a poskytují tak velice přesné výpočty, schopnost adresace velkokapacitní paměti a obvykle dosahují frekvencí mezi 1 a 5GHz [17]. Díky těmto schopnostem poskytují velice vysoký výpočetní výkon, který je dále podpořen širokou instrukční sadou, která dále urychluje široké spektrum výpočetních operací na hardwarové úrovni. Další výhodou je z pravidla modulární řešení sestavení veškerých hardwarových periferií nezbytných k běhu systému založeném na tomto procesoru, díky čemuž je možné použitý hardware plně optimalizovat pro cílovou aplikaci.

Nevýhodou tohoto řešení je vysoká spotřeba elektrické energie, která se obvykle pohybuje mezi 15 až 250W na CPU samotné a obvykle dalších 15 až 50W na základní desku a nezbytné periferie [17]. Dále je zde nutnost využití komplexních operačních systémů, které jsou podporovány patřičnými ovladači procesoru a řadiči zvolené základní desky. Běžně se zde jedná o operační systémy Microsoft Windows či Linux. Vzhledem ke komplexnosti těchto operačních systémů, je jisté procento výkonu procesoru nutně využito právě patřičným operačním systémem a výsledné řešení je zcela závislé na kvalitě implementace správce

procesů a rychlosti odpovědi systému na jakýkoliv požadavek. Tato odpověď bývá zpravidla pomalá, jelikož daný požadavek musí obvykle počkat, až na něj přijde řada.

Díky komplexnímu operačnímu systému je ovšem možné programovat požadovaný řídící program v prakticky libovolném programovacím jazyce a to bez starosti o kompatibilitu či hardwarové omezení, čímž je obtížnost implementace na softwarové úrovni silně minimalizována.

1.1.2 ARCHITEKTURA ARM

Procesory architektury ARM jsou dnes poskytovány více než desíti výrobci [18], z nichž největšími jsou Texas Instruments, Xilinx, Samsung a Ldt-ARM. Ačkoliv jsou procesory architektury ARM dnes již k dispozici také v 64 bitové verzi, běžněji jsou využívány stále jejich 32 bitové deriváty. Procesory mohou mít od 1 až po 80 jader, které zpracovávají dané požadavky sekvenčně. Obvykle jsou tyto procesory párovány s podstatně menší pamětí než je tomu v případě procesorů architektury x86-64. Dalším rozdílem je jejich dosahovaná frekvence. Tyto procesory běžně fungují na podstatně nižších frekvencích, které se obvykle pohybují v rozmezí mezi 150MHz až 2GHz. Instrukční sada těchto procesorů bývá také značně omezena a výpočetní výkon těchto procesorů je podstatně nižší, než je tomu v případě procesorů architektury x86-64 [9].

Tyto procesory ovšem byly od svého začátku navrženy tak, aby poskytovali co možná nejvyšší efektivitu vzhledem ke spotřebě a jejich spotřeba se běžně pohybuje od desetin wattů až po jejich desítky. Tyto procesory také obvykle nemívají modulární řešení a bývají prodávány buďto jako nerozebíratelné celky procesoru, základní desky a periferií, či jako procesory samotné, ale s nutností kompletního návrhu nezbytné základní desky a potřebných periferií, což je velice náročný a obvykle nákladný proces. [9, 19, 20].

Tyto procesory jsou obvykle kompatibilní s výrazně odlehčenými verzemi operačních systémů jako jsou Linux nebo Windows, ale existují i operační systémy navržené především pro ARM jako jsou například Phoenix, RIOT, LiteOS, Frosted a mnoho dalších, které jsou uvedeny v [21]. I tyto systémy jsou silně závislé na kvalitě zpracování správce procesů a rychlosti odpovědi použitého systému, která bývá obvykle delší než je tomu u využití architektury x86-64 vzhledem k nižšímu výkonu procesorů architektury ARM. Tyto odlehčené operační systémy obvykle také podporují běh širokého spektra programovacích jazyků a opět tedy značně ulehčují softwarovou implementaci výsledného řešení.

1.2 GRAPHIC PROCESSING UNIT (GPU)

Grafické čipy ke svému chodu zpravidla potřebují kompatibilní CPU čip, který se stará o řízení chodu systému a plánování specifických úloh, které mohou být pomocí GPU dále zpracovávány. GPU je primárně určeno ke zpracování velkého množství paralelních výpočtů a je tedy v této disciplíně mnohem výkonnější než čipy CPU. Vyžaduje ovšem speciální způsob programování, který je specifický pro každého výrobce těchto čipů. Grafické čipy jsou dnes poskytovány dle dvou hlavních cílových skupin. Tyto skupiny se dají rozdělit dle druhu CPU, se kterým mají spolupracovat a jedná se tedy o čipy určené ke spolupráci s procesory x86-64 nebo ARM.

Skupina určená ke spolupráci s procesory x86-64 poskytuje velmi vysoký výpočetní výkon za cenu vyšší spotřeby elektrické energie a je poskytována především společnostmi Nvidia, AMD a Intel [22]. Tito výrobci také poskytují knihovny nezbytné k implementaci výpočetních akcelerátorů pomocí GPU, jako je například knihovna CUDA určená pro čipy od Nvidie, nebo OpenCL určené pro čipy AMD a Intel [23, 24].

Druhá skupina, která je určená ke spolupráci s procesory ARM, poskytuje podstatně nižší výkon než skupina první, jelikož je zde upřednostňována především spotřeba elektrické energie a efektivita na watt, obdobně jako je tomu právě v případě CPU architektury ARM. Hlavními výrobci těchto čipů jsou společnosti Ldt-ARM, Qualcomm či Boardcom [22]. Výrobci těchto GPU neposkytují, až na speciální omezené případy, vlastní knihovny nezbytné k implementaci výpočetních akcelerátorů pomocí GPU a většina z nich spoléhá na jistou míru kompatibility s OpenCL. Míra náročnosti implementace a efektivity byla podrobně prozkoumána v [25], kde se autoři museli vypořádat s řadou problémů kompatibility a z jejich závěrů vyplývá, že s aktuálními knihovnami je implementace velice náročná.

Vzhledem k nutnosti operačního systému a řídícího CPU, roste doba odpovědi systému na požadavek. Tato doba odpovědi může být dle cílové implementace již kritická a toto řešení může být neimplementovatelné. Softwarová implementace vzhledem k nutnosti využití specifických knihoven a s tím spojené jazykové omezení dále zvyšuje náročnost dané implementace. Z těchto důvodů se akcelerace pomocí GPU využívá především ke zpracování obrazu či offline trénování rozsáhlejších neuronových sítí, u kterých výpočetní čas na CPU převyšuje váhu času nutného k jejich implementaci pomocí GPU.

1.3 FIELD PROGRAMMABLE GATE ARRAY (FPGA)

Programovatelná hradlová pole jsou dnes vyráběna především společnostmi Xilinx, Intel a Altera [26]. V tomto čipu jsou speciální digitální integrované obvody, které obsahují různě složité programovatelné bloky pospojované nastavitelnou maticí spojů. Díky tomuto není FPGA omezeno 32 nebo 64 bitovou architekturou, jelikož se dá volně naprogramovat k zpracování požadované bitové délky. S přibývajícími bity ovšem rostou i požadavky na hardwarové zdroje čipu, které jsou obvykle velmi limitované. Frekvence těchto čipů se vždy navrhuje v závislosti na implementovaném logickém obvodu a obvyklé frekvence se pohybují mezi 10MHz až 500MHz [27].

Implementace pomocí FPGA bývá obvykle obtížnější, jelikož se tento čip musí programovat pomocí jazyků sloužících pro popis hardwaru. Nejpoužívanějšími programovacími jazyky pro popis propojení hradlových polí jsou dnes jazyky VHDL a Verilog. Tento způsob programování je dnes ovšem značně usnadněn díky velice pokročilým implementačním nástrojům, které dokáží relativně spolehlivě přeložit algoritmy napsané v jazyce C a C++ až do jazyků hardwarově popisných.

Hlavní výhodou FPGA je široká variabilita logických obvodů, které je možné na těchto deskách aplikovat, což je velice výhodné během návrhu prototypů, jelikož je lze velice rychle upravovat a testovat v porovnání s ostatními metodami. Kromě toho jsou FPGA také velice energeticky efektivní a jejich obvyklá spotřeba se pohybuje v řádu jednotek až desítek wattů. Nevýhodou těchto čipů je především omezená dostupnost logických prvků na daném čipu a jejich cena, která je i dnes poměrně vysoká [28].

FPGA jako takové ke svému běhu nepotřebuje operační systém a jeho odezva tedy bývá obvykle velice rychlá a závislá především na kvalitě implementace logického obvodu. Výpočetní výkon čipu FPGA bude pro potřeby neuronových sítí silně ovlivněn jednotkami DSP, neboli Digital Signal Processor. Jedná se o specializovaný typ logického obvodu, který je vytvořen pro práci se signály a je schopný pracovat s velkým množstvím dat najednou. Jeho architektura je co nejvíce uzpůsobena práci s vektory a maticemi. Tyto jednotky umí provést operace typu vynásob a přičti (multiply and accumulate) během jediné instrukce [29]. Tyto operace jsou hojně využívány právě v neuronových sítích, jak je dále uvedeno v 2.1 a jejich celkový počet bude velmi důležitý pro výpočetní výkon cílového návrhu obvodu na čipu FPGA.

1.4 TENSOR PROCESSING UNIT (TPU)

Čipy TPU jsou dnes komerčně poskytovány především společností Google, která tento ASIC (Application Specific Integrated Chip) v roce 2016 poprvé představila a postupně jej sama nasadila do svých služeb jako jsou například Google Translate, Photos nebo Street View. Od roku 2018 Google poskytuje online službu cloudového výpočetního centra pro AI, které je založené výhradně na těchto čipech a dosahuje poměrně vysokých výpočetních hodnot (přes 420 teraflops na rack) při spotřebě okolo 250 až 300 wattů. Klíčovým je ovšem konec roku 2019, jelikož Google uvedl TPU čipy i pro širokou veřejnost a to jak v podobě vývojových desek, tak akcelerátorů ale i čipu samotného, čímž značně rozšířil aplikovatelnost těchto čipů v průmyslu [5, 7].

Implementace vlastních řešení je v současné době omezená pouze na využití TensorFlow Lite API či Edge TPU API, kde obě tato API spoléhají na vytvořený model neuronové sítě pomocí TensorFlow či Cloud AutoML Vision, který se nahrává do paměti TPU čipu. Toto řešení je tedy primárně určeno pro zpracovávání obrazu a implementace predikce, adaptivního řízení či detekce anomálií v dynamických systémech je zde do jisté míry komplikováno [30].

TPU čip je určen ke spolupráci s řídícím CPU a potřebuje tedy ke svému chodu kompatibilní operační systém. Pro vývojové desky je od Googlu poskytována odlehčená verze Linuxu, které obsahuje všechny potřebné ovladače. Bohužel zde ale obdobně, jako je tomu v případě GPU, vzrůstá doba odpovědi. Není-li využito integrovaného řešení v jednom čipu, může být doba odpovědi v závislosti na cílovém systému již kritická a toto řešení může být neimplementovatelné.

2 EMBEDDED ZAŘÍZENÍ

V našem každodenním životě se každý z nás dostává do kontaktu s mnoha různými elektronickými zařízeními v té či oné podobě. Některá z nich jsou relativně jednoduchá zařízení používaná pro jednotlivé úkoly, jako jsou například kalkulačky, dálkové ovladače či výtahy, jiná zase stejně složité jako smartphony, počítače či tablety, které lze relativně snadno naprogramovat k provedení mnoha různých úkolů.

Protože je tato kapitola věnována vestavěným systémům, je důležité zmínit, že většina dnes používaných elektronických zařízení jsou právě embedded (vestavěná) zařízení. Je však

stejně tak důležité si uvědomit, že ne každé zařízení je opravdu vestavěné. Embedded zařízení se nachází někde mezi prostými „hloupými“ elektronickými zařízeními, která byla postavena bez jakékoliv složitější logiky, jako jsou například staré pračky bez displeje a procesoru a mezi komplexními zařízeními jako jsou počítače či smartphony. Abychom pochopili, proč se některá zařízení nazývají embedded a jiná ne, je důležité rozlišit co embedded zařízení ve skutečnosti jsou a jaké jsou jejich vlastnosti. Definice jednotlivých embedded zařízení se budou vždy lehce lišit, ale jejich hlavní charakteristiky zůstávají vždy stejné. Tyto charakteristiky jsou:

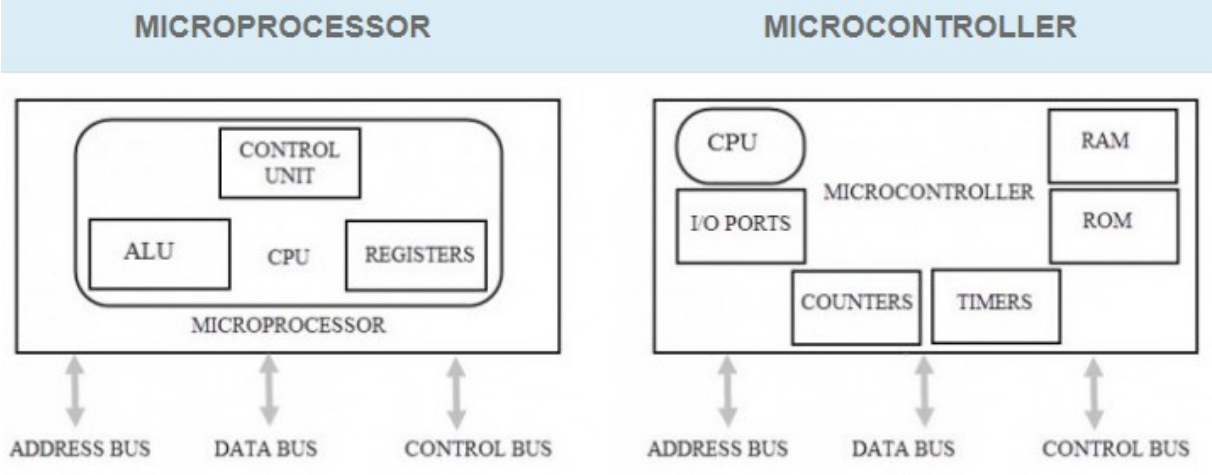
- Mají svůj vlastní procesor
- Jsou postaveny za účelem vykonávání specifického úkolu, či velice malého počtu dílčích úkonů.
- Mají svou vlastní paměť (operační a úložnou)

To znamená, že dle této charakteristiky z pomyslného seznamu embedded zařízení vypadávají smartphony, počítače a zařízení jim podobná kvůli jejich víceúčelovému použití. Využití zařízení na bázi počítačů na místě vyhrazeném pro embedded zařízení může znamenat zvýšení flexibility za cenu vyšších nákladů a obvykle i fyzické velikosti. Obecně spíš platí, že výrobci preferují ve finálních produktech jednodušší řešení s vestavěnými zařízeními před obvykle nadbytečnými komplexními jednotkami. Jejich nadbytečnost výkonu se totiž vždy projeví do ceny konečného řešení. Vzhledem k požadavkům na stále schopnější embedded zařízení je ovšem stále obtížnější kategorizovat, zda je zařízení ještě pořád embedded, nebo se jedná o počítač pro všestranné využití.

Právě proto, že existuje velké množství různých hardwarových konfigurací, je nezbytné provést opravdu důkladný průzkum dostupných řešení na trhu. Je obvyklé, že první zařízení, které se zdá být slibné, nemusí být to nejlepší a to ani při těch nejlepších úmyslech. Volba a návrh vhodného softwarového řešení je neméně důležitá. Vybereme-li hardware s nedostatečnou pamětí či příliš slabým CPU, ani nejlepší software nedokáže překonat tato omezení. Totéž platí i naopak. Nemůžeme očekávat zázraky, pokud náš software bude silně neoptimalizovaný a bude zbytečně využívat systémové zdroje.

Embedded zařízení běžně využívají mikroprocesor nebo mikrokontrolér. Rozdíl mezi nimi spočívá v tom, že mikroprocesor je osazen na desku s mikrokontrolérem a dalšími komponentami jako jsou RAM, ROM, flash, GPIO piny nebo jiné periferie. Mikrokontrolér je vlastně takový miniaturní počítač sám o sobě a nevyžaduje ke svému chodu žádné další

obvody, jelikož v sobě již má všechny potřebné komponenty. Mikrokontroléry lze tedy nazvat jakýmsi srdcem vestavěných zařízení. Mikroprocesory na rozdíl od mikrokontrolerů vždy vyžadují ostatní periferie ke svému fungování. Mikroprocesor samotný obsahuje pouze CPU, mezipaměť, paměť DRAM a někdy i GPU [31].



Obr. 2: Zjednodušené porovnání mikroprocesoru a mikrokontroleru. Mikrokontroler je jakýsi minipočítač sám o sobě a má v sobě již všechny potřebné komponenty ke své funkčnosti. Mikroprocesor bývá zpravidla osazen na desku s mikrokontrolerem, jelikož vždy vyžaduje ostatní periferie ke svému fungování. Mikroprocesor ovšem obvykle bývá podstatně výkonnější než mikrokontrolér (převzato a upraveno dle [31]).

Během výběru embedded zařízení je vhodné položit si následující otázky k určení rozhodujících faktorů. Tyto otázky uvažují jak softwarové, tak hardwarové požadavky na výsledné zařízení, aby bylo možné nalézt nejvhodnější dostupné zařízení.

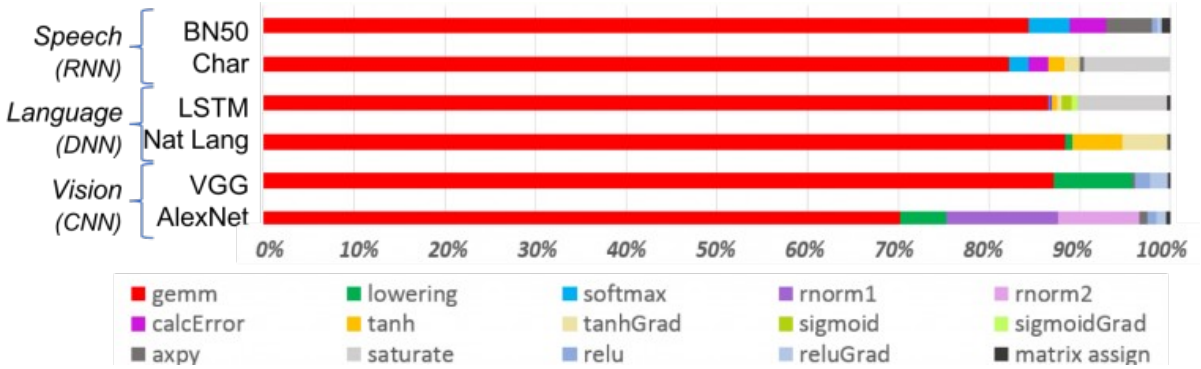
- Kolik RAM / ROM / HDD / SSD / Flash opravdu potřebujeme?
- Musíme užívat CPU / GPU / FPGA / TPU k dosažení našeho výsledku?
- Bude dané zařízení napájeno z baterie či zásuvky?
- Musí dané zařízení poskytovat konektivitu jako jsou USB / Ethernet / Wi-Fi / Bluetooth / a další?
- Jaká je maximální cílová cena?
- Mělo by mít dané zařízení operační systém?
- Jaký programovací jazyk plánujeme využít?
- Jaké problémy budeme řešit pomocí metod strojového učení a jaký pro to potřebujeme výkon?

- Jaké algoritmy / metody / frameworky / architektury jsou pro projekt nezbytné a jsou podporovány daným zařízením?
- Bude trénovací část strojového učení probíhat na daném zařízení?
- Jaká je požadovaná rychlost výpočtů?
- Jak často se bude měnit program zařízení?

Před hledáním odpovědí na uvedené otázky, je však vždy vhodné provést rešerši řešení podobného problému. Je totiž vždy mnohem rychlejší upravit hotové řešení k našim potřebám než začít projekt řešit zcela od začátku. V případě této práce bohužel nebylo obdobné řešení nalezeno a je tedy nutné jít delší vývojovou cestou.

2.1 HARDWAROVÉ MOŽNOSTI PRO EMBEDDED ZAŘÍZENÍ

Pro běžná embedded zařízení je výpočetní rychlost jedno z jejich hlavních omezení, jelikož se vždy jedná o minimalizované verze jejich velkých protějšků. Valná většina těchto embedded zařízení také postrádá širší hardwarovou podporu pro výpočty maticových operací, což je nejběžnější operace při využití neuronových sítí, jak již detailně prozkoumala společnost IBM v [32]. Společnost IBM testovala modely sítí ke zpracování řeči, textu a obrazu, přičemž jak je vidět na Obr. 3 nejvíce výpočetního času připadá právě na general matrix multiplication (gemm).



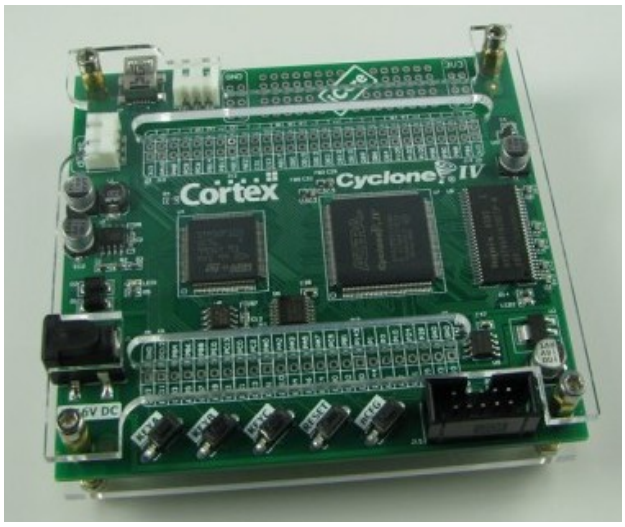
Obr. 3: Výpočetní čas strávený na jednotlivé úkony v uvedených modelech neuronových sítí. Nejvíce výpočetního času je vždy věnováno na "General Matrix Multiply" (gemm) (převzato z [32]).

Ačkoliv se v případě testování společnosti IBM jednalo o hluboké neuronové sítě, v případě využití HONU a jiných mělkých sítí bude gemm také hlavním výpočetním úkonem který bude daná neuronová síť provádět. Na základě poznatků z 1 a 2 byli vybrány následující

možné komerčně dostupné hardwarové řešení pro cílovou implementaci identifikace a řízení dynamických systémů.

2.1.1 STM32F103VC EP4CE6E144

Tato vývojová deska je osazena mikrokontrolerem SMT32F103VC založeným na architektuře ARM Cortex-M [33]. Mikrokontrolery této řady jsou oblíbené především pro jejich vysokou datovou propustnost, velké množství I/O konektorů, nízkou cenu a vysokou spolehlivost. Kromě toho je tato deska také osazena čipem FPGA od společnosti Intel [34], se kterým je možné komunikovat pomocí mikrokontroleru. Tato deska byla vyvinuta jako vývojový kit čímž značně usnadňuje tvorbu prototypu a následné testování před vytvořením výsledného embedded zařízení. Vzhledem k velmi omezeným systémovým zdrojům ovládacího mikrokontroleru, je nutné využít velmi odlehčenou verzi operačního systému



Obr. 4: Vývojová deska SMT32F103VC osazená mikrokontrolérem a čipem FPGA

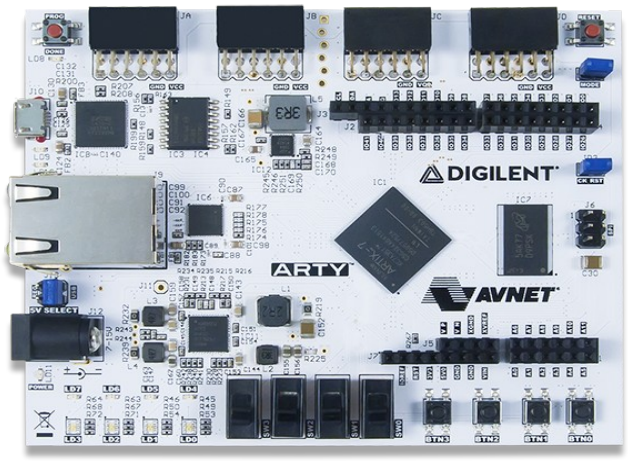
Linux a jako programovací jazyk zvolit například jazyk C pro jeho nízké systémové nároky. Výhodou využití této desky je velká podobnost skutečnému low-cost řešení. Ovšem zcela zde zaniká možnost využití interpretovaných jazyků jako je například Python, vzhledem k nedostatečnému výkonu hardwaru, čímž obtížnost implementace v tomto případě značně narůstá.

Tab. 1: Specifikace pro STM32F103VC [33] EP4CE6E144 [34]

Specifikace pro STM32F103VC EP4CE6E144	
CPU	Single core, ARM® 32-bit Cortex-M3 MCU

Frekvence CPU	Až 72 MHz
SRAM / RAM	8 MB
Logic slices	115 000
DSP	280
BRAM	4 MB
Napájení	5 V
Operační systém	Linux Debian
Konektivita	USB, UART, SPI, CAN, USARTs, I2C, 65 GPIO (CPU), 46 GPIO (FPGA)
Úložný prostor	2 MB Flash
Rozměry	90 x 100 mm
Cena	121 €

2.1.2 ARTY A7 100T



Obr. 5: Vývojová deska Arty A7 100T osazená čipem FPGA

Vývojová deska Arty A7 100T od společnosti Digilent je osazena čipem FPGA Artix-7™ společnosti Xilinx, širokým spektrem konektorů, tlačítek, přepínačů a indikátorů jak je vidět na Obr. 5 [35]. Tato deska byla navržena speciálně pro využití MicroBlaze Soft Processing systému, což je systém, kde se část FPGA chová podobně jako CPU a stará se tak o úkony vykonávané běžně právě pomocí CPU, ačkoliv jím není osazena. Díky tomu mohou být jisté výpočetní úkony akcelerovány na hardwarové úrovní pomocí FPGA, zatím co o chod programu se se může starat procesorová část. Na této desce také probíhal vývoj projektu FμPy [36], jehož cílem bylo umožnit programování v MicroPythonu na čipech FPGA, čímž se

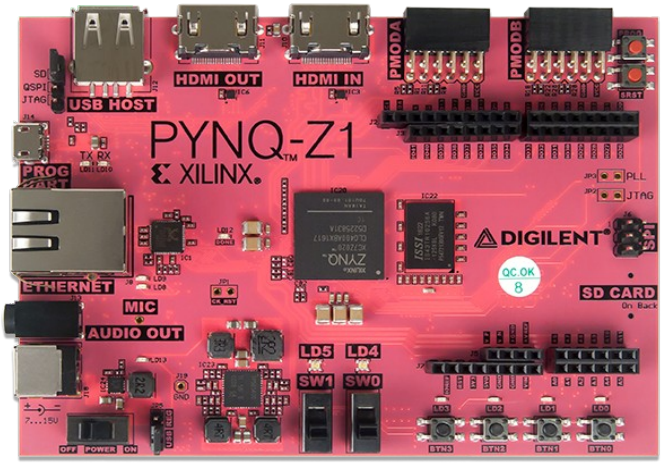
umožnilo akcelerovat různé operace v Pythonu na hardwarové úrovni pomocí FPGA a velice tak urychlit například maticové výpočty, které jsou stěžejní v případě využití neuronových sítí, jak již bylo vysvětleno v úvodu této kapitoly. Vzhledem k absenci dedikovaného CPU se jedná o velmi zajímavou vývojovou desku se snadnou škálovatelností i na podstatně větší FPGA čipy v případě jejich potřeby.

Tab. 2: Specifikace Arty A7 100T [35]

Specifikace pro Arty A7 100T	
Maximální frekvence	450 MHz
RAM	256 MB
Logic slices	101 440
DSP	240
BRAM	5 MB
Napájení	USB či jakýkoliv 7-15V zdroj
Operační systém	Dle Soft CPU
Konektivita	Ethernet, USB-UART, Arduino Shield, 4x Pmod
Úložný prostor	16 MB Q-SPI Flash + microSD slot
Rozměry	76.2 x 101.6 mm
Cena	210 €

2.1.3 PYNQ-Z1

Vývojová deska PYNQ-Z1 byla navržena speciálně k použití s open source frameworkem PYNQ, který značně usnadňuje vývojářům embedded zařízení návrh prototypů pro Xilinx Zynq SoCs bez toho, aniž by nutně museli znát metodiku návrhu designu FPGA. Tento framework umožňuje nahrávat obraz logiky FPGA přímo jako python knihovnu pomocí speciálně navrženého API a výsledné nahrání tedy probíhá podobně jako nahrání softwarové knihovny [6]. Tato vývojová deska je osazena SoC čipem Xilinx ZYNQ XC7Z020, který se skládá z ARM procesoru a FPGA, které jsou vzájemně propojeny velmi rychlou sběrnicí.



Obr. 6: Vývojová deska Pynq-Z1 osazená mikroprocesorem a čipem FPGA

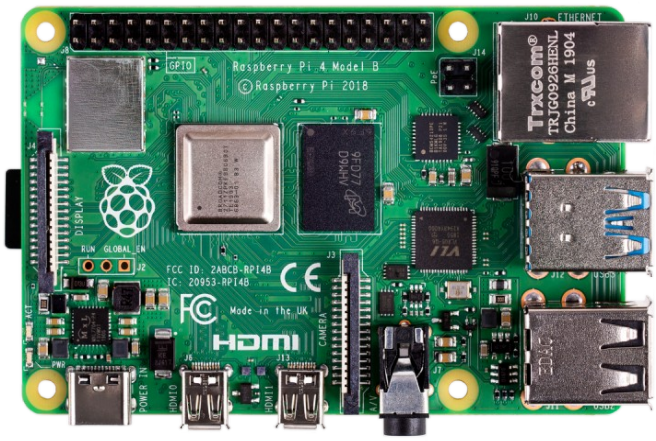
Procesor je architektury Cortex A9 a nese dvě výpočetní jádra. FPGA je ekvivalentní s čipem Artix-7™ avšak je navíc vybaveno dodatečnými vlastnostmi v oblasti propojení s procesorem. Vzhledem k existenci frameworku PYNQ a relativně snadné komunikaci procesoru s FPGA je tato vývojová deska velmi zajímavým kandidátem k rapid prototypingu embedded řešení implementující HW akcelerované výpočty pomocí FPGA.

Tab. 3: Specifikace Pynq-Z1[6]

Specifikace pro Pynq-Z1	
CPU	Cortex A9 dual-core
Frekvence CPU	Až 650 MHz
RAM	512 MB
Logické jednotky	13 300
DSP	220
BRAM	630 KB
Napájení	USB či jakýkoliv 7-15V zdroj
Operační systém	Linux Debian
Konektivita	Ethernet, USB-UART, USB-Host, I2C, SPI, CAN, Arduino Shield, 2x Pmod, 3.5mm audio jack
Úložný prostor	MicroSD slot
Rozměry	88 x 124 mm
Cena	170 €

2.1.4 RASPBERRY PI 4 MODEL B

Raspberry Pi 4 je malý a relativně levný ale výkoný minipočítač. Jeho cena se pohybuje pouze okolo 40€ a ačkoliv se jedná o zařízení s takto nízkou cenou, je tato deska osazena výpočetně výkoným 64 bitovým procesorem ARM Cortex-A72. Hlavní výhodou platformy Raspberry je velice široká komunita uveřejňující nepřeberné množství různých aplikací, které je možné dle vlastních potřeb modifikovat [37].



Obr. 7: Raspberry Pi 4 Model B osazen mikroprocesorem

Další výhodou platformy Raspberry je opravdu široká škála rozšíření a nejrůznějších senzorů, které s touto deskou bezproblémově pracují. Díky USB-C a USB-3.1 je také možné rozšířit výpočetní schopnosti této desky pomocí externího ASIC jako je například Intel Movidius [38] či Google Coral TPU [5].

Tab. 4: Specifikace Raspberry Pi 4 model B [37]

Specifikace pro Raspberry Pi 4 model B	
CPU	Cortex A72 quad-core 64bit
Frekvence CPU	Až 1.5 GHz
RAM	Od 1GB po 4 GB
Napájení	USB či jakýkoliv či 5V/3A zdroj přes GPIO
Operační systém	Raspbian (Linux Debian)
Konektivita	Ethernet, WiFi, Bluetooth, USB (2.0, 3.0, C), 40x GPIO, 2x microHDMI, CSI, DSI, 3.5 mm audio jack
Úložný prostor	MicroSD slot
Rozměry	88 x 124 mm
Cena	39 € za 2GB verzi

2.1.5 NVIDIA® JETSON NANO™

Vývojový kit NVIDIA® Jetson NANO™ je malý, ale velice výkoný minipočítač vyrobený speciálně pro aplikaci neuronových sítí v paralelním režimu pro aplikace jako jsou například klasifikace, detekce objektů, segmentace či zpracování řeči. Tento vývojový kit je nejlevnější a také nejméně výkoný z celé produktové řady Jetson. Jetson NANO™ je osazen GPU, které poskytuje výpočetní výkon 768 GFLOPS a poskytuje tedy dostatečný výkon pro velmi široké spektrum aplikací.

Obr. 8: Vývojová deska NVIDIA® Jetson NANO™ osazená mikroprocesorem a GPU

Tento vývojový kit podporuje SDK NVIDIA JetPack a DeepStream na stejné úrovni jako všechny ostatní produkty Jetson. Tyto SDK jsou zejména výhodné právě pro aplikace strojového vidění a hlubokých sítí na embedded zařízení, jelikož velice zjednodušují obtížnost implementace tohoto druhu neuronových sítí. Díky dedikované GPU jsou také plnohodnotně podporovány frameworky jako jsou Keras, Tensorflow či PyTorch.

Tab. 5: Specifikace NVIDIA® Jetson NANO™ [39]

Specifikace pro NVIDIA® Jetson NANO™	
CPU	Cortex A57 quad-core
Frekvence CPU	Až 1.43 GHz
GPU	NVIDIA Maxwell™ - 128 NVIDIA CUDA jader
RAM	4 GB
Napájení	5V/2A přes micro USB či DC barrel

Operační systém	Linux Debian s SDK JetPack a DeepStream
Konektivita	2x CSI-2, Ethernet, HDMI, display port, 4x USB 3.0, 1x micro USB 2.0, 40x GPIO, I2C, I2S, SPI, UART, M.2 Key
Úložný prostor	MicroSD (min. 16GB bez SDK, min. 32GB s SDK)
Rozměry	69.9 x 45 mm
Cena	87 €

2.1.6 CORAL GOOGLE EDGE TPU

Jak již bylo zmíněno v 1.4 výhradním poskytovatelem TPU čipů je dnes společnost Google, která poskytuje celé portfolio řešení v produktové řadě Coral jejíž část je vyobrazena na obr. 9. Vzhledem k její hardwarové a softwarové otevřenosti v kombinaci s malou prodlevou přenosů mezi CPU a TPU jsem jako vhodný produkt zvolil právě tuto vývojovou desku. Kromě toho je tato deska také vhodná k porovnání s vývojovou deskou osazenou FPGA či GPU, než jak by tomu bylo v případě externího USB modulu.

Obr. 9: Produkty Google Coral. Zleva – Vývojová deska osazená mikroprocesorem a TPU, Akcelerátory do M.2 slotu s čipem TPU, Externí USB-C akcelerátor s čipem TPU a System-on-module s mikroprocesorem a TPU.

Alternativní zvažovanou možností byl i akcelerátor do USB-C slotu v kombinaci s Raspberry. Toto řešení by poskytovalo vyšší výkon na straně CPU než je tomu v případě dedikované vývojové desky Coral a byla by dostupná širší škála periferních modulů z ekosystému Raspberry. Nicméně tato metoda byla zavržena na základě dvou hlavních důvodů, a to kvůli zvýšené době odezvy, která vzniká nutností komunikace po USB, jak je

uváděno například v [40] a problémům s kompatibilitou, jak je řešeno například v [41]. V případě dedikované desky tyto problémy nenastávají, jelikož Google uvolňuje vlastní verze knihoven k jeho modifikovanému operačnímu systému a výsledná implementace je tedy značně ulehčena.

Tab. 6: Specifikace pro Coral Edge Dev Board

Specifikace pro Coral Edge Dev Board	
CPU	Quad core, ARM® 32-bit Cortex-A53, M4F
GPU	Integrated GC7000 Lite Graphics
Frekvence CPU	Až 1.8 GHz
SRAM / RAM	1 Gb
ML akcelerátor	Google Edge TPU coprocessor: 4TOPS (int8); 2 TOPS per watt
Napájení	5 V
Operační systém	Linux Mendel (derivát Debianu)
Konektivita	USB 3.0, USB-C, WiFi, LAN, HDMI, 3.5 mm audio, 40 GPIO
Úložný prostor	8 Gb eMMC + MicroSD slot
Rozměry	88 x 60 mm
Cena	130 €

2.2 VOLBA NEJVHODNĚJŠÍHO HW ŘEŠENÍ PRO RYCHLÉ REÁLNÉ SYSTÉMY

V této práci je cílem aplikovat mělké neuronové sítě architektury HONU na detekci neobvyklých stavů v monitorovaném dynamickém systému. Jelikož takovýto systém může být i velmi rychlý, je nutné zvolit vhodné embedded zařízení, které poskytne jak dostatečný výkon, tak široké možnosti ke zlepšení zde navrženého softwarového řešení. Vzhledem k povaze zadání a skutečnosti, že se v této oblasti nejedná o běžně řešenou problematiku, je nutné provést návrh celého řízení od základu a není možné využít existující frameworky, jako je tomu obvykle v případě hlubokých sítí. Na základě otázek uvedených v 2 jsou zde zvoleny následující ideální požadavky na hledaný hardware, které jsou pro přehlednost shrnuty v tabulce 7 a jejich volba je podrobně prozkoumána na řádcích níže.

Tab. 7: Požadavky na ideální HW dle otázek uvedených v 2

Požadavky na ideální HW dle otázek uvedených v 2	
Velikost RAM / Úložiště	Min. 512 MB RAM / min. 8 GB úložiště
Výpočetní požadavky	FPGA / TPU či velmi výkoné CPU
Napájení	Nižší spotřeba výhodou
Konektivita	Min 8x GPIO / Ethernet / 1x USB
Cílová cena	Do 200€
Využít OS?	Ano
Požadovaný hlavní programovací jazyk	Python
Využité metody strojového učení	Mělké sítě
Nezbytné algoritmy	HONU a jejich adaptace, Learning Entropy
Trénování neuronové sítě	Na cílovém HW
Požadovaná rychlost	Co nejvyšší
Četnost změn SW	Nízká
Možné HW dle možností uvedených v 2.1	Pynq-Z1 (2.1.3), Coral Edge TPU (2.1.6)

Hlavním požadavkem na daný hardware je schopnost efektivně rozběhnout programovací jazyk Python, jelikož v něm bude naprogramována hlavní část řídícího SW a zjednodušená vizualizace probíhajícího řízení. Ačkoliv je Python interpretovaný jazyk a není tedy nejrychlejší, lze jeho kritické funkce velice zefektivnit například pomocí jazyka a kompilátoru Cython, čímž lze dosáhnout téměř rychlostí programovacího jazyka C při minimálních úpravách kódu napsaného v jazyce Python. Mezi největší výhody tohoto jazyka patří plnohodnotný objektový přístup, snadné prototypování, snadné změny v kódu a velice rozsáhlé množství již optimalizovaných open-source knihoven. Kromě toho hlavní výpočty mají být akcelerovány na úrovni výkonných výpočetních čipů, ať se již jedná o FPGA či TPU.

Python jako takový má již ovšem určité minimální požadavky. Jedním z těchto požadavků je operační systém. Ten musí být schopný spustit jeho interpreter. V tomto případě bude nejvhodnější využít operační systém Linux ve vhodné distribuci s ovladači řadičů pro dané embedded zařízení. Stejně tak musí být splněny požadavky na cílový operační systém. Je tedy nutné zvolit vhodné množství operační paměti. Toto množství musí být dostatečné jak pro operační systém, nezbytné služby a program běžící v Pythonu. Ačkoliv absolutně minimální požadavky na paměť RAM pro Linux jsou 8MB [42], jedná se o velice krajní příklad. Bylo by vhodnější poohlédnout se po distribuci určené pro embedded zařízení. Zde se

již dostáváme k minimální hodnotě 128 MB bez grafického rozhraní a 256 MB s grafickým rozhraním [43]. S ohledem na Python, zhruba 64 MB RAM zabere interpretr a jeho další součásti. Proto bude jako nejvhodnější zvolit minimální paměť o velikosti 512 MB, čímž pro vyvinutý program zbude 192 MB RAM, pro případ řešení s grafickým rozhraním. Takto velká operační paměť by měla poskytovat dostatečně velký úložný prostor potřebný ke správnému fungování všech částí systému.

Obdobně jako v případě paměti RAM je nutné zvolit vhodnou velikost úložiště. Výše zmíněná distribuce Linuxu vyžaduje minimálně 2GB bez grafického rozhraní a 10GB s grafickým rozhraním [43]. Python a jeho nezbytné knihovny zaberou v úložišti pouze pár MB. Je však nutné pamatovat na to, že do úložiště bude nutné ukládat různé logy o běhu programu, či nějaká trénovací data. Proto by vhodná velikost úložiště byla 8GB pro distribuci bez grafického rozhraní a 16GB s grafickým rozhraním.

Vzhledem k tomu, že cílem je aplikovat mělké neuronové sítě architektury HONU na detekci neobvyklých stavů v monitorovaném dynamickém systému, je nutné, aby výpočty probíhaly co možná nejrychleji a bylo tak pokryto co nejširší množství systémů pro které bude zde navržené řešení fungovat. Pro toto řešení není vhodné užívat GPU či USB akcelerátory, které potřebují speciální překlad, další ovladače a speciální komunikační rozhraní, čímž se prodlužuje celková doba reakce řídícího systému a úměrně tomu i jeho komplexita. Jako vhodnější řešení se jeví využití integrovaného TPU či FPGA společně s procesorem ARM, jejichž doba odezvy bude podstatně nižší jednak díky vysoké propustnosti sběrnice mezi procesorem a integrovanému TPU / FPGA, tak díky jednoduššímu způsobu překladu instrukcí, které bývá obvykle vyřešeno již výrobcem čipu jako takového.

Klíčovým algoritmem je aplikace HONU (Higher Oder Neural Network) s řídícím systémem. HONU jsou mělkou neuronovou sítí a jejich majoritní výpočetní operací je maticové násobení. Princip HONU a jejich aplikace pro detekci anomálií je detailně vysvětlena v 5 a 6. Trénování neuronové sítě by také mělo být možné provést i na cílovém HW, čímž se opět rozšíří spektrum možných aplikací.

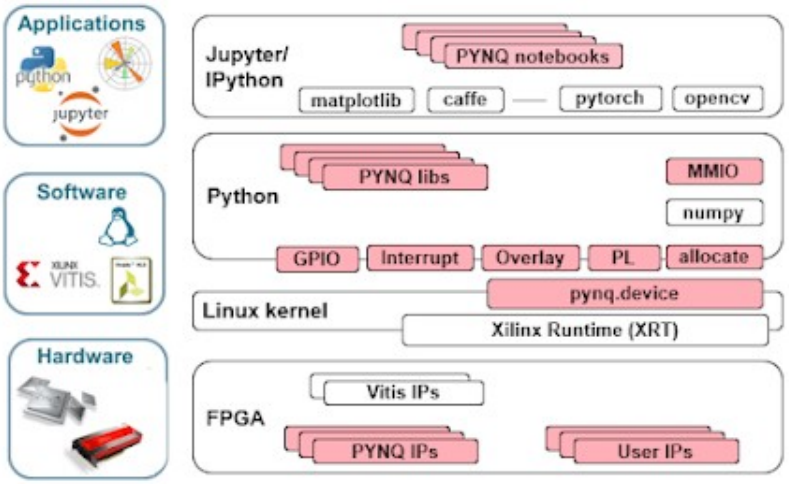
Ačkoliv se váhy neuronové sítě budou měnit a doučovat vždy až na daný systém, řídící systém jako takový se bude měnit velice zřídka. Díky tomu je možné uvažovat i o obtížněji změnitelné části systému, které mohou být ovšem velice optimalizovány a implementovány například na čipy FPGA či TPU.

Konektivita výsledného embedded zařízení musí disponovat alespoň připojením Ethernet, jednak z důvodu podstatně jednodušší správy balíčků Pythonu, tak z možnosti vzdáleného monitorování běžící aplikace. Kromě toho, pokud bude zařízení disponovat rozhraním Ethernet, je možné využít prostředí interaktivního notebooku Jupyter [44] a podstatně tak usnadnit právě vzdálenou správu a monitoring. Další nezbytnou konektivitou je GPIO header s alespoň 8 konektory pro připojení senzoriky a případné zpětné vazby. Posledním požadavkem je alespoň jeden USB konektor. Pokud se totiž ukáže, že je nutné dané embedded zařízení rozšířit o nějakou funkci či úložiště během vývoje řídícího systému, je toto možné provést právě pomocí USB, které poskytuje elegantní univerzální řešení.

Ze zařízení vybraných v 2.1 se dle výše uvedených požadavků na základě otázek z 2 jako nejvhodnější jeví především dvě vývojové desky. Za první vhodnou vývojovou desku byl zvolen PYNQ-Z1, a to především díky frameworku PYNQ, který byl vytvořen pro kombinaci Pythonu a FPGA. Základní princip fungování čipu FPGA byl popsán v 1.3. Tato deska a její framework jsou detailně popsány v 2.1.3 a 2.2.1. Druhá vhodně zvolená vývojová deska je Coral Edge TPU. Tato deska je, jak již z názvu vyplývá, vybavena čipem TPU, který byl speciálně vyvinut pro aplikaci neuronových sítí a jeho základní princip byl popsán v 1.4. Na této desce je TPU plně integrované s procesorem a jeho latence jsou velice nízké. Tato deska je detailněji popsána v 2.1.6 a 2.2.2.

2.2.1 PYNQ-Z1

PYNQ-Z1 je vývojová deska speciálně vyvinuta pro framework PYNQ, což je zkratka pro Python Productivity for Xilinx Zynq. Jedná se o otevřený projekt společnosti Xilinx®, jehož cílem je znatelně usnadnit užití jejich embedded systémů s Zynq SoCs (Systems on Chips). Hlavní výhodou procesorů Zynq a frameworku PYNQ je možnost přímého využití Pythonu ke komunikaci a správě FPGA bez běžného způsobu přehrávání logiky čipu FPGA, která obvykle probíhá pomocí velkých implementačních programů jako je například Vivado HLS [6]. Na Obr. 10 je zobrazen způsob implementace řešení pomocí frameworku PYNQ. Způsob řešení problému obvykle probíhá ve třech hlavních vrstvách a to v následujícím pořadí - hardware, software a aplikace.



Obr. 10: Vývojářská podpora frameworku PYNQ napříč všemi vývojovými vrstvami. Na hardwarové úrovni poskytuje několik základních IP bloků a nástroje ke komunikaci s uživatelskými IP bloky. Na softwarové úrovni poskytuje rozsáhlou knihovnu s velmi schopnou automatizací komunikace s logikou nahranou na čipu FPGA a na aplikační úrovni plně podporuje jak Jupyter notebooky, tak IPython a dále tak usnadňuje implementaci cílového řešení na čipu FPGA (převzato z [6]).

Jednou z hlavních motivací ke vzniku tohoto frameworku je aktuální absence a tedy i částečné řešení dostupnosti open-source IP cores[1], které jsou základními stavebními bloky při řešení daného problému na úrovni hardwarové vrstvy. Výsledkem pak v případě FPGA je takzvaný bitstream[2]. I dnes jsou bohužel designy IP cores obvykle střeženým firemním tajemstvím a je jen velice obtížně najít takové, které by splňovaly moderní požadavky na implementace a požadovaný účel. Mezi stránky, které poskytují jak open-source, tak i komerční zdrojové kódy k IP cores patří především [45]. Tato stránka byla založena v roce 1999 a dnes čítá více než 327 tisíc IP cores. Je však nutné dodat, že většina open-source IP cores byla přidána před rokem 2010 a velké množství z nich zůstalo ve své vývojové části se značným množstvím problémů. Druhou stránkou, a zde již čistě s open-source IP cores, je [46], kterou spravuje FOSSi Foundation [47]. Cílem této stránky je poskytnout velké množství open-source IP cores přímo od vývojářů a stát se jakousi obdobou GitHubu pro vývojáře se specializací v ASIC a FPGA. Jelikož se ovšem jedná o velmi mladý projekt, je stále velice obtížné najít prakticky jakýkoliv požadovaný IP core či verifikaci o jeho spolehlivosti. Vzhledem k aktuálnímu stavu bylo tedy nutné vytvořit IP blok pro cílový bitstream zcela od začátku. Popis hardwarově orientovaných jazyků a postup implementace od softwaru až po hardware je uveden v 3.

1 IP core (Intellectual Property) je opakovatelně použitelná jednotka logického obvodu se specifickým určením. Za IP core lze považovat například řadič Ethernetu či CPU.

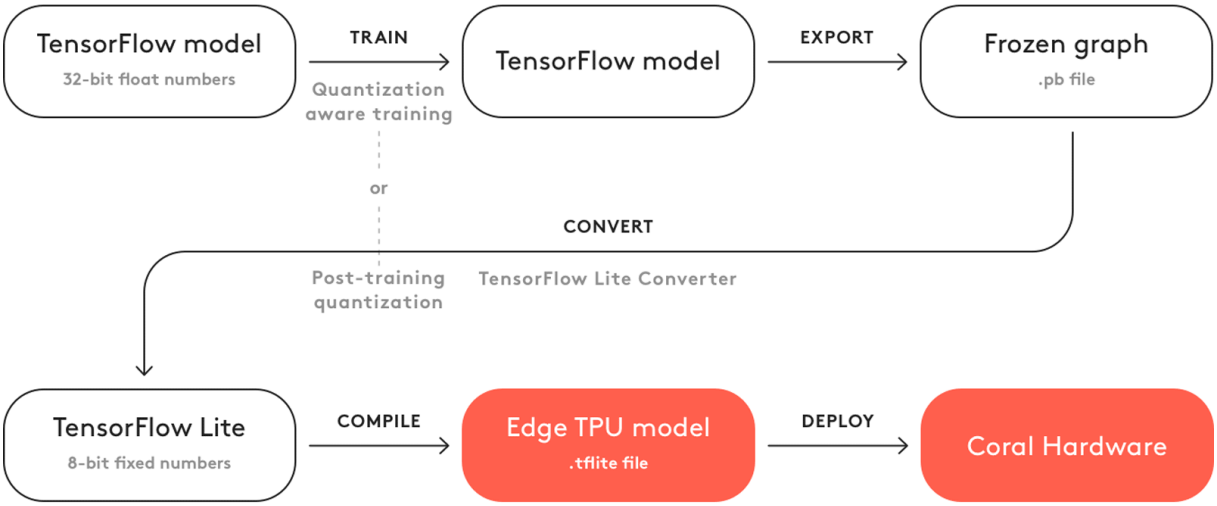
2 Bitstream je soubor, který obsahuje programovací informace o nastavení všech hradel na specifickém FPGA.

Softwarovou vrstvu je nezbytné vytvořit dle navržené hardwarové vrstvy, čímž vznikne jakási přidružená knihovna, která dokáže plnohodnotně komunikovat s FPGA. PYNQ k tomuto účelu poskytuje pokročilé API, které umožňuje přímou komunikaci s čipem FPGA a knihovna jako taková je obvykle vytvářena k dalšímu usnadnění budoucí aplikace na základě metod tohoto frameworku, aby byl koncový uživatel zcela osvobozen od znalosti adres paměti apod. Hlavní výhodou tohoto frameworku v softwarové části je automatické přeložení nahraného bitstreamu do pythoního objektu se kterým se následně podstatně snáze pracuje.

Aplikační vrstva už obsahuje programovou logiku jako takovou. Probíhá zde hlavní programová část využívající hardwarovou akceleraci výpočtů pomocí FPGA s kterým je komunikováno skrze softwarovou vrstvu. Framework PYNQ je postaven okolo aplikací běžících v Jupyter Notebooku [44], ačkoliv to není nezbytné, pro prototyping se jedná o velmi výhodné řešení.

2.2.2 CORAL EDGE TPU DEV BOARD

Coral Edge TPU je vývojová deska, jak již bylo zmíněno v 2.1.6, navržená společností Google. Tato deska je vybavena speciálním ASIC čipem uzpůsobeným ke zpracování tenzorů k aplikaci metodik strojového učení. Tento čip je ovšem omezen na zpracování pouze 8bit celočíselných hodnot [48]. Je tedy nutné zvolit vhodnou metodiku trénování i vhodné vstupy. Tato omezení jsou již do jisté míry vyřešena pomocí frameworku TensorFlow. Tento framework řeší toto omezení pomocí kvantifikace jak vah natrénovaného modelu tak vstupů a výstupů [8]. Metodika vytvoření modelu pro Edge TPU pomocí TensorFlow je graficky znázorněna na Obr. 11.Vzhledem ke skutečnosti, že použitou architekturou neuronové sítě mají být HONUs 5.1, je nutné napsat vlastní vrstvu FrameWorku TensorFlow pro tuto architekturu tak, aby byla kompatibilní jak s běžným FrameWorkem, tak umožňovala kvantifikaci vah pro převod do jeho Lite verze.



Obr. 11: Základní postup pro vytvoření modelu pro Coral Edge TPU. Nejprve je nutné vytvořit model pomocí knihovny TensorFlow, který se na výkoném HW natrénuje s ohledem na následnou kvantifikaci vah. Natrénovaný model se poté převede na verzi TensorFlow Lite, která na rozdíl od běžného modelu nepracuje s 32bit float ale pouze s 8bit int. Tato Lite verze již může být zkompilována a nahrána do paměti čipu TPU (převzato z [8]).

Alternativním možným řešením je implementace neuronové sítě bez TensorFlow a to za využití knihovny JAX [49]. Tato knihovna poskytuje rozhraní, pomocí nějž je možné provádět většinu matematických operací podobnou syntaxí jako je tomu v případě matematické knihovny Numpy, která je určena pro výpočty na CPU, přímo na vybraných GPU či TPU bez nutnosti vytvoření vlastní metodiky překladu pro tyto specifické čipy. Jak se ale bohužel ukázalo, tato knihovna je v tuto dobu velice rozvinutá v oblasti implementací výpočtů na GPU a pro čipy TPU je teprve ve svých začátcích. Pomocí TPU prozatím není možné provádět například maticové násobení, které je pro aplikaci strojového učení klíčové. Tato možnost je tedy v tuto dobu jen obtížně využitelná. Jedná se však o zajímavý projekt, na jehož konci může být velmi slibné řešení.

2.2.3 ZVOLENÉ CÍLOVÉ EMBEDDED ZAŘÍZENÍ

Na základě informací získaných z rešerše uvedené v této kapitole bylo pro cíle této práce zvoleno embedded zařízení PYNQ-Z1. Toto zařízení poskytuje pro cílovou aplikaci dostatečně velkou operační paměť s úložištěm v podobě SD karty a je osazeno čipem FPGA, který umožňuje značné urychlení vybraných matematických operací či provádět určité algoritmy v téměř reálném čase. Díky přítomnosti čipu FPGA je toto řešení také nejvhodnější s ohledem na budoucí možné aplikace a jejich optimalizace až na hardwarovou úroveň. Vlastnosti čipu FPGA umožňují i v budoucnosti dále optimalizovat výpočetní výkon či

upravovat funkcionality softwaru a hardwaru, což v případě čipů typu ASIC je možné jen velmi obtížně. Výhodou této desky proti Google Coral TPU je také možnost využití čísel s desetinou čárkou i bez kvantifikace a usnadnit tak adaptaci neuronové sítě. Jelikož programování FPGA není ani dnes zcela běžné, je v následující kapitole vysvětlen princip hardwarově popisných jazyků a je zde popsána základní metodika hardwarového programování, která je ovšem díky pokročilým softwarovým nástrojům značně usnadněna.

3 HARDWAROVĚ POPISNÉ JAZYKY

Metodika fungování libovolných elektronických obvodů může být popsána pomocí hardwarově popisných jazyků (HDL – Hardware Description Language). HDL je dnes ovšem nejčastěji využíván k popisu digitálních logických obvodů. Program napsaný v HDL je pomocí implementačních softwarů, jako je například Vivado HLS [50], dále překládán na nižší úroveň do specifikací již fyzické interpretace skutečného obvodu. Tato fyzická interpretace HDL programu poté může být umístěna na foto-litografickou masku a nebo nahrána do čipu FPGA [51].

HDL jazyky jsou podobné běžným programovacím jazykům, jakými jsou například jazyky C či Java. Mají ovšem některé klíčové rozdíly, které vyžadují rozdílné řešení problému, než běžné programovací jazyky. První takový rozdíl je, že HDL konkrétně zahrnuje pojem času, zatímco běžné programovací jazyky nikoliv. Druhý klíčový rozdíl je, že programový kód HDL je syntetizován do fyzicky realizovatelného modelu obvodové implementace návrhu, kde je přesně definován tok dat a časování celého obvodu a kód programovacího jazyka je kompilován do objektového kódu specifického pro daný čip. Třetím klíčovým rozdílem je, že kód napsaný v HDL je souběžný, což znamená, že každý proces v popisu běží paralelně a probíhá nezávisle. Běžné programovací jazyky jsou jednovláknové a kód se spouští postupně. Posledním klíčovým rozdílem je skutečnost, že HDL popisuje reálný hardware, což vylučuje použití jakékoliv dynamické struktury, která je běžně využívána v programovacích jazycích [11].

Obdobně jako v případě programovacích jazyků existuje i jazyků HDL velké množství. Ačkoliv klíčové vlastnosti jsou stejné napříč celým jejich odvětvím, na metodické a logické úrovni se často značně liší. Mezi dnes velmi často užívané HDL jazyky patří VHDL a

Verilog. Tyto jazyky je obvykle možné syntetizovat z jazyků na vyšší úrovni jako je například C či C++ díky čemuž je jejich implementace značně usnadněna.

3.1 HARDWAROVĚ POPISNÝ JAZYK: VHDL

Jazyk VHDL, neboli Very High Speed Integrated Circuit Hardware Language byl vyvinut na základě požadavku ministerstva obrany USA (DoD – Department of Defence) v roce 1983. Původní smysl tohoto jazyka bylo dokumentovat chování ASIC čipů, které DoD odebíralo do svého vybavení od subdodavatelů. Dalším logickým krokem vývoje tohoto jazyka tedy bylo vyvinutí simulátoru, který dokáže přečíst a správně simulovat chování modelu ASIC čipu popsaného pomocí VHDL. Vzhledem k tomu, že VHDL obsahuje přesný model chování a bylo možné jej simulovat, začali vznikat nástroje k logické syntéze, které překládali VHDL zpět na fyzické implementace logických obvodů [12].

Tento jazyk je navržen tak, aby podporoval všechny úrovně abstrakce používané pro návrh jak analogických, tak logických obvodů. Umožňuje tedy popsat obvod na hradlové, RTL[3] i algoritmické úrovni. Jedná se o silně typovaný jazyk, který má prostředky pro popis paralelismu, konektivity a explicitní vyjádření času. Popis číslicových zařízení a jejich jednotlivých částí probíhá pomocí dvou komponent:

- entita – definice rozhraní komponenty (pouze určení vstupů a výstupů, nikoliv funkce)
- architektura – určuje funkci a chování (složeno ze dvou částí – deklarační a příkazové)

Příklad syntaxe jazyka VHDL s importováním knihovny, definicí entity a její architektury je uveden na Alg. 1 [10].

3 RTL neboli úroveň přenosu registrů (Register-Transfer Level) představuje v návrhu digitálních obvodů abstrakci návrhu, která modeluje tok digitálních signálů mezi hardwarovými registry a logické operace prováděné těmito signály.

```
-- ( toto je VHDL komentář)
/*
   toto je blokový komentář ve VHDL
*/
-- import std_logic z IEEE knihovny
library IEEE;
use IEEE.std_logic_1164.all;

-- toto je entita
entity BRANAAND is
 port (
   I1 : in std_logic;
   I2 : in std_logic;
   O  : out std_logic);
end entity BRANAAND;

-- toto je architektura
architecture RTL of BRANAAND is
begin
  O <= I1 and I2;
end architecture RTL;
```

Alg. 1: Příklad syntaxe jazyka VHDL s definováním entity a její architektury. Zde uvedený kód definuje funkci brány AND.

3.2 HARDWAROVĚ POPISNÝ JAZYK: VERILOG

Jazyk Verilog, někdy nazývaný Verilog HDL byl vytvořen Philem Moorbym a Prahabu Goelem v roce 1985 jako jazyk pro modelování hardwaru s cílem značně zlepšit původní omezení VHDL v oblasti logické syntézy a simulací hardwaru. Autoři jazyka Verilog se snažili vytvořit takový hardwarový jazyk, který by se co nejvíce podobal programovacímu jazyku C, který byl tehdy již velmi hojně využíván v softwarovém vývoji. Obdobně jako jazyk C je tedy i Verilog citlivý na velikost písmen a má základní preprocesor, díky čemuž je v tomto jazyku možné využívat klíčová slova (if/else, for, while, case, atd.), která upravují běh programu obdobně jako v jazyce C. Na rozdíl od jazyka C je ovšem nutné vždy pevně definovat bitovou šířku a nepodporuje složené datové typy, ukazatele ani rekurzivní podprogramy [13].

Návrh ve Verilogu se skládá z hierarchie modulů. Každý modul v sobě izoluje návrhovou hierarchii a komunikuje s ostatními moduly pomocí deklarovaných vstupů, výstupů a obousměrných portů. Uvnitř modulu mohou být jak sekvenční, tak paralelní bloky či instance jiných modulů. Nicméně všechny bloky uvnitř modulu jsou vždy spuštěny paralelně. Na rozdíl od jazyka VHDL ovšem verilog nepodporuje koncept knihoven a je tedy nutné si vždy vše napsat originálně. Příklad syntaxe jazyka Verilog pro bistabilní klopný obvod s časovou závislostí je uveden na Alg. 2.

```
// toto je Verilog komentář
// Blokový komentář neexistuje
// Import knihoven neexistuje

// Toto je modul
module toplevel(clock,reset);
 input clock;
 input reset;

 reg flop1;
 reg flop2;

 always @ (posedge reset or posedge clock)
  if (reset)
   begin
    flop1 <= 0;
    flop2 <= 1;
   end
  else
   begin
    flop1 <= flop2;
    flop2 <= flop1;
   end
endmodule
```

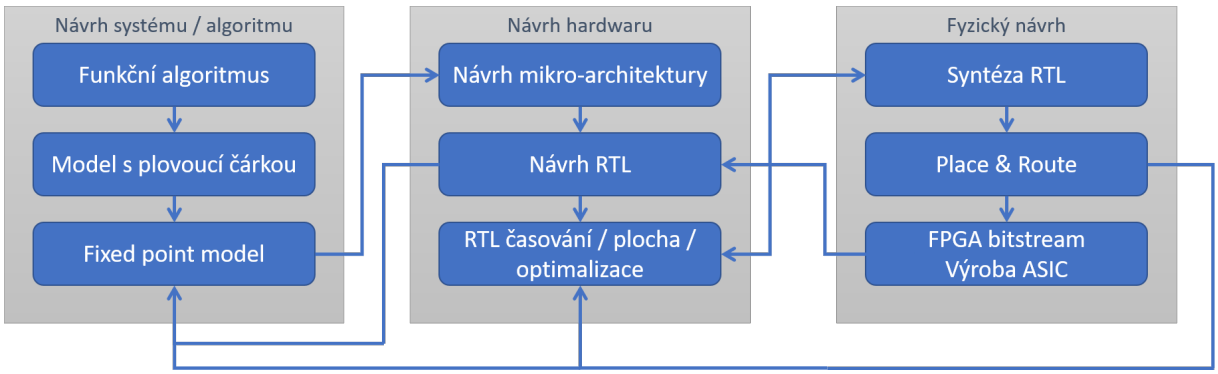
Alg. 2: Příklad syntaxe jazyka Verilog pro bistabilní klopný obvod s časovou závislostí.

3.3 METODIKA HARDWAROVÉHO PROGRAMOVÁNÍ

Původním plánem pro vytvoření hardwarového designu bylo využití Python knihovny myHDL. Velice brzy se však ukázalo, že interpretovaný jazyk vysoké úrovně jako je Python není vhodným nástrojem a vygenerovaný výstup z jazyka Python do Verilogu obsahuje relativně velké množství chyb, kvůli kterým je překlad na hardwarovou úroveň možný jen velice komplikovaně. Kromě toho je i kód v Pythonu nutné psát přesně dle syntaxe Verilogu bez jakýchkoliv knihoven a tím prakticky veškeré výhody Pythonu mizí. Místo toho bylo v této práci přikročeno k běžnějšímu způsobu vývoje a to pomocí jazyka C a C++ a jejich následnému překladu do Verilogu ve vývojovém prostředí Vitis a Vivado HLS, jenž jsou podrobněji popsány v 3.3.1 a 3.3.2.

Vzhledem ke klasickému přístupu k návrhu designu FPGA čipu byl v této práci využit i běžný implementační postup celého návrhu, který je graficky znázorněn na Obr. 12. V postupu návrhu jsou přirozeně některé cesty zpět pro lepší iteraci výsledného návrhu. Jeden z hlavních iterativních cyklů je mezi fixed-point modelem, návrhem mikroarchitektury a návrhem RTL. Fixed-point model a RTL obvykle potřebují několik cyklů, než mohou být úspěšně realizovány na úrovni RTL. Toto je obvykle nezbytné, protože implementace modelu algoritmu do RTL není snadná a i fixed-point model může obsahovat struktury, jejichž

fyzická reprezentace není vhodná či dokonce možná pro RTL implementaci. Návrháři algoritmu a hardwaru v této části obvykle intenzivně spolupracují dokud neodladí dokonalý FP model, jehož mikroarchitektura může být úspěšně implementována do hardwaru [52].



Obr. 12: Běžný postup při návrhu digitálního logického obvodu. Vždy se začíná od návrhu funkčního algoritmu, který je běžně napsán v jazyce C/C++ a pak následuje jeho překlad až na hardwarovou úroveň. Obvykle je nutné se z různých vývojových částí vrátit až na začátek a původní algoritmus vhodně upravit. Použitý implementační program Vivado poskytuje velmi rozsáhlou automatizaci napříč celým vývojem a značně tak celý tento proces usnadňuje (převzato a upraveno dle [52]).

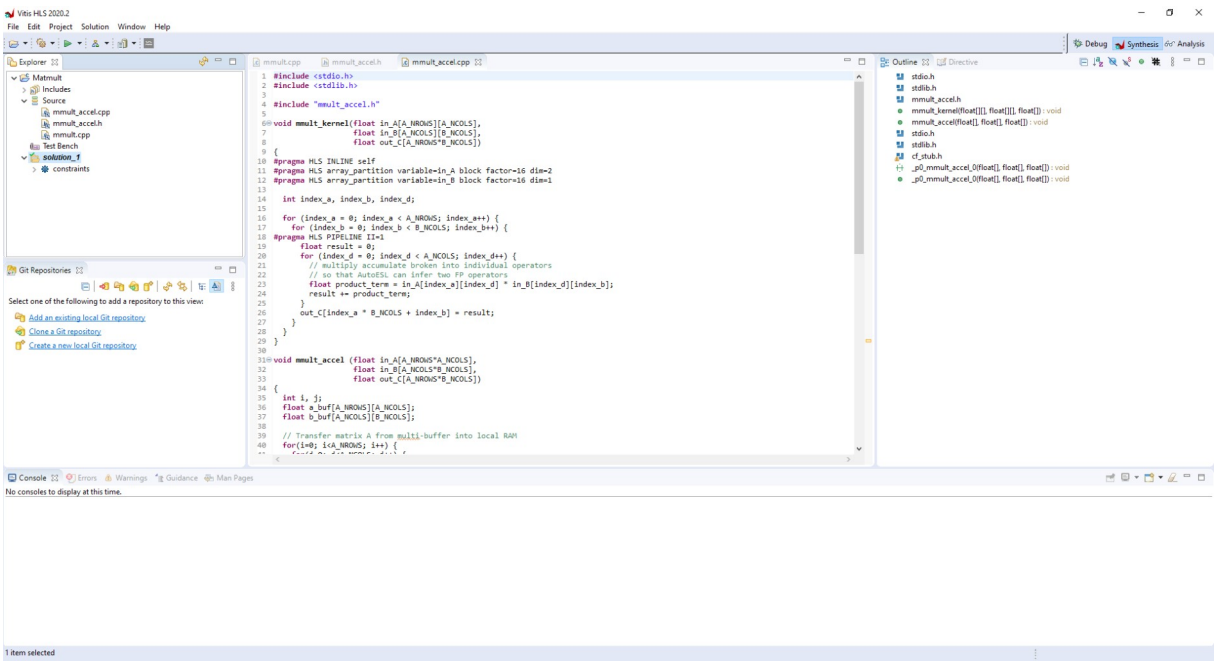
Dalším důležitým krokem během návrhu hardwarového designu je část RTL časování, plochy a optimalizace. Tato část vyžaduje obvykle zpětnou vazbu z reálného hardwaru, jelikož velice silně závisí na polovodičové technologii a typu zařízení cílového FPGA čipu. Tuto zpětnou vazbu poskytuje z velké části syntéza RTL a place & route, jedná se totiž již o návrh fyzického rozložení logického obvodu. Je běžné, že požadavků na RTL časování není dosaženo i po několika iteracích, což je obvykle zapříčiněno příliš velkým počtem logických úrovní na jeden časový puls bez mezi registru obdržené hodnoty, příliš komplexní logikou s velkým množstvím simultánních vstupů nebo příliš velkou logickou strukturou, kterou je velmi obtížně umístit na cílový FPGA čip. Problémy s RTL časováním se obvykle řeší úpravou původního fixed-point modelu, zjednodušením algoritmu či snížením přesnosti nebo omezením bitové šířky. Z pravidla platí, že čím vyšší frekvence požadujeme, tím obtížnější je dosáhnout stabilního RTL návrhu.

Po úspěšné syntéze návrhu a vytvoření bitstreamu pro FPGA je vždy nutné provést dostatečný počet testů spolehlivosti. Tyto testy obvykle porovnávají softwarový výpočet daného algoritmu s výpočtem akcelerovaným pomocí hardwaru během různých scénářů, v kterých se může dané zařízení nacházet. Pokud se objeví nějaké zásadní problémy, je nutné jít zpět do RTL a dané chyby opravit [52].

Vzhledem k tomu, že zcela klasický přístup je velmi zdlouhavý a komplikovaný, bylo v této práci využito překladače jazyka C a C++ do Verilogu od společnosti Xilinx. Během překladu bylo velmi důležité nastavit vhodné metodiky překladu, takzvané pragmy 8.3, které značně měnily výsledný výpočetní výkon hardwarového řešení. K syntéze Verilogu do RTL bylo využito také automatizované optimalizace programu Vivado. Tento druh optimalizace nikdy nedosáhne naprosto ideálního výsledku, nicméně značně snižuje náročnost na znalost hardwarového programování a tak i snižuje potřebný vývojový čas.

3.3.1 NÁSTROJ SYNTÉZY: VITIS HLS

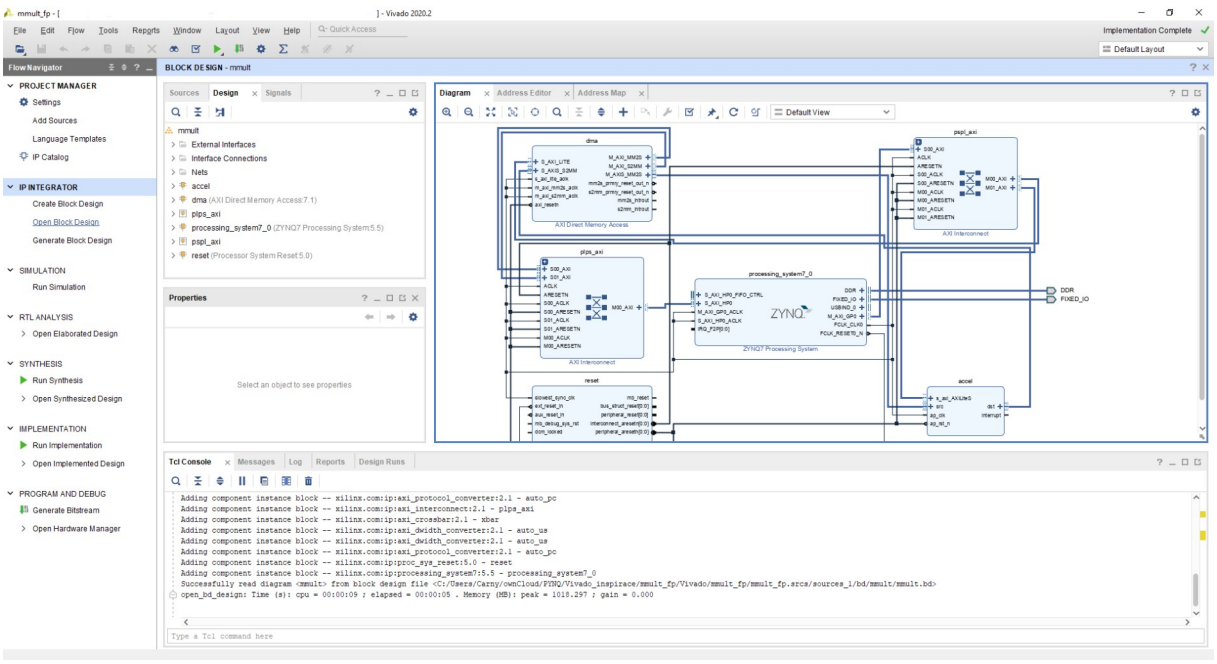
Vitis HLS od společnosti Xilinx je nástroj vysokoúrovňové syntézy, který umožňuje překlad funkcí jazyků C, C++ a OpenCL do požadovaného hardwarového jazyka a dokáže provést syntézu až na RTL úroveň. Tento nástroj automatizuje velké množství modifikací kódu, které jsou nezbytné k implementaci a optimalizaci C/C++ kódu do programovatelné logiky. Kromě toho má tento program i bohaté rozhraní ke snadné implementaci nezbytných direktiv překladu k optimálnímu rozkladu logiky napsané v jazyce C/C++ nebo OpenCL. Kromě bohaté podpory při překladu, obsahuje tento software i značné množství knihoven sloužících například ke komunikaci s hardwarovým komunikačním rozhraním AXI či poskytuje základní matematickou knihovnu, která obsahuje například definici struktury čísla s desetinou čárkou či goniometrické funkce [50].



Obr. 13: Uživatelské prostředí Vivado HLS, které slouží k syntéze jazyka C do Verilog či VHDL. Vlevo se nachází struktura projektu, uprostřed funkce napsaná v jazyce C/C++ a napravo specifikace direktiv překladu.

3.3.2 NÁSTROJ HARDWAROVÉHO NÁVRHU: VIVADO

Vivado Design Suite je softwarový nástroj společnosti Xilinx pro syntézu a analýzu HDL návrhů, který nahrazuje od roku 2012 starý vývojový software Xilinx ISE. Oproti starému softwaru poskytuje tento software rozsáhlou automatizaci jak v oblasti optimalizace návrhu, tak v oblasti syntézy RTL až do výstupního bitstreamu. Tento software také obsahuje základní knihovnu s ihned implementovatelnými IP bloky jako jsou například řadiče Ethernetu, HDMI či přímé propojení logiky FPGA do systémové paměti.



Obr. 14: Uživatelské prostředí Vivado, které slouží k implementaci a syntéze jazyků Verilogu, VHDL či RTL do bitstreamu pro FPGA.

V tomto softwarovém nástroji tedy probíhá jak návrh propojení jednotlivých IP bloků na úrovni logických propojení, tak jejich syntéza do RTL, implementace a generování výstupního bitstreamu pro FPGA.

Instrukce k instalaci prostředí Vitis a Vivado jsou uvedeny v příloze 12.1.

4 VOLBA HLAVNÍHO PROGRAMOVACÍHO JAZYKA

Hlavní programovací jazyk, který by nejvíce vyhovoval naprogramování vrstvy HONU a byl podporován frameworkem PYNQ pro účely detekce pomocí metody learning entropy jsou Python a C++. Oba tyto jazyky jsou objektově orientované a jsou podporovány daným frameworkem. Hlavním důvodem pro jazyk C++ je jeho vyšší rychlost oproti interpretovanému Pythonu. V případě Pythonu je ovšem mnohem rychlejší tvořit nové funkce a odpadá nutnost silného typování.

Pro zpracování mé diplomové práce jsem zde zvolil jazyk Python. Zdůvodnění je uvedeno v bodech níže:

- Neuronové sítě lze naprogramovat zcela objektově, tedy tak aby práce s nimi byla jednodušší a co nejvíce automatizovaná. Je zde také možnost použití širokého množství knihoven k usnadnění práce s neuronovými sítěmi.
- Ačkoliv je Python výpočetně pomalejší než C++, všechny hlavní výpočty budou probíhat na hardwarové úrovni a Python se pouze bude starat o logiku programu.
- Python se pro mne stal hlavním programovacím jazykem a využívám ho i pro své další projekty. Objektově naprogramované knihovny neuronových sítí s akcelerací pomocí hardwaru budu tedy moci použít i v budoucnosti.
- Python má široké možnosti konektorů na další programy i v jiných jazycích, včetně C++. Budu tedy moci zde vyvinutý modul použít i pro jiné aplikace v mých budoucích projektech, které nebudou muset být nutně v Pythonu.

Jako editor jazyku Python jsem si vybral program Spyder, jelikož je optimalizovaný pro kód aplikovaný na obtížné výpočty a navíc je také nainstalován během instalace balíčku Anaconda. Kromě programu Spyder budu využívat k drobnější editaci Jupyter Notebook, který běží přímo na embedded zařízení. Díky tomu je cílová aplikace mnohem snadnější a rychlejší. V případě potřeby lze tento notebook využít i k vizualizaci aktuálně probíhající detekce.

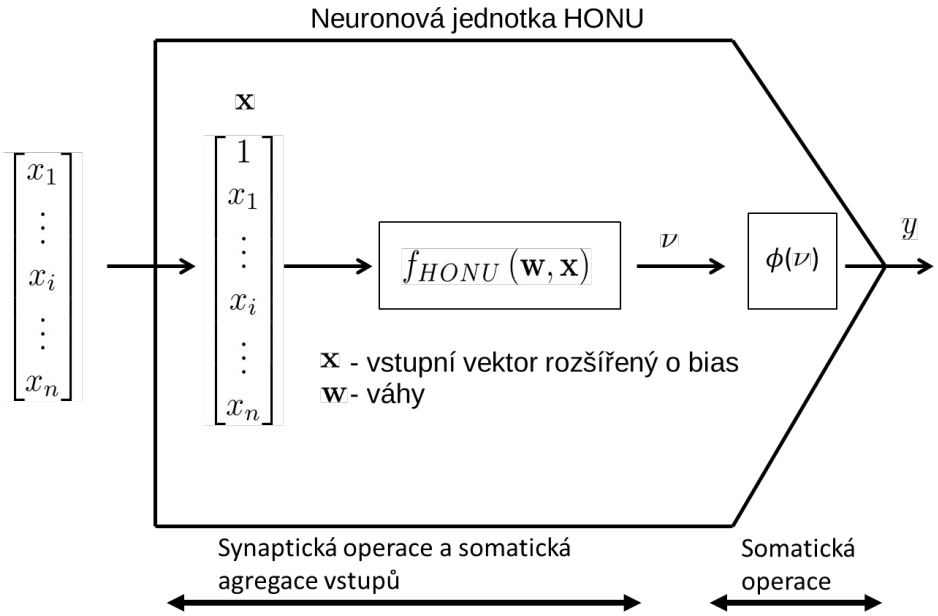
Instrukce k instalaci použitého prostředí jsou uvedeny v příloze 12.1.

5 NEURONOVÉ ARCHITEKTURY

V této práci je využito k aplikaci Learning entropy na embedded zařízení několik různých neuronových architektur, které se dají rozdělit do dvou skupin. První skupinou je lineární neuronová jednotka (LNU) a druhou jsou HONU s druhým a třetím stupněm polynomu, tedy kvadratické (QNU) a kubické (CNU) neuronové jednotky. LNU lze pokládat za HONU prvního stupně a proto je zde zahrnuto společně s HONU. Dále uvedený text předpokládá u čtenáře obecnou znalost principu fungování neuronových sítí, která byla detailněji popsána a vysvětlena v mé dřívější práci [53], jejíž teoretická část ohledně HONU byla založena především na [2]. V následujících kapitolách je princip této architektury stručně zopakován.

5.1 NEURONOVÉ JEDNOTKY VYŠŠÍCH STUPŇŮ, HONUS

Neuronové jednotky vyšších stupňů vycházejí z vrstevnatých neuronových sítí, což znamená, že mají pouze dopředné spoje. Tyto jednotky někdy bývají označovány také jako HONNU (Higher Order Nonlinear Neural Units). Nelineární neuronové modely HONU jsou v přímé analogii k Taylorovu polynomiálnímu rozvoji. HONU má volitelný řád polynomu a proto lze nastavovat kvalitu nelineární aproximace.



Obr. 15: Schéma neuronové jednotky HONU, překresleno a upraveno dle [2]

I dnes jsou HONU využívány maximálně do polynomu 3. stupně. S vyššími stupni polynomu a množstvím vstupů roste exponenciálně složitost výpočtů a tedy i časová náročnost. V této práci jsou použity modely HONU LNU, QNU a CNU. LNU má na rozdíl od QNU a CNU lineární synaptickou operaci. Všechny tři tyto modely však mohou být vystiženy pomocí Obr. 15. Rozdíl mezi jednotlivými modely HONU spočívá v jejich agregační funkci f_{HONU}.

Pro LNU je agregační funkce lineární a rovna

$$f_{HONU} = \sum_{i=0}^{n} x_i \cdot w_i, \tag{5.1}$$

pro QNU je nelineární a rovna

$$f_{HONU} = \sum_{i=0}^{n} \sum_{j=i}^{n} x_i \cdot x_j \cdot w_{i,j}, \tag{5.2}$$

a pro CNU je též nelineární a rovna

$$f_{HONU} = \sum_{i=0}^{n} \sum_{j=i}^{n} \sum_{k=j}^{n} x_i \cdot x_j \cdot x_k \cdot w_{i,j,k}. \tag{5.3}$$

5.1.1 MODIFIKACE HONUS

V této podkapitole je popsána metoda dle [2], se kterou lze implementovat neuronové jednotky HONU jednodušeji a především touto metodou snížit nutné výpočetní časy. Hlavní výhodou této metody je převod N-dimenzionálních matic do vektorů a pracujeme tedy jen s vektory. Tím lze značně snížit výpočetní nároky na HW, který neposkytuje hardwarovou akceleraci maticových výpočtů, jako je například procesor. Jelikož se nemusí pracovat právě s N dimenzionálními maticemi, které při větších rozměrech ubírají mnoho operační paměti a násobení jednotlivých prvků matic je nahrazeno jednou operací – vektorovým násobením.

Synaptická operace pro neurony typu HONU má tvar

$$f_{HONU}(\mathbf{w}, \mathbf{x}), \tag{5.4}$$

kde, $\mathbf{x}$ je vstupní vektor a $\mathbf{w}$ je matice vah. S ohledem na výpočetní rychlost je obvykle výhodné provést synaptickou operaci maticově. Což v případě neuronové jednotky QNU, polynomu druhého stupně, lze vyjádřit jako

$$y_n = \sum_{i=0}^{n} \sum_{j=i}^{n} x_i \cdot x_j \cdot w_{i,j}, \tag{5.5}$$

a kde matice vah w je 2D a má tvar

$$\mathbf{W} = \begin{bmatrix} w_{0,0} & w_{0,1} & \cdots & w_{1,n} \\ 0 & w_{1,1} & \cdots & w_{2,n} \\ \vdots & \ddots & \ddots & \vdots \\ 0 & \cdots & \cdots & w_{n,n} \end{bmatrix}. \tag{5.6}$$

V matici (5.6) je využit pouze horní trojúhelníková matice, tedy jen polovina všech prvků, jelikož spodní polovina je vždy nulová. Aby se tedy předešlo náročným výpočtům neuronového výstupu sumací součinu prvků vstupního vektoru a prvků v matici vah (5.5), je převeden vstupní vektor a matice vah do dvou tzv. dlouhých vektorů

$$row\,(\mathbf{w}) = row_{\mathbf{w}}, \tag{5.7}$$

$$row^r\,(\mathbf{x}) = row_{\mathbf{x}}, \tag{5.8}$$

kde r je řád polynomu.

Má-li vstupní vektor n vstupů, potom je délka l dlouhého vektoru $row_{\mathbf{w}}$ a $row_{\mathbf{x}}$ výsledkem sumace synaptické operace 5.4 a je rovna hodnotám uvedeným v Tab. 8.

Tab. 8: Délka dlouhého vektoru dle stupně polynomu

Polynom 1. stupně (LNU)	**Polynom 2. stupně (QNU)**	**Polynom 3. stupně (CNU)**
$l^{r=1} = n$	$l^{r=2} = \frac{n \cdot (n+1)}{2}$	$l^{r=3} = \frac{n \cdot (n+1) \cdot (n+2)}{6}$

N-Dimenzionální matici vah tedy lze přepsat do 1D vektoru o délce l^r. Nový vektor vstupů bude mít tvar pro QNU

$$row^{r=2}(\mathbf{x}) = \left[x_0^2 \; x_0 x_1 \; \cdots \; x_i x_j \; x_n^2\right], \tag{5.9}$$

kde budou jednotlivé součiny vstupního vektoru seřazeny stejným pravidlem, ze kterého vyplývá sumace z (5.5).

Neuronový model HONU lze tedy na základě předešlé implementace vyjádřit obecně pro libovolný stupeň polynomu jako

$$y_n = row^r(\mathbf{x}) \cdot row(\mathbf{W})^T = row^r(\mathbf{x}) \cdot col(\mathbf{W}), \tag{5.10}$$

kde $col()$ analogicky představuje operaci vytvoření dlouhého sloupcového vektoru. Tuto modifikaci jsem implementoval v aplikaci pro rychlý výpočet HONU pomocí CPU pro porovnání s maticovým přístupem čipu FPGA.

V knihovně nn_accel v souboru honu.py jsou funkce, pomocí kterých lze využívat tuto implementaci na cílových embedded zařízeních a ve skriptu examples jsou vzorové příklady použití neuronových sítí s touto implementací, viz příloha (kap. 12).

5.2 VYBRANÉ METODY ADAPTACE HONU

Aby bylo možné využívat modely neuronových sítí, je nutné nejdříve naučit závislosti vstupů na výstupu pomocí trénovacích dat. V této kapitole je představeno několik nejznámějších učících algoritmů s učitelem.

5.2.1 GRADIENT DESCENT

Jednou možností k natrénování neuronových sítí je metoda klesání podle gradientu (Gradient descent, GD). K tomu musíme být schopní vypočítat gradient G ztrátové funkce vzhledem ke každé váze w_{ij}. Tento gradient nám říká, jak malá změna právě té váhy nám ovlivní celkovou chybu E. Jde tedy o nalezení minima této chybové funkce [53]. Tato metoda patří do skupiny takzvaného online neboli sample učení a váhy se aktualizují vždy po každém novém vzorku. Výchozí rovnice pro výpočet gradientu jsou

$$E = \sum_{p} E^{p}, \;\; E^{p} = \frac{1}{2} \sum_{i} \left(y_{t}^{p} - y_{n}^{p}\right)^{2}, \tag{5.11}$$

kde p označuje aktuální trénovací bod, nikoliv umocnění a i je počet všech vah. Po jejich odvození pro neuronovou síť, které je uvedeno v [54], získáme rovnici pro adaptaci jednotlivých vah i kterou lze zapsat včetně chybové funkce jako

$$w_{i}\left(k+1\right) = w_{i}\left(k\right) - \frac{1}{2} \cdot \mu \cdot \frac{\partial e^{2}}{\partial w_{i}}. \tag{5.12}$$

K výpočtu gradientu G celé sítě, sečteme jednotlivé podíly každé dané váhy podle rovnice (5.12) přes všechna data. Poté můžeme odečíst malý podíl μ, také známou jako hodnota učení (learning rate) ze všech vah od G a tím provést metodu gradient descent a získat nové, lehce upravené váhy. Tato úprava vah ovšem přímo závisí na vstupech a výstupech reálného systému což může značně zkomplikovat vhodnou volbu hodnoty učení. K vyřešení tohoto problému je tedy vhodné použít normalizovaný gradient descent jak je popsán v [55] a je definován dle

$$w_{i}\left(k+1\right) = w_{i}\left(k\right) - \frac{\mu}{1 + \left\| \frac{\partial e^{2}}{\partial w_{i}} \right\|_{2}^{2}} \cdot \frac{\partial e^{2}}{\partial w_{i}}. \tag{5.13}$$

Vzhledem k relativně snadnému způsobu úpravy vah, je tento algoritmus vhodný k implementaci pomocí FPGA. Nicméně tato adaptační metoda nedokáže využít dostupné možnosti hardwaru tak jako je tomu například u dávkové metody Levenberg-Marquardt.

5.2.2 LEVENBERG-MARQUARDT

Algoritmus LM byl vyvinut nezávisle na sobě Levenbergem a Marquardtem. Tento algoritmus je iterační metoda, která hledá minimum chybové funkce. Metoda LM se stala standardní nelineární metodou nejmenších čtverců, která je kombinací metod Gradient

Descent a Gauss-Newton. Ze začátku postupuje LM jako gradient descent, tj. Pomalu ale s garancí konvergence k minimu, když je řešení blízko, stane se LM metodou Gauss-Newtonovou a konvergence k minimu se značně urychlí. Tato metoda patří do skupiny dávkového učení a váhy se aktualizují vždy přes celá data [56].

LM dosahuje vyšší rychlosti učení bez nutnosti výpočtů Hessovi matice. Hessova matice je čtvercová matice druhých parciálních derivací skalární funkce [57]. Autoři LM uvažují hledání minima chybové funkce jako sumy kvadrátu chyb,

$$E = \frac{1}{2}\sum_{N}^{k=0} e^2\,(k), \tag{5.14}$$

a díky této úvaze, lze pak aproximovat Hessovu matici jako

$$\mathbf{H} = \mathbf{J}^T \cdot \mathbf{J}, \tag{5.15}$$

kde gradient se spočte jakoukoliv

$$\mathbf{g} = \mathbf{J}^T \cdot \mathbf{e}, \tag{5.16}$$

kde e je vektor chyb neuronové sítě a $\mathbf{J}$ je matice jakobiánu, která obsahuje první derivace chyb neuronové sítě k neurálním váhám a biasu.

Algoritmus LM využívá aproximaci Hessovi matice pro adaptaci vah následovně

$$\mathbf{w} = \mathbf{w} - \left[\mathbf{J}^T\mathbf{J} + \mu\mathbf{I}\right]^{-1}\mathbf{J}^T\mathbf{e}, \tag{5.17}$$

kde **I** je jednotková matice.

Pokud je tedy rychlost učení $\mu = 0$, pak se LM chová jako Newton-Gaussova metoda. Pokud je rychlost učení μ velké, pak se bude LM chovat jako Gradient Descent s malým krokem. Po každém iteračním kroku se hodnota μ zmenší a v určitém momentu začne převažovat Newton-Gaussova metoda [58].

Tento algoritmus je výpočetně komplikovanější než je tomu v případě GD 5.2.1 a jeho aplikace do akceleračního hardwaru není jednoduchá, avšak obzvláště akcelerace maticového násobení pomocí FPGA značně zvýší výkon v případě nasazení na dané embedded zařízení.

5.2.3 NORMALIZED LEAST-MEAN-SQUARES

Normalizovaná metoda nejmenších čtverců (NLMS) [59] je rozšíření běžného algoritmu nejmenších čtverců která, je i přes její jednoduchost jednou z nejvíce užívaných adaptačních metod. Adaptační pravidlo metody NLMS může být popsáno následujícím způsobem

$$\mathbf{w}\left(k+1\right) = \mathbf{w}\left(k\right) + \Delta\mathbf{w}\left(k\right), \tag{5.18}$$

kde $\Delta\mathbf{w}\left(k\right)$ odpovídá

$$\Delta\mathbf{w} = \frac{\mu}{\epsilon + \mathbf{x}\left(k\right)^{T} \cdot \mathbf{x}\left(k\right)} \cdot \mathbf{x}\left(k\right) \cdot \mathbf{w}\left(k\right) = \eta \cdot \mathbf{x}\left(k\right) \cdot \mathbf{w}\left(k\right), \tag{5.19}$$

kde μ je hodnota učení (learning rate) a ϵ je konstanta (regularizační hodnota) jejíž cílem je zachovat stabilitu pro vstupy blízko nuly [55]. Adaptace je stabilní, pokud je splněna následující podmínka

$$0 \leq \mu \leq 2 + \frac{2\epsilon}{\mathbf{x}\left(k\right)^{T} \cdot \mathbf{x}\left(k\right)} \tag{5.20}$$

nebo pro případ bez regularizační hodnoty ϵ

$$\mu \in < 0, 2 >. \tag{5.21}$$

5.2.4 ADAM

Optimalizační metoda Adam je ze zde aplikovaných metod nejmladší a také nejkomplexnější. Jedná se o metodu, která je založena na stochastickém klesání podle gradientu (Stochastic gradient descent, SGD). Tato metoda byla prezentována v roce 2015 autory Diederik Kingma z OpenAI a Jimmy Ba z Univerzity v Torontu [60] a dnes je tato adaptační metoda hojně využívána pro svou efektivitu především pro hluboké neuronové sítě. Jméno algoritmu Adam není akronymem nýbrž je odvozeno od „adaptive moment estimation“ což je možné přeložit jako „odhad adaptivního momentu“.

Hlavním rozdílem mezi metodou SGD a Adam je, že Adam na rozdíl od SGD adaptuje i hodnotu učení pro každou váhu v celé neuronové síti během každého adaptačního kroku na základě odhadů prvních a druhých momentů, zatím co hodnota učení v případě SGD je po celou dobu učení pro všechny váhy konstantní. Autoři popisují metodu Adam jako kombinaci výhod dvou rozšíření metody SGD a to:

- AdaGrad (Adaptive Gradient Algorithm, Adaptivní Gradientní Algoritmus), který určuje rychlost učení na základě charakteristik, které zlepšují výkon při problémech s

řídkým gradientem, jako jsou například problémy se zpracováním řeči či strojovým viděním.

- RMSProp (Root Mean Square Propagation, Propagace Kvadratického Průměru), který také určuje rychlost učení podle charakteristik, které jsou přizpůsobeny na základě průměru posledních velikostí gradientů pro danou váhu, např. jak rychle se mění. Díky tomu funguje algoritmus dobře při online i nestacionárních problémech (zašuměných datech).

Jelikož metoda Adam kombinuje výhody AdaGrad i RMSProp, tak namísto přizpůsobení rychlosti učení na základě průměrného prvního momentu (průměr), jako v případě RMSProp využívá také průměr druhých momentů gradientu (necentrovaná variance). Algoritmus konkrétně vypočítává exponenciální momentový průměr gradientu a umocněného gradientu a pomocí vstupních parametrů β_1 a β_2 řídí rychlost zániku těchto momentů. Pseudokód optimalizační metody Adam je uveden v Alg. 3. Gradient optimalizovaný pomocí metody Adam je vypočten obdobně jako v případě 5.2.1.

Definice μ: hodnota učení
Definice $\beta_1, \beta_2 \in [0, 1)$: exponenciálních hodnoty rychlosti zániku odhadu momentů
Definice $f(w)$: Stochastická cílová funkce s parametry w
Definice w_0: Počáteční vektor vah
$m_0 \leftarrow 0$ (Počáteční vektor prvního momentu jdoucí k nule)
$v_0 \leftarrow 0$ (Počáteční vektor druhého momentu jdoucí k nule)
$k \leftarrow 0$ (Počáteční krok)
while w_k nedosáhlo konvergence **do**
$k \leftarrow k + 1$
$g_k \leftarrow \nabla_0 f_k(w_{k-1})$ (Výpočet gradientu stochastické cílové funkce v kroku k)
$m_k \leftarrow \beta_1 \cdot m_{k-1} + (1 - \beta_1) \cdot g_k$ (Úprava odhadu prvního momentu)
$v_k \leftarrow \beta_2 \cdot v_{k-1} + (1 - \beta_2) \cdot g_k^2$ (Úprava odhadu druhého momentu)
$\hat{m}_k \leftarrow \dfrac{m_k}{1 - \beta_1^k}$ (Výpočet upraveného odhadu prvního momentu)
$\hat{v}_k \leftarrow \dfrac{v_k}{1 - \beta_2^k}$ (Výpočet upraveného odhadu druhého momentu)
$w_k \leftarrow w_{k-1} - \dfrac{\mu \cdot \hat{m}_k}{\sqrt{\hat{v}_k} + \epsilon}$ (Aktualizace vah cílové stochastické funkce)
end while
return w_k (Výsledné váhy cílové funkce)

Alg. 3: Pseudokód průběhu výpočtu optimalizace cílové funkce pomocí metody Adam (převzato a upraveno dle [60]).

Tyto optimalizační metody jsou implementovány v knihovně nn_accel v souboru honu_optimizer.py. Pomocí těchto optimalizerů lze adaptovat neurony typu HONU. Optimalizér jako takový potřebuje ke své práci instanci neuronu HONU, kterou je možné vytvořit pomocí třídy uvnitř lnu_honu.py, qnu_honu.py či cnu_honu.py (kap. 12).

6 VYBRANÉ METODY DETEKCE ANOMÁLIE

V této kapitole jsou představeny dvě metody k detekci anomálie a chyb v monitorovaném systému. Je-li HONU dostatečně natrénované na datech a již se neučí, pak je jeho jediným úkolem detekovat anomálie v chování monitorovaného systému. Pokud ovšem monitorovaný systém vyžaduje přeučování dané jednotky v určitých intervalech, pak je nezbytné kromě detekce anomálií zavést i detekci učení na chybných datech. V opačném případě by mohlo dojít k naprosté nefunkčnosti celého systému.

6.1 LEARNING ENTROPY FOR NOVELTY DETECTION

Metoda Learning Entropy nebo také AISLE (Approximate individual sample learning entropy) dle [4] detekuje změny neurálních vah kde každá změna vah generuje určitou entropii. Pokud je tato změna dostatečně velká, je touto metodou detekována jako anomálie.

Entropie učení se získává z přírůstku vah $\Delta\mathbf{w}$ jednotlivých adaptačních metod. Pokud je tato metoda implementována spolu s algoritmem Levenberg-Maruqard 5.2.2 jedná se o metodu ABSLE (Approximate Batch Sample Learning Entropy) jak byla prezentována v [3]. Pro metodu Learning Entropy (LE) je nutné vhodně zvolit citlivost detekce

$$\boldsymbol{\alpha} = [\alpha_1 \quad \alpha_2 \quad \alpha_3 \quad \cdots \alpha_n], \tag{6.1}$$

kde platí $\alpha_1 < \alpha_2 < \cdots \alpha_n$. Zda je přírůstek vah neobvykle velký se určí z podmínky

$$|\Delta\mathbf{w}_i\,(k)| > \alpha_j \overline{|\Delta\mathbf{w}_i\,(k)|}, \tag{6.2}$$

kde $\overline{|\Delta\mathbf{w}_i\,(k)|}$ je průměrný přírůstek vah za určité období. Výpočet změny entropie se pak vypočte jako

$$\mathbf{E_A} = \frac{1}{n_w \cdot n_\alpha} \sum_{j=1}^{n_\alpha N(\alpha_j)}, \tag{6.3}$$

kde n_w je rozměr vektoru vah, n_α je rozměr vektoru citlivosti detekce a $N\,(\alpha_j)$ je množství neobvykle velkých přírůstků vah. Změna entropie může nabývat pouze hodnot $E_\mathbf{A} \in \langle 0, 1\rangle$.

6.2 DETEKCE POZVOLNĚ VZNIKAJÍCÍCH ANOMÁLIÍ

Tato metoda je zvolena jako podpůrná k metodě LE 6.1 a má podstatně jednoduší implementační algoritmus. Jejím hlavním cílem je odhalit neobvyklé stavy, které trvají delší dobu, aby na těchto stavech nedocházelo k přeučování jednotek HONU.

Následující metody dlouhotrvajících anomálií jsou založeny na vyhodnocování chyby neuronové sítě. Jsou zde implementovány dvě běžně využívané metody RMSE (Root Mean Square Error) a MAE (Mean Absolute Error) a jedna metoda dle vlastního uvážení.

RMSE je odmocnina z průměru kvadrátu chyb predikce vypočtena dle

$$\mathbf{RMSE} = \sqrt{\frac{1}{N}\sum_{k=0}^{N}\left(y_t - y_n\right)^2} \tag{6.4}$$

a MAE je průměrná absolutní chyba predikce vypočtena dle

$$\mathbf{MAE} = \frac{1}{N}\sum_{k=0}^{N}|y_t - y_n|. \tag{6.5}$$

Pro vlastní metodu je nejprve nutné zvolit počet hodnot n vypočtených neuronovou sítí na základě kterých se určí chyba pro aktuální vstupní hodnotu k. Dále pak dle 6.6 získám výslednou průměrnou chybu

$$e_{avrg.}\left(k\right) = \frac{\sum_{i=0}^{n}\left|y_{neuron}\left(k-i\right)\right|}{n}. \tag{6.6}$$

Neobvyklý stav se poté určí z poměru chyby $e_{avrg.}\left(k\right)$ k uživatelem definované hodnotě D pokud bude splněna podmínka 6.7, kde q je zvolená procentuální hodnota nedovolené chyby

$$100 \cdot \frac{e_{avrg.}\left(k\right)}{D} > q. \tag{6.7}$$

Tento způsob sledování chyby má především tu výhodu, že velikost chyby závisí na předešlých chybách a tím pádem se detekují neobvyklé stavy trvající delší dobu. Volbou n lze tedy vhodně navolit délka vyhodnocované chyby a zásadně tím ovlivnit detekční schopnosti.

V programu vyvinutým v této práci je detekční metoda AISLE převzata od doc. Ing. Bukovského, Ph.D. přičemž její jisté části jsou akcelerovány pomocí výkoného čipu zvoleného embedded zařízení. Metoda ABSLE, jejíž hlavní inspirace je dle AISLE je originální přístup k problematice jako takové a je díky ní možné vyhodnocovat například velké množství testů daného výrobku v podstatně vyšší výpočetní rychlosti než je tomu v metodě AISLE.

Implementované metody learning entropy a detekce dlouhodobých anomálií jsou uvedeny v souboru detector.py s příklady užití v examples (kap. 12).

7 METODY SIMULACE VSTUPNÍCH SYSTÉMŮ

Dynamický systém a data zpracovaná v kapitole 10 jsou níže detailněji popsány. Vysvětluji zde způsoby simulačních přístupů dynamického systému, který je simulován jak lokálně na desce PYNQ-Z1, tak externě pomocí Raspberry a popisuji zde původ a motivaci k využití dat z testovacích standů. Dále zde popisuji nutné základní předzpracování vstupních dat, které detekční SW automaticky provádí, aby bylo možné s těmito daty dále pracovat pomocí neuronových sítí. Jelikož se nejedná o zařízení, které by bylo schopné zpracovávat BigData, jako exportní formát historického průběhu detekčních algoritmů byl zvolen formát CSV především pro jeho jednoduchost. Více optimalizované formáty jako například HDF5 jsou zde zbytečně komplikované a většina jejich výhod stejně nebude využita.

7.1 SIMULACE DYNAMICKÉHO SYSTÉMU

Jako simulovaný dynamický systém byl pro účely této práce zvolen jako

$$\ddot{\hat{y}}(t) + \big(0.3 + 0.03 \cdot \hat{y}(t)\big) \cdot \dot{\hat{y}}(t) + 2 \cdot \hat{y}(t) = \Big(1 + 0.1 \cdot sin\big(\hat{y}(t)\big)\Big) \cdot u(t), \tag{7.1}$$

tento systém je buzený vstupem $u\,(t)$ a simuluje chování výstupu y nelineárního dynamického systému a skutečný výstup je určen jako

$$y(t) = \hat{y}(t) + \epsilon, \tag{7.2}$$

kde $\hat{y}$ je skutečné chování systému, y je měřená (simulovaná) hodnota a ϵ představuje šum v měření. Zvolené simulační vzorkování je konstantní $t \leftarrow \Delta t \cdot k; k = 1, 2, \ldots N$ (v Pythonu $k = 0, 1, \ldots N - 1$) pro $\Delta t = 0.1$. Zamýšleným simulačním nástrojem je ODE solver z knihovny Scipy, který ovšem dokáže řešit pouze diferenciální rovnice prvního řádu. Je proto nezbytné provést úpravu (7.1) vhodnou substitucí na stavovou formulaci soustavy rovnic prvního řádu, kterou získáme dle

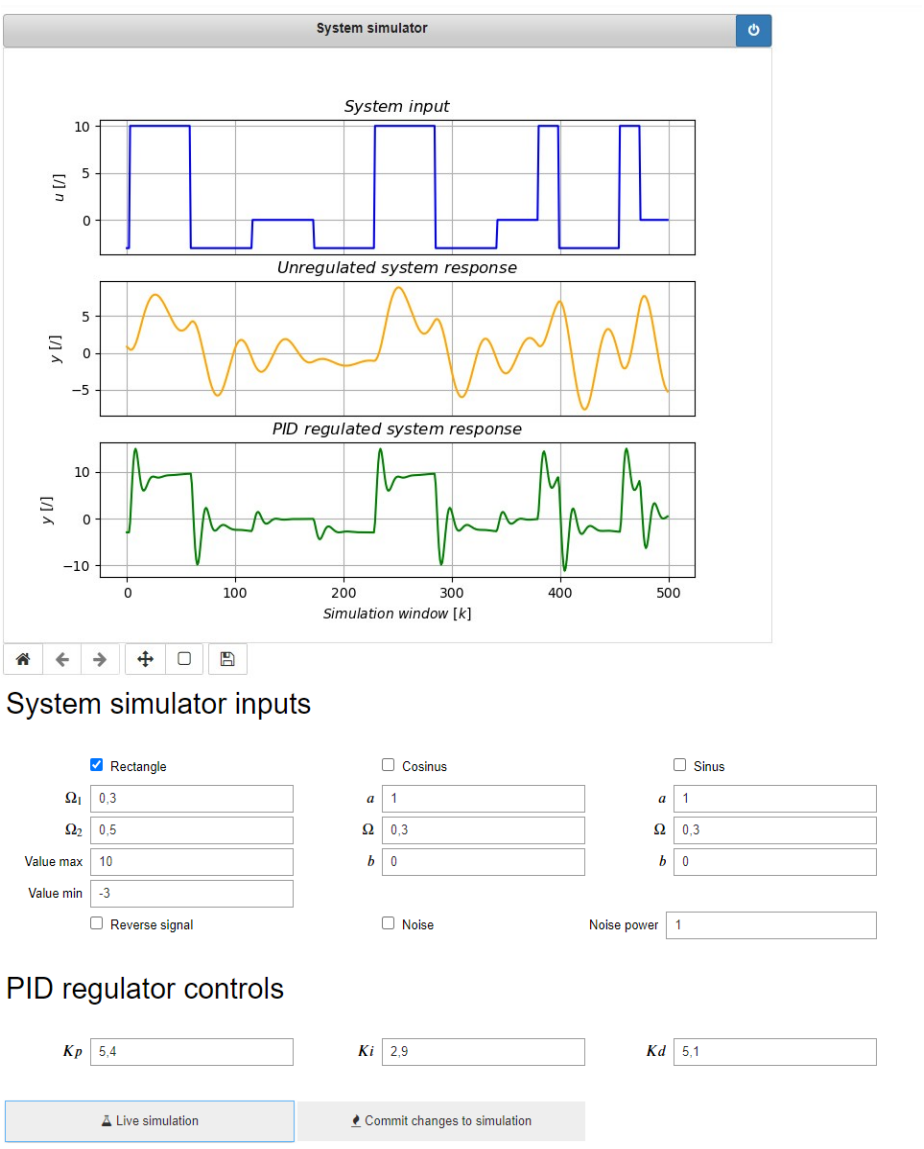
$$x_1 = \hat{y}, \tag{7.3}$$

$$x_2 = \dot{\hat{y}}, \tag{7.4}$$

$$\dot{x_1} = x_2, \tag{7.5}$$

$$\dot{x_2} = -0.3 \cdot x_2 - 0.03 \cdot x_1 \cdot x_2 - 2x_1 + u + 0.1 \cdot \sin(x_1) \cdot u. \tag{7.6}$$

Získané substituční členy lze již bez problému zapsat do funkce a řešit pomocí ODE solveru. Chování tohoto systému pro různé buzení u je zobrazeno na Obr. 16 včetně možných nastavení simulátoru. Všechna tato nastavení je možné měnit v průběhu probíhající simulace.



Obr. 16: Simulace dynamického systému pro různé buzení u v grafickém prostředí Jupyter notebook s možnostmi nastavení vstupního buzení a PID regulace.

Zobrazená vizualizace simulovaného systému je prováděna v Jupyter notebooku, který je na platformě zcela nezávislý a je tak nejvhodnějším řešením k implementaci jak na testovacím výpočetním PC, vývojové desce PYNQ-Z1, tak je vhodný k vizualizaci a generování dat pomocí externího systému jako je například Raspberry. Změnou nastavení (například fázový

posun) během průběhu simulace, která bude monitorována by mělo dojít k detekci anomálie a ověření funkčnosti detekčního systému.

K simulaci tohoto systému je využit ODE solver knihovny Scipy [61]. Tato simulace bude detekčními algoritmy na daném embedded zařízení monitorována a uživatelem zanášené anomálie budou detekovány. Pokud je simulace prováděna pouze na zařízení PYNQ-Z1, je nezbytné vypnout simulaci neregulovaného systému, jelikož se jedná o zbytečnou zátěž, která není k detekci nikterak důležitá, jelikož vstupem do detekčního modulu jsou data ze systému regulovaného pomocí PID.

Pokud je celý systém simulován externě, jak bylo i dle zadání této práce zamýšleno, je nezbytné během inicializace specifikovat, kam má být výstup simulace posílán. Simulátor jako takový podporuje odesílání dat přes rozhraní UART a SPI kde jsou data zasílána „na přeskáčku“, tedy nejdřív vstup do systému poté výstup simulovaného systému a tak pořád dokola. Zapojení portů v případě využití rozhraní SPI na GPIO desky Raspberry je uvedeno v Tab. 9. Pro rozhraní UART je doporučeno využít port USB.

Tab. 9: Rozložení komunikační pinů pro přenos pomocí SPI na Raspberry PI

Číslo pinu	Číslo BCM	Funkce
19	GPIO 10	MOSI – Master Out Slave In
21	GPIO 9	MISO – Master In Slave Out
23	GPIO 11	SCLK – Serial Clock
24	GPIO 8	CE0 – Chip Select 0
26	GPIO 7	CE1 – Chip Select 1

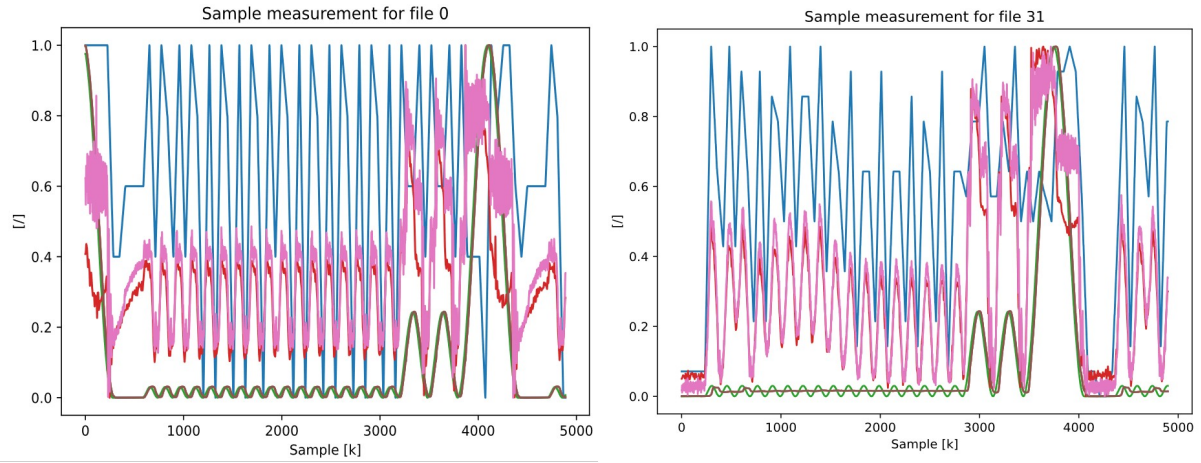
Skript použitý k simulaci dynamického systému použitého v této práci je uveden v knihovně system_simulator v souboru simulator.py s grafickým rozhraním v téže složce v souboru „Simulator_jupyter_GUI.ipynb“ (kap. 12).

7.2 DATA Z TESTOVACÍCH A MĚŘÍCÍCH STANIC

Navržená detekční SW aplikace (kap. 9) nemusí nutně pouze monitorovat dynamický systém, ale lze jej využít i k rychlému automatickému vyhodnocení dat ze zařízení, které testují či měří výrobky a v průběhu těchto měření data ukládají po částech a není tak možné provádět online monitorování průběhů. Zde použitá anonymní data pochází z interních testovacích standů automotive průmyslového partnera pro jeden jejich výrobek. Vzhledem k

povaze těchto dat není možné přesně sdělit jejich původ více, než je uvedeno na těchto řádcích.

Testování daného výrobku probíhá obvykle velice dlouhou dobu, proto je měřeno vždy ve vybraných časových intervalech s vysokým vzorkováním. Ve zde použité datové sadě se jedná o 510 měření jak pro test úspěšný tak neúspěšný. Vzorky měřených dat jsou uvedeny na Obr. 17 a tyto představují jednu dávku pro dávkovou metodu učení zvoleného neuronu. V dávce měřených dat je vždy více veličin a není ihned zřejmé, které veličiny spolu souvisí a které indikují zdar nebo nezdar testování výrobku.



Obr. 17: Ukázka dávky dat měření dvou časových úseků testování jednoho výrobku u kterých dopředu nevíme, které veličiny indikují úspěšnost testu a neúspěšnost testu detekujeme jako anomálie. Vpravo jsou anomální data, která ještě neznamenají jasný neúspěch testu, ale byla detekována a test po delší době vyšel negativně.

Jak již bylo řečeno výše, data z těchto stanic jsou vždy obdržena po zakončení měřícího či testovacího cyklu. V jednu chvíli ovšem běží větší množství těchto stanic najednou a obvykle každá s lehce odlišným zpožděním vzhledem k úkonům, které musí obsluha fyzicky provést u každé stanice. Na základě vlastností a původu dat uvedených výše byla tato data zvolena především k otestování schopností dávkového zpracování a vyhodnocování vstupních dat zde navrženého softwaru pro embedded zařízení. Pro potřeby tohoto způsobu měření by postačovalo jedno toto zařízení, jelikož by dokázalo monitorovat vícero měřících standů.

8 VÝVOJ BITSTREAMU PRO PYNQ-Z1

Jak již bylo uvedeno v 5.1 a 6 nejnáročnější výpočetní operace během detekce anomálií je maticové násobení. Proto bylo rozhodnuto provést na hardwarové úrovni právě maticové násobení co možná největší matice s ohledem na hardwarové zdroje a prozatím ponechat jednoduší operace jako je sčítání, odčítání, anebo logiku řízení na ARMovém procesoru. Čip FPGA na desce PYNQ-Z1 2.2.1 není příliš velký a je očekáváno, že valná většina jeho zdrojů se využije právě na požadované maticové násobení. Vývoj bitstreamu byl proveden v několika iteracích, jak je dle 3.3 běžné a jeho základní princip fungování byl inspirován základní implementací společnosti Xilinx dle [62].

8.1 VELIKOSTNÍ LIMITY HW AKCELEROVANÝCH NÁSOBIČŮ

Počet vstupů, které mohou do detektoru vstupovat se bude zpravidla lišit v závislosti na vstupním systému. Proto je vhodné vytvořit maticový HW násobič, který bude jak dostatečně velký pro co nejširší spektrum systémů vzhledem k dostupným zdrojům na čipu FPGA, tak dokáže využít co nejvyšší teoretickou výpočetní rychlost. Kromě toho je ovšem nutné mít na paměti i čas, který zabere přesun dat z paměti RAM do paměti BRAM, jelikož se může jedna o velmi limitující faktor a i seberychlejší implementace maticového násobení bude tímto přesunem omezena. Také je důležité si uvědomit, že pro dávkové trénování pomocí metody Levenberg-Marquardt 5.2.2 je velikost matice také zcela zásadní, jelikož přímo určuje maximální velikost jedné dávky a zásadně tak ovlivňuje rychlost adaptace. Algoritmus, který je v následujících podkapitolách využit má výpočetní složitost $\Theta(n^3)$ a jedná se tedy o zcela

```
template <typename T> void kernel_mmult(T a[N2], T b[N2], T out[N2]) {
L1:
 for (int m = 0; m < N; ++m) {
 L2:
  for (int n = 0; n < N; ++n) {
   T sum = 0;
  L3:
   for (int k = 0; k < N; ++k)
    sum += a[m * N + k] * b[k * N + n];
   out[m * N + n] = sum;
  }
 }
 return;
}
```

Alg. 4: Základní algoritmus k násobení matic. N2 odpovídá počtu prvků v matici. V tomto kódu nejsou uvedeny žádné direktivy překladu, aby byla zachována jeho čitelnost.

základní způsob maticového násobení. I přesto dokáže tento algoritmus pracovat velice rychle díky možnostem paralelizace a rozkládání smyček na čipu FPGA. Zdrojový kód logiky výpočtu algoritmu je uveden v Alg. 4.

Možné optimalizace překladu jsou detailně popsány v 8.3 a využité optimalizace překladu každou řešenou implementací jsou vždy popsány v jednotlivých podkapitolách shrnujících jejich rychlost 8.2.

8.1.1 MATICE OMEZENÁ DLE AXI DMA

První možností je omezit velikost matic na základě maximální propustnosti AXI DMA[4]. Dle specifikací v [6] a možností nastavení implementace v programu Vivado 3.3.2 je nejširší nastavení komunikace AXI-512, které umožňuje namapovat 512 bloků paměti o délce až 1024 bitů každý. Celkové namapované pole paměti tedy odpovídá velikosti 524 288 bitů. Při užití datového typu float32, který zabírá 32 bitů, je možné namapovat 16 384 číselných hodnot. Maximální velikost jedné matice odpovídá tedy dle tohoto omezení velikosti 128x128 a jedná se tedy o největší matici, kterou je použité FPGA schopné zpracovávat na rozumné výpočetní rychlosti.

Velikost této matice je vhodná i vzhledem k velikosti interní blokové paměti FPGA samotného, která má velikost 630Kb. Pro násobení i takto velkých matic poskytuje dostatečný prostor k dočasnému ukládání číselných hodnot obou vstupních matic, které jsou postupně čteny z paměti RAM pomocí AXI DMA rozhraní a následně násobeny. Výstupní matice bude opět postupně ukládána do BRAM a přes paměťové rozhraní nahrávána do paměti RAM. Této správy paměti lze dosáhnout především pomocí direktiv překladu, uvedených v Alg. 5, které se uvádí před logikou načítání a přesunu dat. Je zde však nutné poznamenat, že po čipu požadujeme adresaci více paměťových bloků, než je na daném čipu dostupné, což je možné jen díky dodatečným adresám speciálních úložných bloků LUTRAM[5] a průběžnému ukládání stavů výpočtů. Jak je podrobněji vysvětleno v 8.3.1. Rozdělení matic na 16 adresovatelných celků je minimum, které je zde nutné, jelikož jeden celek může nést maximálně 1024 číselných hodnot a menší bloky tedy nejsou implementovatelné.

4 AXI DMA – Advanced eXtensibe Interface Direct Memory Access slouží k přímému přístupu do části operační paměti procesoru. Toto rozhraní umožňuje programovatelné logice číst a zapisovat přímo do specifikované části paměti a tím značně urychlit výměnu dat mezi PL a PS.

5 LUTRAM – nebo běžněji známá pod pojmem distribuovaná paměť je speciální jednotka LUT, která dokáže uložit malé množství dat, běžně dokáže uchovat až 64bitů v jedné jednotce. Tento typ paměti je také ten nejrychlejší, který je na čipu FPGA dostupný.

```
extern "C" {
void matmult_accel(hls::stream<axis_t> &in, hls::stream<axis_t> &out) {
#pragma HLS INTERFACE s_axilite port = return bundle = control
#pragma HLS INTERFACE axis port = in
#pragma HLS INTERFACE axis port = out

  DataType l_A[N2];
  DataType l_B[N2];
  DataType l_C[N2];

#pragma HLS ARRAY_PARTITION variable = l_A factor = 16 dim = 1 cyclic
#pragma HLS ARRAY_PARTITION variable = l_B factor = 16 dim = 1 block
#pragma HLS ARRAY_PARTITION variable = l_C factor = 16 dim = 1 cyclic

/*
Logika načítání a přesunu dat mezi RAM a BRAM s voláním funkce násobení
*/

}
}
```

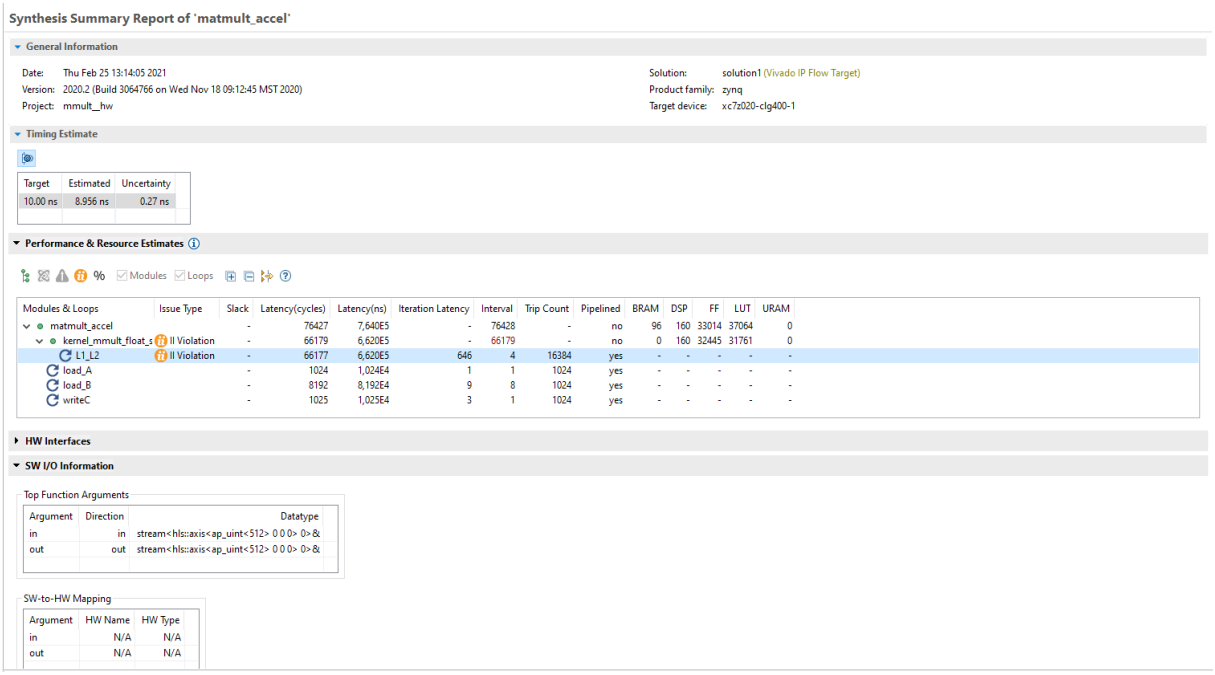
Alg. 5: Direktivy překladu specifikující komunikační porty AXI a způsob správy paměti. Každá matice je rozložena na 16 částí, aby bylo dosaženo vyšší datové propustnosti. Matice A a C jsou rozloženy cyklicky zatímco matice B je rozdělena na bloky.

Kromě výše uvedené logiky a postupu výpočtu je k úspěšnému automatickému překladu na úroveň hardwarově popisného jazyka nutné vytvořit ještě kód s takzvaným testbenchem, dle kterého bude otestován výsledný překlad. Testbench obsahuje čistě softwarovou cestu k provedení požadovaného algoritmu a porovnává se s výsledky simulovaného HW řešení. Vzhledem k možnému zaokrouhlování, které může probíhat jak na SW tak HW úrovni je nutné vhodně definovat povolenou odchylku, ke které může dojít a nebude považována za chybnou.

S vytvořeným testbenchem je možné provést automatickou syntézu tohoto řešení jehož shrnutí je uvedeno na Obr. 18. Hlavní informací je, že algoritmus dokáže pracovat na zvoleném časování 10ns což odpovídá frekvenci 100MHz a teoretický výpočetní čas odpovídá

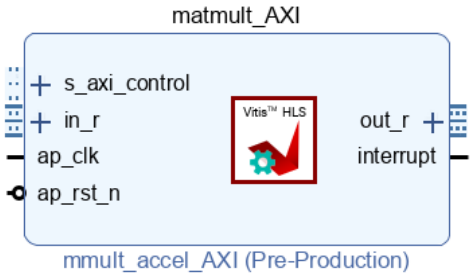
$$76427 \times 8.956ns = 0.684ms. \tag{8.1}$$

Další důležité informace jsou shrnutí očekávaného výkonu a využití hardwarových zdrojů. Pokud by kterýkoliv zdroj spotřebovával více jednotek, než jaké jsou na cílovém čipu dostupné, není možné provést syntézu návrhu a dané řešení by nebylo implementovatelné. Pokud je vše v pořádku je možné přejít k exportu RTL bloku, který je možné implementovat v softwaru Vivado HLS. Společně s exportem uvede program také maximální očekávanou stabilní frekvenci, která pro toto řešení odpovídá 111.66MHz.



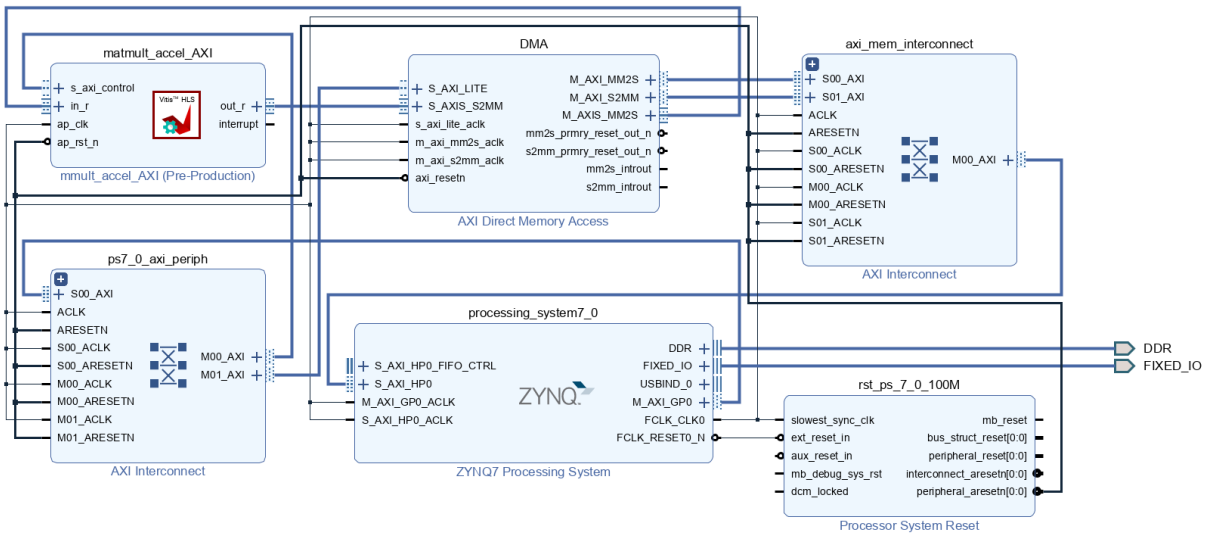
Obr. 18: Shrnuti syntézy algoritmu maticového násobiče s omezením dle AXI a upozorněním na vysokou latenci při výpočtu.

Po úspěšném exportu se musí vytvořit přidružený IP Package v programu Vivado na základě vygenerovaných souborů uvnitř exportovaného zipu. To se provede rozbalením zipu do požadované složky a jejím přidáním do externích IP v nastavení projektu programu Vivado. Po úspěšném přidání jej lze přidat do pracovního prostoru návrhu obvodu, takzvaného Block Diagramu. Výsledný blok již obsahuje definice datových propojení, jak je vidět na Obr. 19 a je možné jej nyní implementovat.



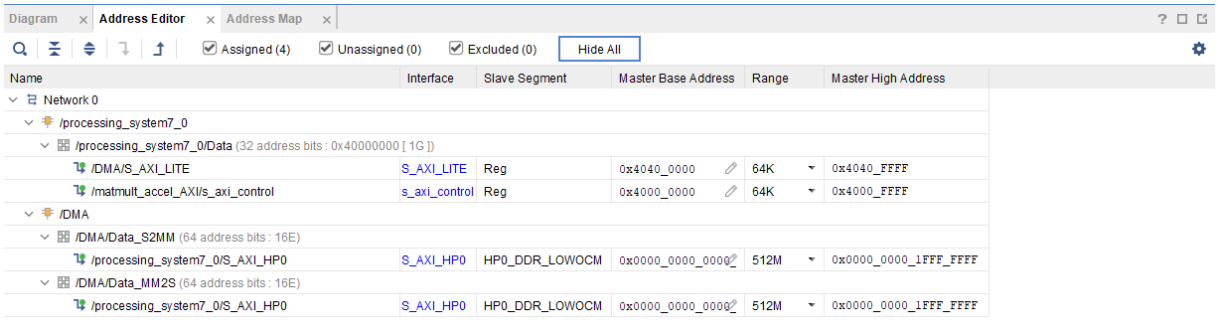
Obr. 19: Grafické zobrazení vyvinutého bloku maticového násobiče v implementačním SW Vivado. Vstupní matice jsou do bloku přenášeny na portu in_r a výstupní matice je přenášena portem out_r. Ovládání činnosti bloku probíhá pomocí portu s_axi_control. Ostatní porty slouží k přenosu časového signálu, resetu a přerušení.

Vytvořený IP blok je nyní nutné integrovat na hardwarové úrovni do funkčního celku. K tomu se použijí již existující IP bloky v programu Vivado. Konkrétně jde o 2x AXI Interconnect což jsou správci komunikace po sběrnici AXI, 1x AXI Direct Memory Access, který se stará o přístup do operační paměti, 1x Processor System Reset, který umožňuje asynchronní reset a nakonec 1x blok procesoru architektury Zynq, který je instalován na desce PYNQ-Z1. Všechny tyto bloky je nyní nutné správně propojit jak na datové úrovni, tak na úrovni resetu a časovače. Blokový diagram se zapojením maticového násobiče je zobrazen na Obr. 20. Všechny datové spoje jsou zobrazeny tučnými modrými spojnicemi a orientace datových toků je definována na úrovni jednotlivých bloků. Obecně platí, že levá část bloku odpovídá vstupům a pravá část odpovídá výstupům. Tučnou černou barvou je přenášen asynchronní reset a tenkou černou čarou je přenášen synchronní signál času. Ačkoliv Vivado podporuje automatizaci zapojení, obvykle tato funkce nefunguje příliš dobře, jelikož nemůže vědět, jakým způsobem se mají data přenášet či co má být asynchronní reset a proto je obvykle nutné vše zapojit manuálně.



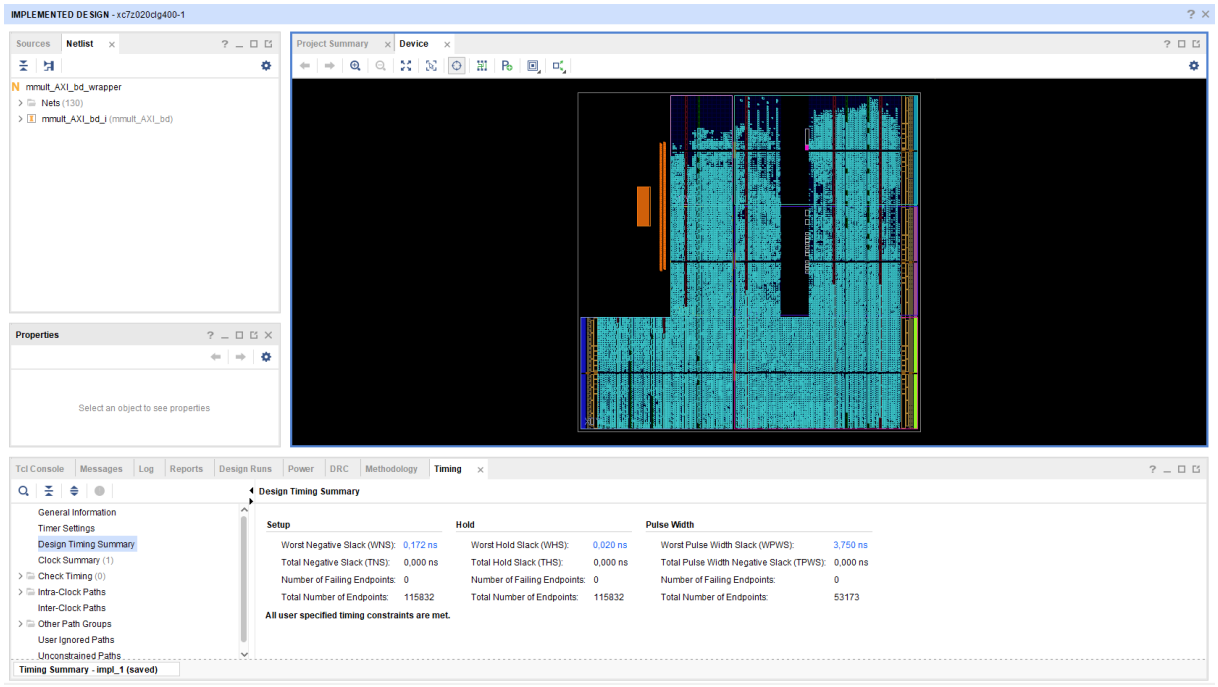
Obr. 20: Navržené schéma zapojení maticového násobiče v Block Diagramu. Tmavě modré spojnice odpovídají datovým spojům. Tučné černé spojnice přenáší signál asynchronního resetu. Slabé černé spojnice přenáší synchronní signál času.

Hotový Block Diagram se před dalším krokem musí vybrat za hlavní modul, neboli „top level module“, aby byla možná další syntéza obvodu na FPGA. Dále se musí specifikovat oblast paměti, do které bude mít IP DMA přes AXI přístup. Toto nastavení je možné provést pomocí editoru adres v záložce vedle Block diagramu. Adresy a velikosti namapovaných polí jsou zobrazeny na Obr. 21.

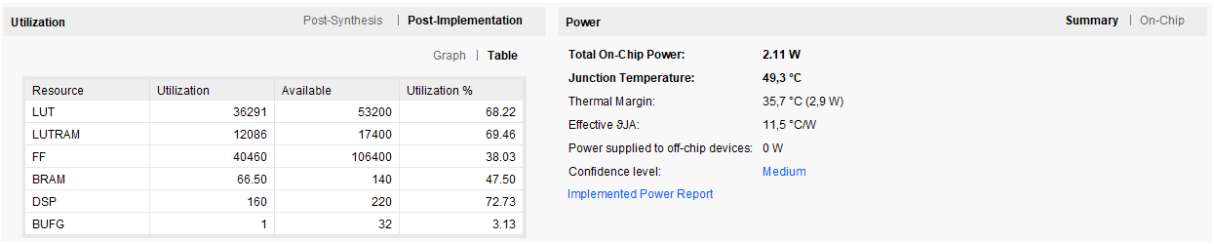


Obr. 21: Namapované adresy paměti pro bloky AXI a DMA s jejich fyzickou adresou v paměti RAM a specifikací přidruženého rozhraní.

Nyní je možné provést krok syntézy a implementace. Ačkoliv jsou tyto dva kroky převážně automatizované, v případě jakékoliv chyby při překladu je nutné se obvykle navrátit až k návrhu algoritmu. Tedy do programu Vitis a danou chybu odstranit vhodnější volbou direktiv překladu či upravením logiky implementované do IP bloku. Po úspěšném průběhu je možné provést analýzu návrhu z pohledu využití FPGA, očekávané spotřeby, spolehlivosti výpočtů či očekávané teploty čipu během výpočtů. Grafické zobrazení využití čipu s časováním je zobrazeno na Obr. 22 a tabulka celkového reálného využití zdrojů FPGA s jeho očekávanou teplotou jsou uvedeny na Obr. 23.



Obr. 22: Grafické shrnutí výsledku vlastní implementace maticového násobiče o velikosti násobených matic 128x128 na čip FPGA se zobrazením výsledného časování.



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	36291	53200	68.22
LUTRAM	12086	17400	69.46
FF	40460	106400	38.03
BRAM	66.50	140	47.50
DSP	160	220	72.73
BUFG	1	32	3.13

Power **Summary** | On-Chip

Total On-Chip Power:	**2.11 W**
Junction Temperature:	**49,3 °C**
Thermal Margin:	35,7 °C (2,9 W)
Effective ϑJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Obr. 23: Shrnutí využití hardwarových zdrojů čipu FPGA násobičem 128x128 s očekávanou spotřebou a teplotou čipu. Dle očekávání je vyžito velké množství jednotek DSP (72.73%) a jednotek LUT (68.22%).

Už z grafického přehledu je jasné, že velká část zdrojů čipu FPGA je využita. Vysoké využití jednotek DSP (160 z 220) bylo očekáváno, vzhledem k tomu, že právě tyto jednotky se starají o samotné násobení a sčítání. Velmi vysoké využití je ovšem i u LUT[6], což je způsobeno jak nedostatkem DSP k vykonání maticového násobení „najednou“, tak požadavky na přístup do paměti BRAM, jak je vysvětleno v 8.3.1.

Software Vivado při syntéze a implementaci logiky algoritmu na nižší úroveň automaticky vytvořil logiku výpočtu na hardwarové úrovni za maximálního, ale efektivního využití dostupných zdrojů specifikovaného čipu. Pokud by tento čip měl více jednotek DSP, bylo by možné provádět výpočty rychleji. Alternativní možností zde je opět se vrátit do softwaru Vitis a provést další optimalizaci jak algoritmu, tak direktiv překladu, aby bylo dosaženo efektivnějšího využití hardwarových zdrojů. Je zde však nutné zdůraznit, že ke značné optimalizaci pomocí direktiv překladu již došlo, jak je uvedeno v 8.2.1 a i další pokusy o optimalizaci by nemusely přinést požadované zlepšení.

Posledním krokem v programu Vivado je vytvoření samotného bitstreamu, který je možné nahrát na FPGA pomocí frameworku PYNQ. Spolu s bitstreamem budou vytvořeny i soubory s příponou .hwh a .tcl, ve kterých je obsažena hardwarová specifikace celého bitstreamu. Tyto soubory jsou nezbytné pro framework PYNQ, jelikož na jejich základě vytváří přímé připojení pythonu do FPGA. Na základě těchto připojení vytvoří dynamickou knihovnu, pomocí které je možné přistupovat k IP blokům obsaženým v bitstreamu čistě objektovým přístupem. Funkčnost bitstreamu a jeho výpočetní rychlost je otestována v 8.2.1.

6 LUT – Lookup table neboli vyhledávací tabulka je datová struktura, ve které je uložen omezený počet hodnot určité funkce nebo matematické operace, takovým způsobem, aby bylo možné pro některé argumenty rychle vyhledat hodnotu funkce.

8.1.2 MATICE OMEZENÁ DLE BRAM

Druhou možností je omezit velikost matice dle velikosti BRAM, která má na desce PYNQ-Z1 velikost 630Kb. Do této paměti se tedy musejí vejít všechny členy všech tří matic. Při uvažování typu float32 připadá na každou matici maximálně 6562 číselných hodnot. Čemuž by odpovídala matice 81x81. Dostupná BRAM se ovšem skládá z adresovatelných 36Kb bloků, které mohou být zapojeny jako duální 18Kb bloky, z čehož pramení omezení na hardwarové úrovni a je tedy možné najednou číst či zapisovat z nebo do 35 paměťových bloků kde každý 18Kb blok může nést maximálně 562 číselných hodnot.

Ačkoliv FPGA nemusí pracovat pouze s násobky dvou a hardwarově akcelerovaná matice o velikosti 81x81 by byla možná, vznikal by zde značný overhead dodatečným načítáním a přesunem dat po sběrnici AXI, která v dvojkové soustavě pracuje. Dále by takto velká matice nemusela přinést dostatečně vysoký přínos rychlosti oproti matici 128x128 jelikož má stále vysoké požadavky na jednotky DSP. Z těchto důvodů byla jako vhodná zvolena matice o velikosti 64x64, jelikož ji lze vhodně namapovat na souběžné paměťové bloky.

Dále byl proveden také pokus o zmenšení této matice na velikost 32x32 a 16x16 s cílem urychlení výpočtů u systémů, kterým bude stačit méně vstupních hodnot a maximalizovat tak výpočetní rychlost ideálně až na úroveň megahertzů. Matice o velikostech 32x32 a 16x16 bohužel nepřináší očekávanou výpočetní rychlost. I přes velmi intenzivní snahu optimalizovat přesun dat mezi FPGA a RAM trvá tento přesun příliš dlouho a výpočet čistě na procesoru ARM pomocí knihovny numpy dosahuje vyšších výpočetních rychlostí.

Jako problematický se ukázal blok DMA, který dokáže adresovat paměťový prostor o minimální velikosti 2^{16} $(65,536)$ bitů. Pro využití matic menších než 64x64 by tedy bylo nutné i část algoritmu, která je napsaná v Pythonu přepsat na hardwarovou úroveň, čímž by se ovšem velmi zkomplikovala celá implementace a značně se tak přesáhl rozsah této práce.

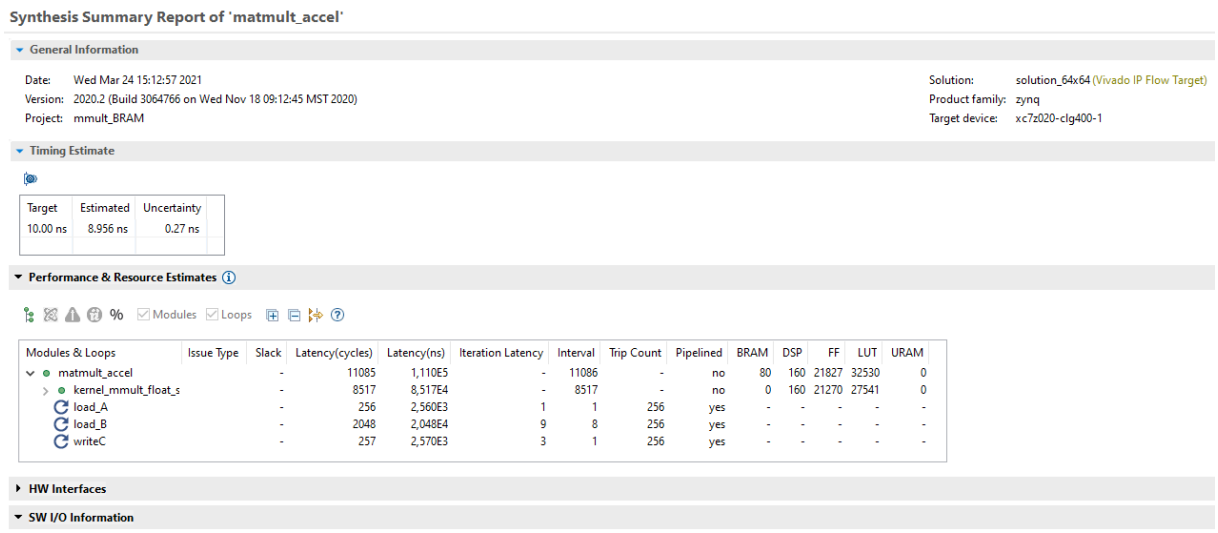
Oproti násobiče omezeného dle AXI 8.1.1 je zde kladen větší důraz na optimálnější využití paměti BRAM dle metodik 8.3.1, a je změněna velikost rozkladu pro výstupní matici C, jak je uvedeno v Alg. 6. U nejmenšího násobiče došlo ještě ke změně rozložení matice B z bloků na cyklické, díky čemuž bylo možné dosáhnout ještě vyšší teoretické výpočetní rychlosti.

```
extern "C" {
void matmult_accel(hls::stream<axis_t> &in, hls::stream<axis_t> &out) {
#pragma HLS INTERFACE s_axilite port = return bundle = control
#pragma HLS INTERFACE axis port = in
#pragma HLS INTERFACE axis port = out

  DataType l_A[N2];
  DataType l_B[N2];
  DataType l_C[N2];

#pragma HLS ARRAY_PARTITION variable = l_A factor = 16 dim = 1 cyclic
#pragma HLS ARRAY_PARTITION variable = l_B factor = 16 dim = 1 block
#pragma HLS ARRAY_PARTITION variable = l_C factor = 8 dim = 1 cyclic

/*
Logika načítání a přesunu dat mezi RAM a BRAM s voláním funkce násobení
*/

}
}
```

Alg. 6: Direktivy překladu specifikující komunikační porty AXI a způsob správy paměti pro optimální využití BRAM pro násobiče o velikostech 64x64, 32x32 a 16x16. V případě nejmenšího násobiče došlo ještě ke změně rozložení matice B na cyklické a bylo tak dosaženo ještě vyšší teoretické výpočetní rychlosti.

Celkově je zde pro všechny tři možnosti velikostí matic využito 40 z 35 fyzických paměťových bloků a jsou tedy opět využity dodatečné adresy pomocí speciálních úložných bloků. Ačkoliv se nabízí zde zmenšit počet bloků pro matici C na minimální velikost 4 pro matici 64x64 a přiblížit se tak počtu fyzických adres, jedná se o nevýhodný krok vzhledem k velkému omezení zápisu do paměti a z toho plynoucího pomalejšího výpočtu. Dodatečné hardwarové zdroje potřebné pro virtuální adresy jsou v tomto případě zanedbatelné.

Matice B zde bohužel obdobně jako v případě Alg. 5 musí zůstat namapována jako bloky. Pokud by se matice B namapovala cyklicky a upravili by se ostatní direktivy překladu, je možné dosáhnout na velmi vysoké výpočetní rychlosti. Spotřeba DSP by ovšem pro matici 64x64 přesahoval 5200 jednotek čímž by se přesáhl počet reálných jednotek více než 24x a výsledný návrh by tak byl neimplementovatelný. Shrnutí syntéz algoritmů všech tří zvažovaných násobičů jsou uvedeny na Obr. 24, Obr. 25 a Obr. 26.



Synthesis Summary Report of 'matmult_accel'

General Information

Date: Wed Mar 24 15:12:57 2021
Version: 2020.2 (Build 3064766 on Wed Nov 18 09:12:45 MST 2020)
Project: mmult_BRAM

Solution: solution_64x64 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	8.956 ns	0.27 ns

Performance & Resource Estimates

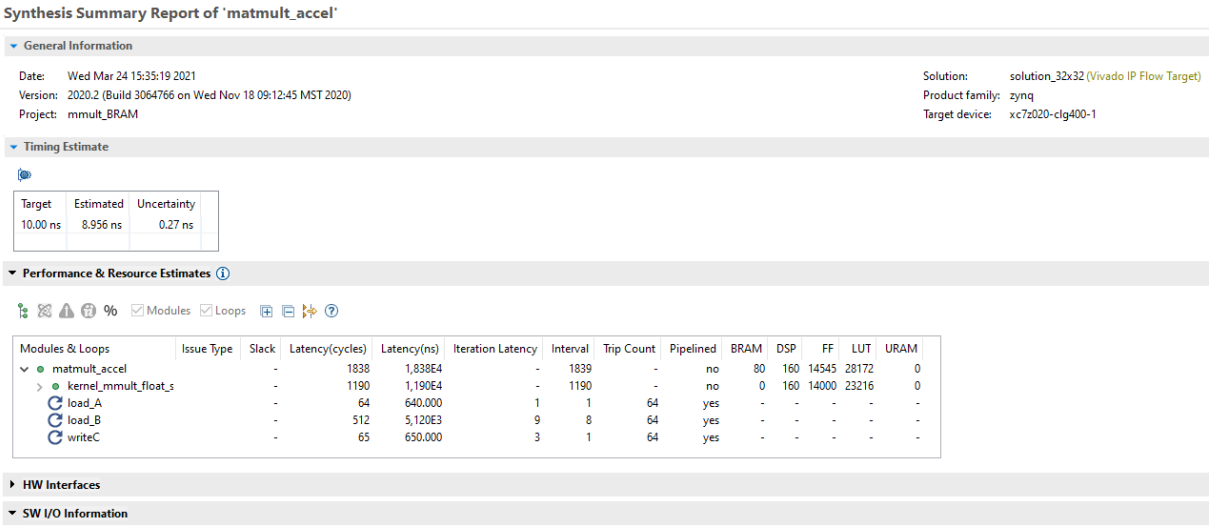
Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmult_accel		-	11085	1,110E5	-	11086	-	no	80	160	21827	32530	0
kernel_mmult_float_s		-	8517	8,517E4	-	8517	-	no	0	160	21270	27541	0
load_A		-	256	2,560E3	1	1	256	yes	-	-	-	-	-
load_B		-	2048	2,048E4	9	8	256	yes	-	-	-	-	-
writeC		-	257	2,570E3	3	1	256	yes	-	-	-	-	-

HW Interfaces

SW I/O Information

Obr. 24: Shrnutí syntézy algoritmu maticového násobiče matice 64x64.



Synthesis Summary Report of 'matmult_accel'

General Information

Date: Wed Mar 24 15:35:19 2021
Version: 2020.2 (Build 3064766 on Wed Nov 18 09:12:45 MST 2020)
Project: mmult_BRAM

Solution: solution_32x32 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	8.956 ns	0.27 ns

Performance & Resource Estimates

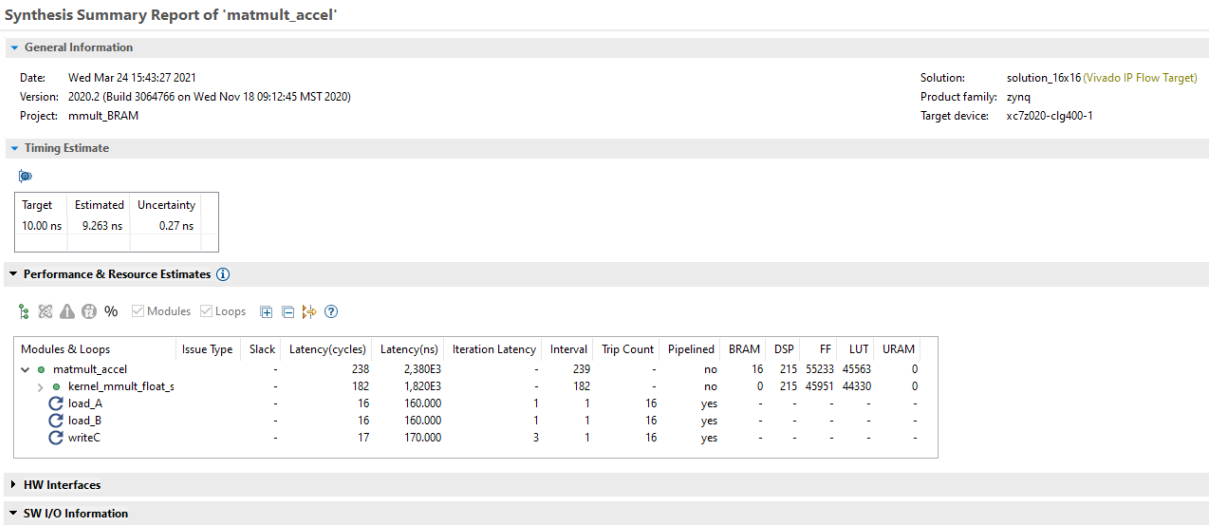
Modules Loops

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmult_accel		-	1838	1,838E4	-	1839	-	no	80	160	14545	28172	0
kernel_mmult_float_s		-	1190	1,190E4	-	1190	-	no	0	160	14000	23216	0
load_A		-	64	640.000	1	1	64	yes	-	-	-	-	-
load_B		-	512	5,120E3	9	8	64	yes	-	-	-	-	-
writeC		-	65	650.000	3	1	64	yes	-	-	-	-	-

HW Interfaces

SW I/O Information

Obr. 25: Shrnutí syntézy algoritmu maticového násobení matice 32x32



Synthesis Summary Report of 'matmult_accel'

General Information

Date: Wed Mar 24 15:43:27 2021
Version: 2020.2 (Build 3064766 on Wed Nov 18 09:12:45 MST 2020)
Project: mmult_BRAM
Solution: solution_16x16 (Vivado IP Flow Target)
Product family: zynq
Target device: xc7z020-clg400-1

Timing Estimate

Target	Estimated	Uncertainty
10.00 ns	9.263 ns	0.27 ns

Performance & Resource Estimates

Modules & Loops	Issue Type	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
matmult_accel		-	238	2,380E3	-	239	-	no	16	215	55233	45563	0
kernel_mmult_float_s		-	182	1,820E3	-	182	-	no	0	215	45951	44330	0
load_A		-	16	160.000	1	1	16	yes	-	-	-	-	-
load_B		-	16	160.000	1	1	16	yes	-	-	-	-	-
writeC		-	17	170.000	3	1	16	yes	-	-	-	-	-

HW Interfaces

SW I/O Information

Obr. 26: Shrnutí syntézy algoritmu maticového násobení matice 16x16

V případě Obr. 24 a Obr. 25 je využit podobný způsob překladu a optimalizace jako v případě násobiče 8.1.1 a očekávané teoretické výpočetní rychlosti jsou

$$11085 \times 8.965ns = 0.099ms \tag{8.2}$$

pro násobič 64x64 a

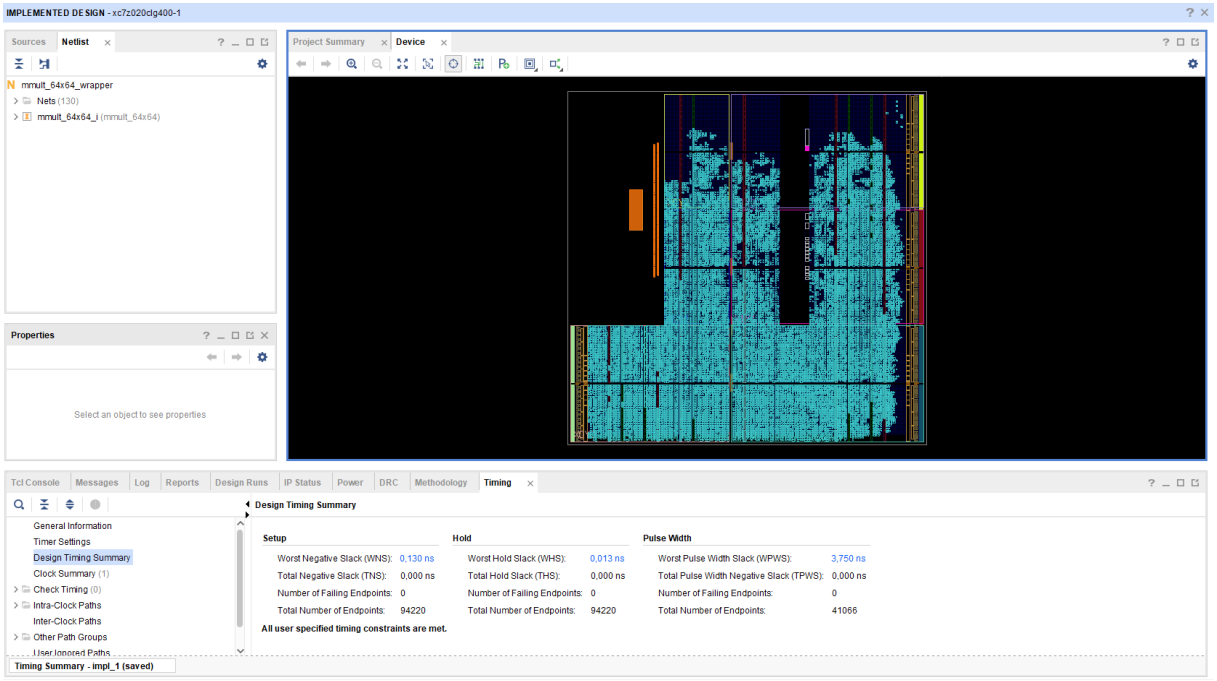
$$1838 \times 8.965ns = 0.016ms \tag{8.3}$$

pro násobič 32x32. U násobiče Obr. 26 byl změněn především způsob správy dat matice B z bloků na cyklický a přidružené direktivy překladu, díky čemuž je dosaženo podstatně vyšší teoretické výpočetní rychlosti

$$238 \times 9.263ns = 2.204\mu s, \tag{8.4}$$

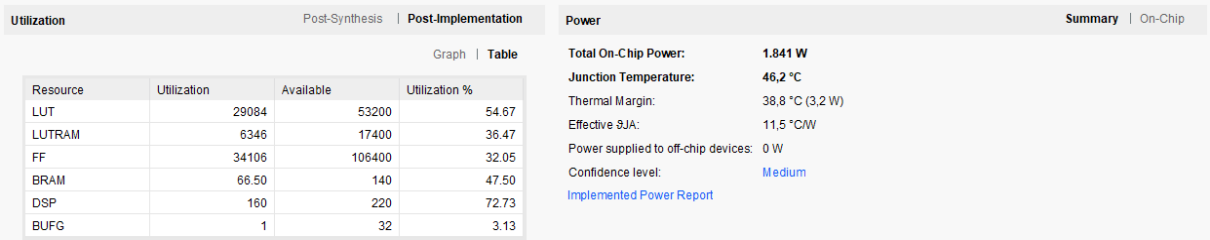
za cenu znatelně vyššího využití hardwarových zdrojů čipu FPGA než u jiného zde zvažovaného řešení.

Řešení exportovaná pomocí programu Vitis HLS byla dále implementována v programu Vivado obdobným způsobem, jako tomu bylo v případě matice omezené dle AXI a bylo tedy využito i obdobné hardwarové zapojení jako v případě Obr. 20. Jediné co bylo nutné vždy změnit je velikost bufferu bloku AXI DMA a šířku sběrnice AXI tak, aby vždy odpovídala velikostem cílových matic. Shrnutí reálného využití zdrojů FPGA implementace maticového násobiče 64x64 je graficky zobrazeno na Obr. 27.



Obr. 27: Grafické shrnutí výsledku vlastní implementace maticového násobiče o velikosti násobených matic 64x64 na čip FPGA s zobrazením výsledného časování.

Vzhledem k silné analogii řešení pro násobič 32x32 zde není dále graficky zobrazeno grafické shrnutí výsledku implementace pro toto řešení a je zde níže uvedeno pouze jeho shrnutí. Oproti grafickému přehledu v implementaci matice 128x128 je zřejmé, že je využito méně jednotek LUT což bylo očekáváno vzhledem k obdobnému způsobu překladu na hardwarovou úroveň a je to i číselně doloženo v Obr. 28 pro násobič 64x64 respektive v Obr. 29 pro násobič 32x32.



Utilization — Post-Synthesis | Post-Implementation — Graph | Table

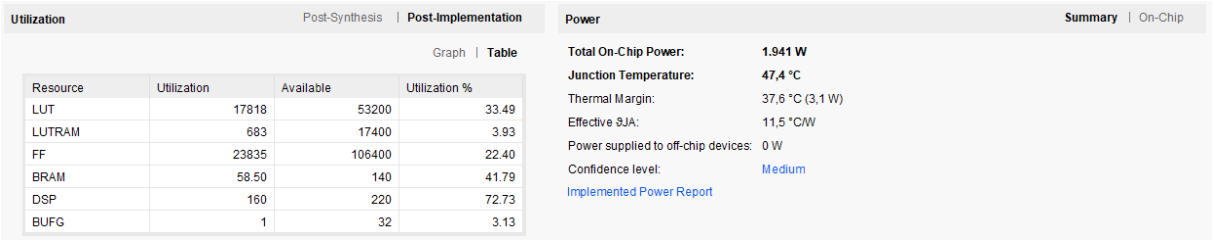
Resource	Utilization	Available	Utilization %
LUT	29084	53200	54.67
LUTRAM	6346	17400	36.47
FF	34106	106400	32.05
BRAM	66.50	140	47.50
DSP	160	220	72.73
BUFG	1	32	3.13

Power — Summary | On-Chip

Total On-Chip Power:	1.841 W
Junction Temperature:	46,2 °C
Thermal Margin:	38,8 °C (3,2 W)
Effective ϑJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Obr. 28: Shrnutí využití hardwarových zdrojů FPGA násobičem 64x64 s očekávanou spotřebou a teplotou čipu. Využití jednotek DSP je identické jako v případě násobiče 128x128 a využití ostatních zdrojů je jen lehce nižší. Zato je dosahováno podstatně vyšší teoretické výpočetní rychlosti oproti násobiči 128x128.Y



Utilization — Post-Synthesis | Post-Implementation — Graph | Table

Resource	Utilization	Available	Utilization %
LUT	17818	53200	33.49
LUTRAM	683	17400	3.93
FF	23835	106400	22.40
BRAM	58.50	140	41.79
DSP	160	220	72.73
BUFG	1	32	3.13

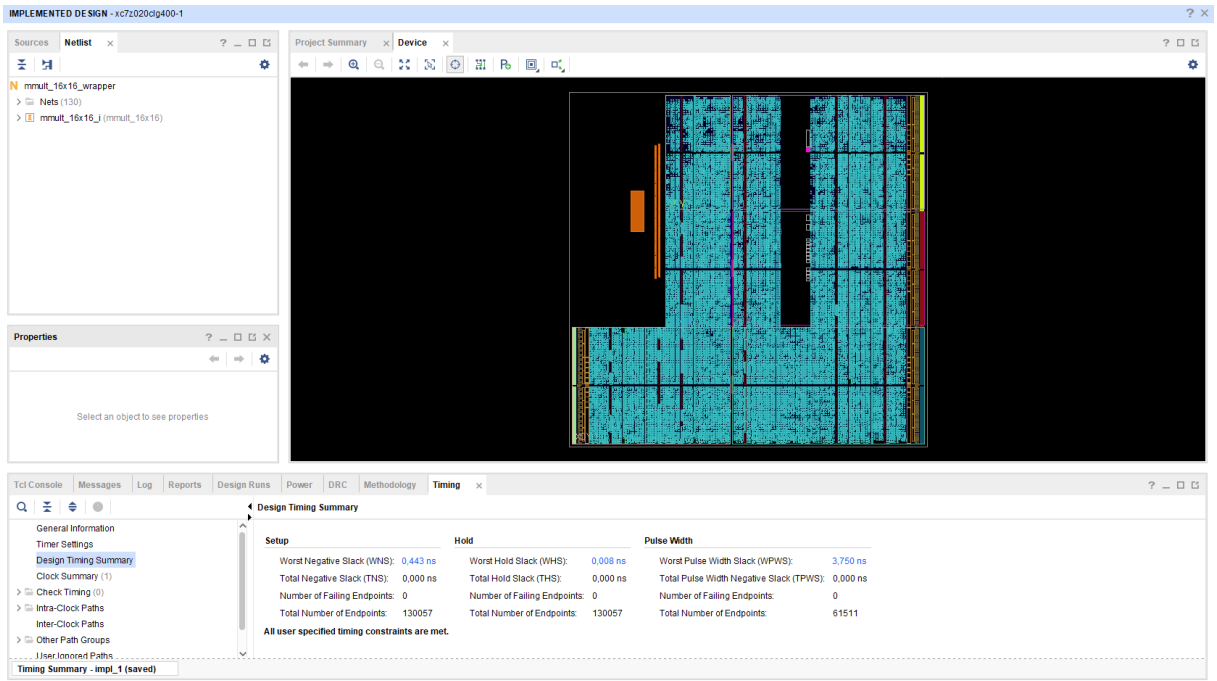
Power — Summary | On-Chip

Total On-Chip Power:	1.941 W
Junction Temperature:	47,4 °C
Thermal Margin:	37,6 °C (3,1 W)
Effective ϑJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

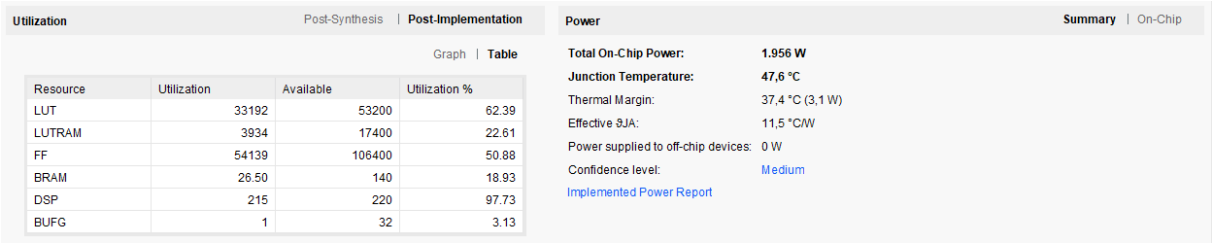
Obr. 29: Shrnutí využití hardwarových zdrojů FPGA násobičem 32x32 s očekávanou spotřebou a teplotou čipu. Využití jednotek DSP je identické jako v případě násobiče 128x128 a 64x64 avšak využití ostatních zdrojů je již znatelně nižší což je dáno znatelně nižším počtem násobených členů a jejich přiblížením se hardwarovým schopnostem čipu.

Rozdílný případ je to však u násobiče 16x16, který, jak již bylo řečeno, byl řešen s rozdílnou správou paměti pro matici B, díky čemuž je možné dosáhnout podstatně vyšších teoretických výpočetních rychlostí, za cenu velmi vysokého využití čipu FPGA, jak je graficky znázorněno na Obr. 30 a je zde patrné, že pro tuto implementaci nemá čip FPGA prakticky žádné volné zdroje pro případné další bloky.



Obr. 30: Grafické shrnutí výsledku vlastní implementace maticového násobiče o velikosti násobených matic 16x16 na čip FPGA se zobrazením výsledného časování.

Jak je uvedeno i v Obr. 31, je využití čipu opravdu vysoké. Je zde využito 215 z 220 jednotek DSP a i využití ostatních zdrojů dosahuje hodnot násobiče 128x128. I díky těmto hardwarovým nárokům ovšem dosahuje tak vysoké teoretické výpočetní rychlosti.



Utilization Post-Synthesis | **Post-Implementation**

Graph | **Table**

Resource	Utilization	Available	Utilization %
LUT	33192	53200	62.39
LUTRAM	3934	17400	22.61
FF	54139	106400	50.88
BRAM	26.50	140	18.93
DSP	215	220	97.73
BUFG	1	32	3.13

Power **Summary** | On-Chip

Total On-Chip Power:	**1.956 W**
Junction Temperature:	**47,6 °C**
Thermal Margin:	37,4 °C (3,1 W)
Effective ϑJA:	11,5 °C/W
Power supplied to off-chip devices:	0 W
Confidence level:	Medium

Implemented Power Report

Obr. 31: Shrnutí využití hardwarových zdrojů FPGA násobičem 16x16 s očekávanou spotřebou a teplotou čipu. Tato implementace využívá téměř všechny dostupné jednotky DSP (215 z 220) a využitím ostatních zdrojů se přibližuje násobiči 128x128 i díky tomu dosahuje tato implementace nejvyšší teoretické rychlosti ze všech zde zvažovaných možností.

Bitstreamy pro jednotlivé implementace jsou opět vygenerovány obdobným způsobem jako v případě 8.1.1 a nahrány pomocí frameworku PYNQ do čipu FPGA. Opět je ovšem nezbytné poskytnout frameworku nové soubory s příponou .hwh a .tcl, aby framework dokázal bezpečně poznat, jak je celý čip rozložen a mohl vytvořit dynamickou knihovnu sloužící k přístupu k jednotlivým IP blokům obsaženým v bitstreamu. Jak již bylo naznačeno v úvodu této podkapitoly, nedosahují všechny bitstreamy teoretické výpočetní rychlosti. Toto je dále vysvětleno a diskutováno v 8.2.2.

8.1.3 MOŽNOSTI VEKTOROVÉHO NÁSOBENÍ

Poslední zvažovanou možností je omezit HW násobič pouze na násobení vektorů. Tato metoda by měla dosahovat také velmi vysoké teoretické výpočetní rychlosti až na úroveň násobiče 16x16 a to s větším množstvím vah za potenciálně nižšího využití hardwarových zdrojů. I přesto nebyla tato metoda v této práci již implementována, vzhledem k omezení reálné rychlosti přesunu dat mezi PS a PL, jak je zdokumentováno v 8.2.2. Tato metoda by již patrně dosáhla jen velmi nízkého či dokonce žádného reálného zrychlení násobiče pro aktuální způsob implementace a jednalo by se aktuálně pouze o zbytečně vynaložený vývojový čas a to i přes to, že možná modifikace HONU 5.1.1 zvažuje pouze vektorové násobení. Tato modifikace HONU ovšem uvažuje především výpočet na CPU a nikoliv na FPGA, které dokáže N-Dimenzionální násobení provést velmi rychle. Pokud by se ovšem řešila celá implementace detekčního řešení čistě na hardwarové úrovni, mohla by implementace vektorového násobiče ušetřit cenné hardwarové zdroje, které by mohly být využity jiným způsobem. Stejně tak by případná hardwarová implementace mohla přinést vyšší výpočetní rychlost, jelikož by nebylo nutné přenášet data mezi čipem FPGA a čipem ARM.

8.2 VÝPOČETNÍCH RYCHLOSTI

Výpočetní rychlosti jednotlivých implementačních možností jsou na následujících řádcích podrobněji prozkoumány. Jelikož není možné jednotlivé implementace porovnávat přímo, vzhledem k jejich daným hardwarovým omezením, je jejich výkon vždy porovnáván s integrovaným CPU ARM na desce PYNQ-Z1. Rychlost hardwarového násobení je porovnána jak vzhledem k základnímu algoritmu na CPU s výpočetní složitostí $\Theta(n^3)$, tak proti optimalizovanému algoritmu Coppersmith–Winograd knihovny Numpy v jazyku C s výpočetní náročností $\Theta(n^{2.37})$.

8.2.1 RYCHLOST MATICOVÉHO NÁSOBIČE DLE AXI

Velikost této matice je největší ze všech zvažovaných možností. V případě implementace na FPGA se využilo velké množství jednotek LUT a během syntézy proběhlo varování o vysoké latenci výpočtu, jak bylo uvedeno v 8.1.1. Z významných direktiv překladu bylo u této implementace využito pipeline 8.3.2, a to jak pro načítání dat, tak pro výpočet nejvnitřnější vnořené smyčky maticového násobiče.

Data vstupních matic byla rozdělena na 16 částí metodou 8.3.1 kde matice A a C byly rozloženy cyklicky a matice B byla rozložena na bloky. Jelikož jsou data vstupních matic rozložena do více částí paměti BRAM (celkem 48 bloků) než je reálně fyzicky adresovatelné (35 bloků) a Vivado musí zajistit při implementaci dodatečné adresy pomocí speciálních úložných bloků. Toho je docíleno vytvořením dodatečné paměti v jednotkách LUTRAM a přidruženého registru v jednotce LUT [63].

Tento přístup je náročnější na hardwarové zdroje, ale díky tomu je možné dosáhnout vyšších teoretických výpočetních rychlostí a výpočetní rychlost násobiče 128x128 byla skutečně podstatně lepší než tomu bylo v případě maticového násobení na dedikovaném CPU desky PYNQ-Z1. Detailní přehled je shrnut v Tab. 10.

Tab. 10: Výsledky výpočetní rychlosti maticového násobení matice omezené dle AXI v porovnání jak teoretického a reálného výpočetního času, tak v porovnání s výpočetním rychlostí na CPU ARM.

Použitá metoda	Teoretický výpočetní čas	Reálný výpočetní čas
FPGA	684 [μs]	842 [μs]
CPU - Naive	-	23.4 [s]
CPU - Numpy	-	5.69 [ms]

Dle očekávání jsou výpočty na FPGA podstatně rychlejší než na CPU. Oproti metodě Numpy je FPGA rychlejší 5.4x a oproti základnímu algoritmu dokonce 22 285x. Je však nutné dodat, že základní algoritmus v Pythonu je zde v opravdu velké nevýhodě vzhledem k tomu, že se jedná o interpretovaný jazyk. I přes jasné vítězství FPGA se ovšem nejedná o rychlost výpočtu, kterou by bylo FPGA v optimálním případě schopné provádět. Omezení pramenící z paměti BRAM, počtu jednotek DSP a přesunu dat dovoluje tomuto řešení dosáhnout výpočetní rychlosti až 952 Hz. Nejedná se sice o možné MHz výpočty, kterých by teoreticky FPGA mohlo dosáhnout, avšak zde využitý čip je opravdu malý. V případě většího čipu či menších matic by mělo být možné dosahovat rychlostí podstatně vyšších, pokud by se podařilo více optimalizovat přesun dat, jak je dobře vidět v 8.2.2, především u výpočetní rychlosti matice 16x16, kde rozdíl mezi teoretickou a reálně dosahovanou rychlostí je opravdu enormní.

Načtení bitstreamu a test výpočetní rychlosti je uveden v interaktivních přílohách ve složce Bitstreams. Každá implementace zde má vlastní jupyter notebook (kap. 12).

8.2.2 RYCHLOST MATICOVÉHO NÁSOBIČE DLE BRAM

Násobiče omezené dle možností BRAM byly navrženy kromě ohledu na velikost paměti BRAM také s ohledem na její reálně adresovatelné bloky, aby bylo dosaženo co nejvyšší datové propustnosti. Kromě omezení BRAM bylo přihlédnuto i na optimální využití násobiček DSP, které také zásadně ovlivňují celkovou rychlost implementace.

I přes veškerou vynaloženou snahu se ovšem nepovedlo dostatečně optimalizovat přenos mezi procesorem a programovatelnou logikou. IP blok zodpovědný za přenos dat, DMA, dokáže bohužel vždy adresovat paměťový prostor o minimální velikosti 2^{16} $(65,536)$ bitů. Z významných direktiv překladu bylo využito především pipeline 8.3.2, obdobně jako v případě násobiče omezeného dle AXI byla tato optimalizace využita jak pro načítání dat matic, tak pro výpočet nejvnitřnější vnořené smyčky maticového násobiče.

Data vstupních matic byla rozdělena na 16 adresovatelných částí a výstupní matice na 8 adresovatelných částí. Data matic A a C byla rozdělena cyklicky a pro případ násobičů o velikostech 64x64 a 32x32 byla data matice B opět rozložena na bloky. Pouze v případě násobiče 16x16 byla také data matice B rozložena cyklicky, jelikož tato velikost byla již dostatečně malá pro to, aby bylo možné lépe využít všechny dostupné hardwarové zdroje a dosáhnou tak vyšší teoretické výpočetní rychlosti.

Jelikož i pro tyto aplikace je opět využíváno více adresovatelných částí než je reálně dostupných, je opět využito dodatečné paměti v jednotkách LUTRAM s přidruženým registrem v jednotce LUT. Detailní přehled skutečných a teoretických rychlostí pro jednotlivé implementace maticových násobičů na čipu FPGA jsou uvedeny v Tab. 11 a pro porovnání jsou uvedeny výpočetní rychlosti na CPU ARM v Tab. 12.

Tab. 11: Výsledky výpočetní rychlosti násobičů o velikostech 64x64, 32x32 a 16x16 na čipu FPGA v porovnání s jejich teoretickým a reálným výpočetním časem.

Výpočetní čas FPGA	**64x64**	**32x32**	**16x16**
Teoretický výpočetní čas	99 [μs]	16 [μs]	2.2 [μs]
Reálný výpočetní čas	342 [μs]	324 [μs]	318 [μs]

Tab. 12: Výsledky výpočetní rychlosti násobičů o velikostech 64x64, 32x32 a 16x16 na CPU ARM v porovnání s výpočetní rychlostí knihovny Numpy a přístupem v Pythonu.

Výpočetní časy na CPU	**64x64**	**32x32**	**16x16**
Naive	2.75 [s]	346 [ms]	44.5 [ms]
Numpy	804 [μs]	132 [μs]	44 [μs]

Jak je v Tab. 11 vidět, rozdíl mezi teoretickým a reálným výpočetním časem je opravdu velmi markantní a přenos dat mezi pamětí procesoru a programovatelnou logikou zabírá opravdu značnou část celkového výpočetního času a značně tak omezuje možnosti využití akcelerátoru. S přihlédnutím ještě k výpočetním časům na CPU Tab. 12 je patrné, že výpočetní akcelerace na FPGA má smysl pouze pro násobič o velikosti 64x64 a více, jelikož stále přináší velmi značné zrychlení oproti přístupu knihovny Numpy.

Žádné ze zde uvedených řešení bohužel nedosahuje potencionálně možných MHz výpočtů a všechny tři implementace s omezením dle BRAM se pohybují okolo výpočetní rychlosti 3 kHz, jelikož jejich maximální výpočetní rychlost je vždy silně omezena právě datovým přesunem, jak již bylo řečeno výše.

Načtení bitstreamů a test výpočetních rychlostí je uveden v interaktivních přílohách ve složce Bitstreams. Každá implementace zde má vlastní jupyter notebook (kap. 12).

8.3 DIREKTIVY PŘEKLADU PRO VYSOKOÚROVŇOVOU SYNTÉZU

Kromě optimalizace logiky a algoritmu jako takového hrají u hardwarového programování také velkou roli direktivy překladu, neboli takzvané pragmy. Tyto pragmy nejsou součástí gramatiky programovacího jazyka C/C++ jako takového, ale nesou velmi důležité informace o způsobu chování a překladu určitých částí kódu čímž může být dosaženo nižších latencí, vylepšené datové propustnosti, anebo mohou snížit využití hardwarových zdrojů výsledného RTL kódu.

Program Vitis HLS podporuje celkem 24 různých direktiv specifických pro hardwarový překlad, které se řadí mezi 9. typů a jsou pro přehlednost shrnuty v Tab. 13 a jejich popis je detailně uveden v [63]. Pro potřeby této práce jsou zde dále detailně představeny čtyři zde nejvíce uživatelsky využívané direktivy, jejichž význam je kritický během optimalizace překladu a přináší nejvyšší změny v chování zde řešeného algoritmu. Díky rozsáhlé automatizaci překladu je velké množství direktiv automaticky aplikováno během překladu programem Vitis HLS samotným a na uživateli je tedy pouze provést vhodné optimalizace.

Tab. 13: Vitis HLS pragmy (direktivy překladu) rozřazené dle jejich typu

Typ	Atributy
Optimalizace jádra	• pragma HLS allocation • pragma HLS clock • pragma HLS expression_balance • pragma HLS latency • pragma HLS reset • pragma HLS resource • pragma HLS top
Vložení funkce	• pragma HLS inline • pragma HLS function_instatiate
Syntéza rozhraní	• pragma HLS interface • pragma HLS protocol
Task-level pipeline	• pragma HLS dataflow • pragma HLS stream
Pipeline	• pragma HLS pipeline • pragma HLS occurence
Rozvinutí smyček	• pragma HLS unroll • pragma HLS dependence
Optimalizace smyček	• pragma HLS loop_flatten • pragma HLS loop_merge • pragma HLS loop_tripcount

Optimalizace řad	• pragma HLS array_map • pragma HLS array_partition • pragma HLS array_reshape
Strukturální balíčky	• pragma HLS data_pack

8.3.1 HLS ARRAY_PARTITION

Tato direktiva rozloží jednu velkou řadu čísel na vícero malých řad či jednotlivé elementy. Na úrovni RTL dojde k využití vícero menších paměťových bloků nebo registrů místo jednoho velkého paměťového bloku, díky čemuž se efektivně rozšíří počet IO portů paměti a je tak možné dosahovat vyšší míry paralelizace. Nevýhodou zůstává, že toto rozdělení obvykle využívá podstatně větší množství paměťových bloků nebo registrů a maximální počet bloků, na které může být vstupní řada rozložena je omezena hardwarovými zdroji čipu. Ke správné funkci je nutné vložit tuto direktivu na začátek funkce v jazyce C, která načítá požadovaná data. Syntaxe včetně všech nastavení je definována dále v Alg. 7.

```
#pragma HLS array_partition variable=<name> <type> factor=<int> dim=<int>
```

Alg. 7: Syntax direktivy rozkladu číselné řady včetně všech parametrů. Kde variable odpovídá proměnné, která má být rozložena na menší části. Type je volitelný parametr a specifikuje způsob rozdělení řady. Factor určuje počet částí, na které bude řada rozložena a dim specifikuje která dimenze má být rozložena v případě multidimenzionálních řad.

Argument *variable=<name>* definuje, která řada pod kterou proměnou v jazyce C má být rozložena na vícero menších částí. Argument *type* definuje, jakým způsobem má dojít k rozložení. Na výběr je zde ze tří možností a to:

- complete – Rozloží řadu na jednotlivé elementy, kde každý jeden element bude odpovídat jednomu registru. Tato možnost je vhodná pouze pro menší řady, jelikož je velmi náročná na hardwarové zdroje.
- cyclic – Cyklické dělení vytváří menší řady prokládáním prvků z původní řady. Řada je cyklicky rozdělena tak, že do každého nového pole vloží jeden prvek, než se vrátí do prvního pole, aby celý cyklus opakoval, dokud nebude řada plně rozdělena. Například pokud bude použit *factor=3* pak,
  - Element 0 je vložen do první řady
  - Element 1 je vložen do druhé řady

- Element 2 je vložen do třetí řady
- Element 3 je opět vložen do první řady

- block – Blokové dělení vytváří menší řady z po sobě jdoucích bloků řady původní. Tím se efektivně rozdělí celá řada na N stejných bloků, kde N je celé číslo definované argumentem *factor*.

Jak již bylo naznačeno, argument *factor* definuje počet menších řad, které budou během rozložení vytvořeny a argument *dim* specifikuje, která dimenze z případné multidimenzionální řady bude rozložena na řady menší. Pokud je zvoleno *dim=0*, pak je rozložena celá multidimenzionální řada.

8.3.2 HLS PIPELINE

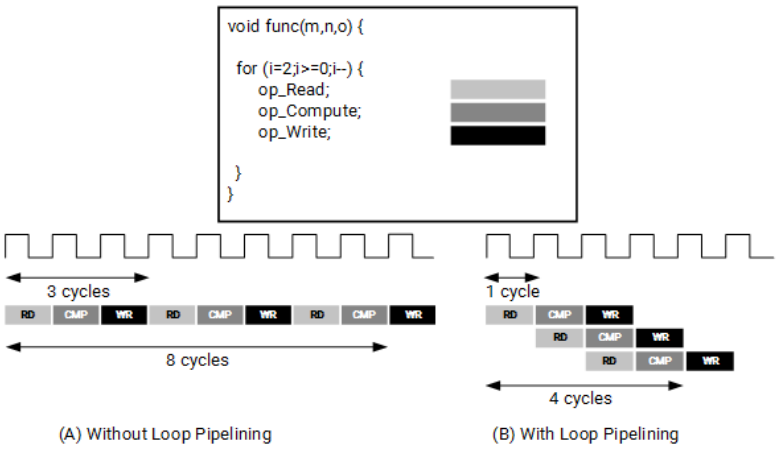
Direktiva PIPELINE slouží ke snížení inicializačního intervalu (II) mezi funkcemi nebo smyčkami tím, že umožňuje souběžné provádění většího množství operací. Zřetězená funkce nebo smyčka může zpracovávat nové vstupy každých N časových cyklů, kde N je II smyčky nebo funkce. Výchozí hodnota II pro direktivu PIPELINE je 1, což znamená, že zpracovává nový vstup každý hodinový časový cyklus. Tento interval je nutné vhodně zvolit, aby bylo fyzicky možné provést zřetězení implementovaného řešení.

Pokud je direktiva vložena uvnitř smyčky, ve které se nachází další smyčka, bude tato vnořená smyčka automaticky rozložena dle 8.3.3 nebo 8.3.4 aby bylo dosaženo nižší latence. Je však nutné pamatovat na to, že toto rozložení si vyžádá dodatečné hardwarové zdroje. Grafické znázornění principu HLS PIPELINE je uvedeno na Obr. 32. Syntaxe včetně všech nastavení je definována dále v Alg. 8.

```
#pragma HLS pipeline II=<int> enable_flush rewind
```

Alg. 8: Syntax direktivy řetězení včetně všech parametrů. Kde II odpovídá iniciačnímu intervalu v cyklech. Volitelné klíčové slovo enable_flush implementuje řetězení, které bude vyprázdněno, pokud data na vstupním kanálu budou neaktivní. Volitelné klíčové slovo rewind implementuje řetězení které umožňuje převíjet zpět nebo kontinuálně opakovat smyčku bez pauzy mezi koncem jedné iterace smyčky a začátkem další iterace.

Atribut *II=<int>* určuje požadovaný iniciační interval pro pipelinu. Program Vitis HLS se vždy snaží tento požadavek splnit, na základě závislostí v datech může mít ovšem skutečný výsledek větší iniciační interval. Pokud se nepodaří splnit požadovaný iniciační interval Vitis zvolí nejnižší možnou hodnotu a informuje uživatele.



Obr. 32: Pipelining umožňuje, aby byly operace smyčky nebo funkce implementovány souběžně. Část (A) na obrázku ukazuje výchozí sekvenční operaci, kde jsou potřeba 3 cykly mezi každým načtením vstupu a celkem je zde vyžadováno 8 cyklů než dojde k provedení posledního zápisu výstupu. Zatímco v části (B) je využito optimalizace pomocí PIPELINE a mezi každou čtecí operací je potřeba pouze jeden časový cyklus a celkem je vyžadováno pouze 4 cyklů než dojde k provedení posledního zápisu výstupu (převzato z [63]).

Volitelné klíčové slovo *enable_flush*, implementuje řetězení, které bude vyprázdněno, pokud data na vstupním kanálu budou neaktivní. Tato funkce je ovšem podporována pouze pro zřetězené funkce nikoliv pro zřetězené smyčky.

Poslední volitelné klíčové slovo *rewind*, které umožňuje převíjet zpět nebo kontinuálně opakovat danou smyčku bez pauzy mezi koncem jedné iterace smyčky a začátkem iterace další. Funkce *rewind* je aplikovatelná pouze pokud je aplikovaná na smyčku nejvýše jedné úrovně a neobsahuje žádné podmínky if-else. Tato funkce je také podporována pouze pro smyčky a nelze aplikovat na zřetězené funkce.

8.3.3 HLS UNROLL

Rozvíjením smyčky dojde k vytvoření více nezávislých operací z původně jedné operace. Direktiva UNROLL transformuje smyčky tak, že vytvoří více kopií struktury smyčky na úrovni RTL, což umožňuje provést některé nebo dokonce všechny iterace smyčky paralelně.

Smyčky v jazycích C/C++ jsou ve výchozím nastavení překladu udržovány v sekvenční podobě a syntéza vytváří logiku pouze pro jednu iteraci smyčky, která je poté sekvenčně provedena pro požadovaný počet iterací. Celkový reálný počet iterací je vždy dán logikou uvnitř smyčky (například přerušení nebo dodatečná úprava výstupu). Direktiva *unroll* může

takto všechny možné iterace rozbalit, zvýšit přístup k datům a tak zvýšit celkovou propustnost.

Pokud je požadováno celkové rozvití, je nutné předem znát hranice smyčky a tyto musí být vstupem již během kompilace. Dochází totiž k vytvoření struktur smyčky pro všechny iterace na hardwarové úrovni. Nejsou-li hranice smyčky předem známy nebo je smyčka příliš velká na celkové rozložení, je možné provést pouze částečné rozbalení, kde se specifikuje počet kopií struktury smyčky a lze takto dosáhnout alespoň částečného snížení iteračního času.

Částečné rozvinutí nepožaduje ani aby počet cílových iterací šel rozložit vždy na všechny kopie. Během syntézy Vitis HLS automaticky vytvoří logiku, která odpojí přebytečné smyčky během poslední iterace od výstupu a zachová tak plnou funkcionalitu navrženého algoritmu bez nutnosti uživatelského zásahu. Syntaxe včetně všech nastavení je definována dále v Alg. 9.

```
#pragma HLS unroll factor=<N> region skip_exit_check
```

Alg. 9: Syntax direktivy rozvíjení smyčky včetně všech parametrů. Kde factor odpovídá počtu instancí smyčky. region je volitelné klíčové slovo, které spustí rozvinutí všech vnitřních smyček bez rozvinutí smyčky, ve které se direktiva nachází. skip_exit_check je volitelné klíčové slovo, které je aktivní pouze pokud je specifikován factor a toto slovo ruší automatickou kontrolu, zda smyčka již doběhla.

Atribut *factor=<N>* specifikované nenulovou celočíselnou hodnotou indikuje, že je požadováno částečné rozvinutí, kde N je počet instancí požadované smyčky, které mají být vytvořeny. Pokud *factor* není specifikován, Vitis HLS se pokusí provést kompletní rozvití dané smyčky. Volitelné klíčové slovo *region*, které rozvine všechny vnořené smyčky bez rozvinutí smyčky, ve které se direktiva nachází. Poslední volitelné slovo *skip_region_check* je aktivní pouze, pokud je specifikována hodnota *factor*. Odstranění výstupní kontroly je závislé na tom, zda je předem znám počet iterací:

- Známé hranice – Neprovádí se kontrola podmínek ukončení, pokud je počet iterací násobkem atributu *factor*. Pokud počet iterací není celočíselným násobkem tohoto atributu, pak
    1. Je zakázáno rozvinutí dané smyčky
    2. Vitis HLS varuje, že je potřeba provést výstupní kontrolu manuálně.

- Neznámé hranice – Kontrola podmínek ukončení je odstraněna, jak bylo požadováno a uživatel musí zajistit, aby

    1. Proměnné hranice byly celočíselným násobkem zadaného atributu *factor*.
    2. Program byl schopný pracovat i bez výstupní kontroly.

8.3.4 HLS LOOP FLATTEN

Direktiva LOOP_FLATTEN slouží ke sloučení vnořených smyček do hierarchie jedné smyčky s vylepšenou latencí. Běžně v implementaci na úrovni RTL je vyžadován jeden časový cyklus pro přesun z vnější do vnitřní smyčky, nebo z vnitřní do vnitřní smyčky. Sloučením vnořených smyček je tento přesun optimalizován a je realizován jako by se stále jednalo o jednu smyčku, což šetří hodinové cykly a potencionálně umožňuje větší optimalizaci logiky smyčky. Syntaxe včetně všech nastavení je definována dále v Alg. 10.

```
#pragma HLS loop_flatten off
```

Alg. 10: Syntax direktivy sloučení vnořených smyček včetně všech parametrů. Kde off je volitelné klíčové slovo, které zakazuje sloučení dané smyčky, zatímco jiné smyčky ve stejné lokaci sloučeny být mohou.

Tato direktiva se běžně aplikuje na nejvnitřnější smyčku v hierarchii smyček, pokud tato smyčka je v takzvaném dokonalém či téměř dokonalém stavu. Neperfektní smyčky nejsou podporovány a je potřeba jejich strukturu vhodně upravit, aby splňovali podmínky dokonalých či téměř dokonalých smyček, nebo tyto neperfektní smyčky zcela vynechat pomocí vložení direktivy s klíčovým slovem *off*. Dokonalé a téměř dokonalé smyčky se vyznačují následujícími specifikacemi:

- Dokonalé smyčky
    - Pouze nejvnitřnější smyčka má nějaký obsah
    - Pouze nejvnitřnější smyčka obsahuje jakoukoliv logiku
    - Všechny hranice všech smyček jsou konstantní
- Téměř dokonalé smyčky
    - Pouze nejvnitřnější smyčka má nějaký obsah
    - Pouze nejvnitřnější smyčka obsahuje jakoukoliv logiku
    - Hranice všech smyček nejsou konstantní

9 APLIKACE K DETEKCI NEOBVYKLÝCH STAVŮ

V této kapitole jsou uvedeny implementované metody a především způsob jejich nastavení ve vyvinuté aplikaci k detekci s možnou akcelerací na čipu FPGA. Je zde popsán způsob ovládání aplikace v uživatelském prostředí Jupyter Notebook, ale i způsob provádění změn přímo v konfiguračním souboru či pomocí konzole. Dále je zde stručně vysvětlen způsob nastavení jednotlivých částí detektoru v grafickém rozhraní se vztahem na zde vysvětlené problematiky a jejich doporučené nastavení.

9.1 SPUŠTĚNÍ APLIKACE POŽADOVANÉM MÓDU

Vyvinutá aplikace jako taková má tři módy. Prvním je konzolový běh, který je nastaven jako výchozí metoda spuštění a tento mód je zamýšlen k testování SW na výpočetním PC. Další možností je spuštění detektoru v Jupyter módu. Tento mód je primárně zamýšlen pro výpočetní PC k otestování uživatelského prostředí a je ho možné spustit již na vývojové desce PYNQ-Z1 či jiném embedded zařízení. Není zde ovšem spuštěna podpora násobiče na FPGA či podpora externí simulace a veškeré výpočty tak stále probíhají na CPU. Posledním módem je Jupyter runtime, který je možné spustit pouze na desce PYNQ-Z1. V tomto módu je aktivní akcelerátor maticového násobení na FPGA a data do systému vstupují pomocí komunikačního rozhraní SPI nebo UART.

9.1.1 KONZOLOVÝ MÓD

Provádí se pomocí příkazové řádky, jak je uvedeno v Alg. 11, a všechny důležité kroky ke spuštění jsou plně automatizovány. Po automatické inicializaci vyskočí okno s živou simulací dle nastavení zapsaných v config.py. Parametry simulace lze i přes konzoli během jejího běhu libovolně měnit pomocí změn parametrů uvnitř vnořené třídy detektoru, jak je příkladně uvedeno ve spodní části Alg. 11.

```
from detector import Detector
# Start detector in console mode with simulator
det = Detector()
det.start()

# Change parameters of simulation in console mode
det._simulator.kp = 3.6
```

Alg. 11: Import a spuštění detekčního SW v konzolovém módu. Veškeré nezbytné kroky jsou plně automatizovány. Po zavolání vyskočí okno knihovny matplotlib s animaci živé simulace. Dále je zde uveden příklad změny parametru během simulace. Změna jeho nastavení je ošetřena na úrovni třídy a není tedy nutné se starat o běh simulace (její zastavení a opětovné spuštění).

Využití tohoto módu je především testování implementovaných algoritmů a jejich celkové funkčnosti na výpočetním PC nikoliv na desce PYNQ-Z1. Žádné z algoritmů v tomto módu nejsou akcelerovány pomocí vyvinutých maticových násobičů a vše je generováno lokálně pomocí CPU.

9.1.2 JUPYTER MÓD

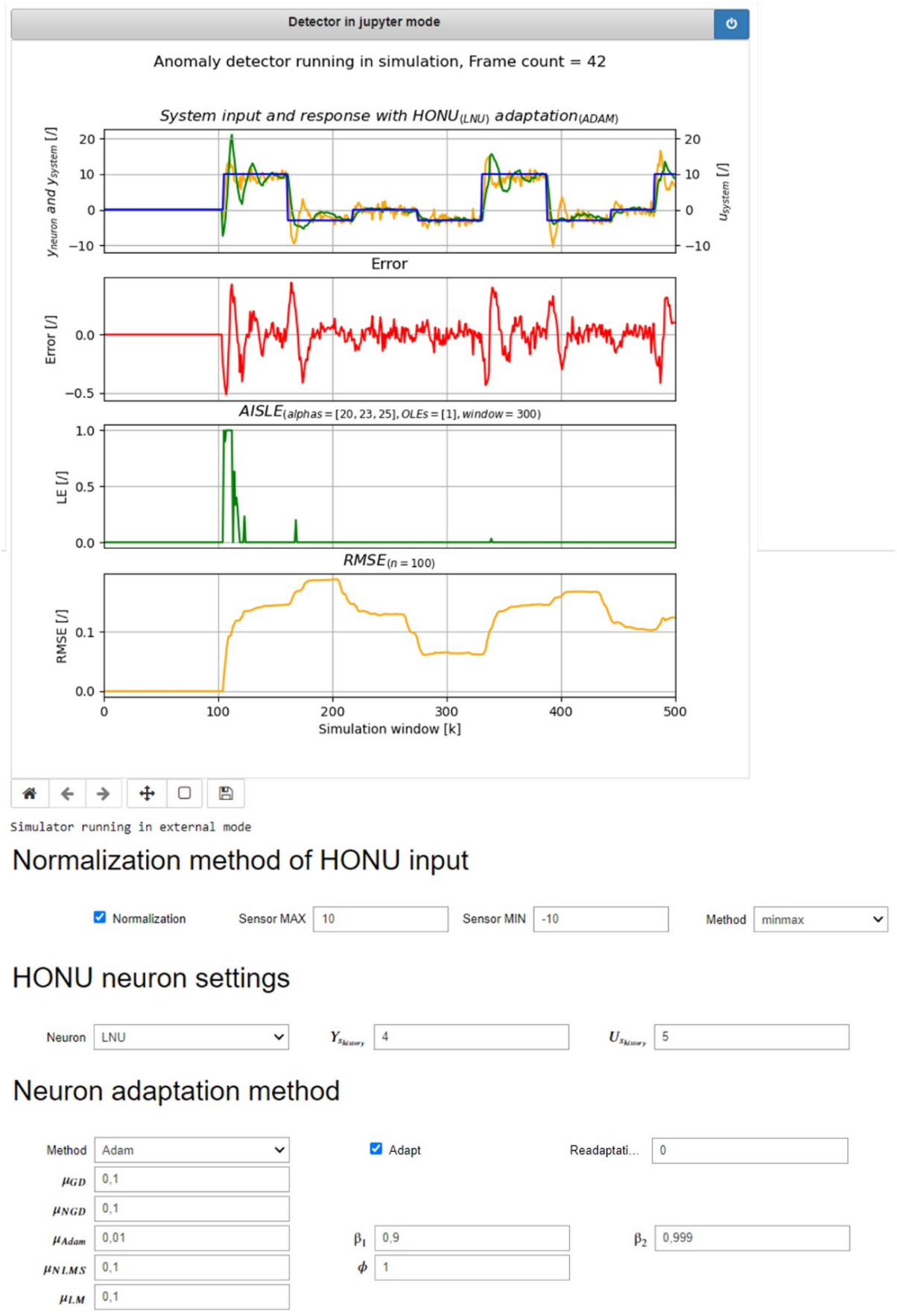
Jupyter mód na rozdíl od konzolového módu poskytuje grafické rozhraní pro snadnou změnu všech parametrů HONU, detekčních algoritmů a parametrů lokální simulace. Všechny inicializační kroky opět probíhají automaticky. Ke spuštění je tedy pouze nutné zadat příkaz do Jupyter notebooku dle Alg. 12. Poté je již možné všechny ostatní parametry měnit přímo pomocí grafického rozhraní a následně celý detektor spustit.

```
%matplotlib notebook
from detector import Detector
det = Detector(mode='jupyter')
```

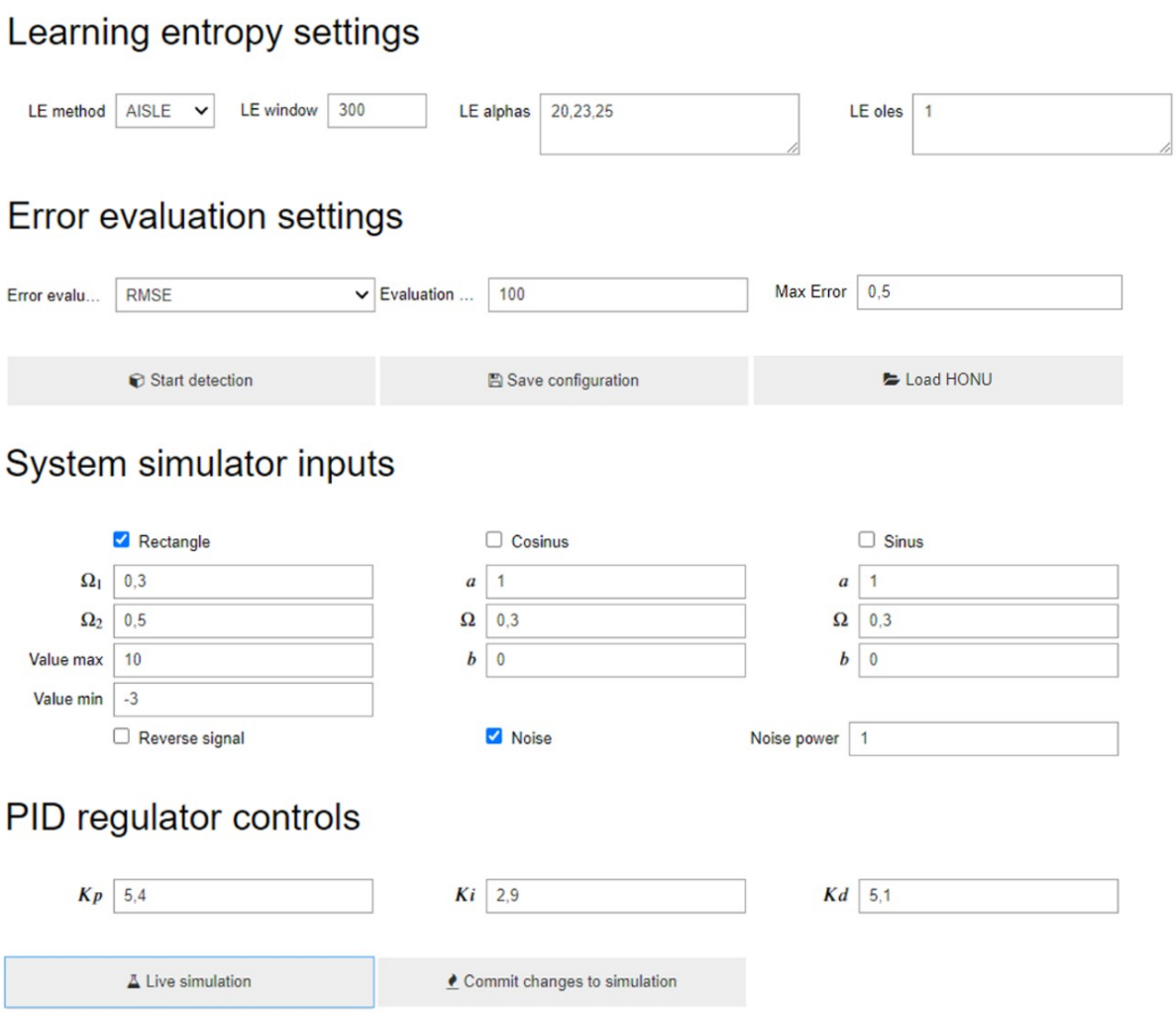
Alg. 12: Import a spuštění detekčního SW v Jupyter módu. Veškeré nezbytné kroky jsou plně automatizovány. Po inicializaci se zobrazí okno, kde bude zobrazena nejbližší historie simulované detekce a pod ní všechna dostupná nastavení pro HONU a simulátor.

Celé grafické rozhraní je zobrazeno na Obr. 33 a Obr. 34. Jelikož je celé toto rozhraní v Jupyter notebooku, je k němu přistupováno pomocí internetového prohlížeče a bohužel není možné vše vykreslit na jedné obrazovce najednou. Výhodou však je, že k tomuto rozhraní je možné přistupovat například i z mobilního telefonu či tabletu.

Nahoře celého GUI je zobrazena aktuálně probíhající detekce vykreslená pomocí grafické knihovny matplotlib. Šířka monitorovaného okna je nastavena dle parametru *sim_xlim* v config.py a není možné jí po inicializaci detektoru nikterak měnit.



Obr. 33: Uživatelské prostředí vyvinuté aplikace ve frameworku Jupyter s vizualizací průběhů detekčních algoritmů 1. část.



Obr. 34: Uživatelské prostředí vyvinuté aplikace ve frameworku Jupyter s vizualizací průběhů detekčních algoritmů 2. část.

V tomto grafickém okně (Obr. 33) je úplně nahoře zobrazen mód ve kterém detektor aktuálně běží a na jakém simulovaném snímku se nachází. Jeden snímek pro zde prezentované nastavení odpovídá devíti hodnotám, a to z toho důvodu, aby bylo dosahováno rozumných simulačních časů, jelikož vykreslení samotné potřebuje relativně velké množství strojového času a vykreslení po jednom vzorku by bylo velmi hardwarově náročné. Počet hodnot na jeden snímek lze změnit pomocí nastavení *sim_frame_batch* v konfiguračním souboru. Dále jsou zde zobrazeny čtyři grafy, kde v prvním je zobrazen simulovaný systém s jeho vstupem a aproximace pomocí zvoleného HONU se zvolenou adaptační metodou. Poté je zde zobrazena chyba aproximace systému, výstup detekční metody learning entropy a zvolená chybová metrika k detekci dlouhodobě trvajících anomálií.

Pod grafy se již nachází veškeré nastavení, které je možné změnit po základní inicializaci detektoru. Jako první je zde nastavení normalizace vstupních dat do HONU. K

dispozici jsou připraveny algoritmy pro metodu MinMax a Z-score, které je možné zvolit v kontextovém menu. Normalizaci je nutné vhodně zvolit dle vstupních dat, jinak budou detekční schopnosti SW jen velmi limitované.

Dále se zde nachází nastavení neuronu typu HONU. V kontextovém menu jsou na výběr jednotlivé polynomiální stupně, tedy LNU, QNU a CNU a volba historických hodnot na jejich vstupu. Poté je zde možné zvolit metodu požadované adaptace jakožto i vypnout či zapnout adaptaci na živé simulaci. Kromě toho zde lze navolit interval pro pravidelné přeučování, kde nula znamená nikdy nepřeučovat. Dostupné metodiky lze zobrazit pomocí kontextového menu *method* a volit je možné mezi Gradient Descent, Normalized Gradient Descent, Adam, Normalized Least Mean Squares a Levenberg-Marquardt, což jsou všechno metody detailně popsané v 5.2.

Předposlední nastavení detektoru spadá již do kategorie detekcí, konkrétně learning entropy. Na výběr je zde mezi metodami AISLE a ABSLE jak je vysvětleno v 6.1. Pro funkční detekci je nutné vhodně zvolit na základě vstupních dat délku vyhodnocovaného okna a k tomu vhodně zvolit jak vektor citlivost, tedy LE alphas, tak stupně vyhodnocení neboli LE oles. Jednotlivé hodnoty alphas a oles se oddělují čárkou a musí být vždy zapsány od nejmenšího po největší, jinak detekce pomocí learning entropy nebude fungovat správně.

Poslední nastavení detektoru je také z kategorie detekcí a to z vyhodnocení dlouhodobě trvajících anomálií. V kontextovém menu je na výběr ze tří metod, jak jsou popsány v 6.2, a to tedy RMSE, MAE a Evrg, kde první dvě metody jsou běžně používané a třetí metoda je metodou vlastní. Správná funkce je zajištěna, pokud je vhodně nastaveno jak vyhodnocované okno, tak maximální povolená chyba.

Kliknutím na tlačítko S*tart Detection* dojde k uložení konfigurace do konfiguračního souboru a spuštění detektoru se simulátorem. Současně se spuštěním dojde k zablokování změny parametrů, které nejsou měnitelné během již spuštěné detekce, jako je například volba historických vstupů do HONU nebo typ neuronu. Tlačítko *Save configuration* slouží k ukládání a změně parametrů, které je možné měnit během již spuštěné detekce, a k uložení aktuálních vah a typu zvoleného neuronu. Poslední tlačítko *Load HONU* slouží k načtení již dříve upečeného HONU.

Další možnosti nastavení jsou již identické s dedikovaným simulátorem 7.1 a všechna jeho nastavení je možné měnit během již probíhající detekce pomocí tlačítka *Commit changes to simulation.*

9.1.3 JUPYTER RUNTIME MÓD

Tento mód je možné spustit pouze na desce PYNQ-Z1, poněvadž již přijímá vstupní data pouze skrze rozhraní SPI na GPIO portech či UART přes port USB a v tomto módu je již možné aktivovat akceleraci násobiče na FPGA. Graficky se tento mód od Obr. 33 neliší, pouze je změněna jeho inicializace, během které se dodatečně aktivuje akcelerace, jak je uvedeno v Alg. 13 a na Obr. 34 v tomto módu chybí možnost změny nastavení simulátoru, které je nyní nutné provádět na externím zařízení.

```
%matplotlib notebook
from detector import Detector
det = Detector(mode='runtime', accelerate=True)
```

Alg. 13: Import a spuštění detekčního SW v Jupyter módu s akcelerací na FPGA a příjmem dat z externího zdroje. Nastavení vstupního zdroje musí být definováno předem v konfiguračním souboru. Veškeré nezbytné kroky jsou plně automatizovány. Po inicializaci se zobrazí okno, kde bude zobrazena nejbližší historie detekce a pod ní všechna dostupná nastavení pro HONU.

Způsob komunikace s externím systémem nebo simulátorem je nutné definovat v konfiguračním souboru pod proměnnou *external*. Tato proměnná musí být ve formátu *list* kde na první pozici je metoda a na druhé pozici je počet vstupů se senzorů, které budou přenášeny. Nastavení této proměnné vstupují do detekčního SW již během jeho inicializace a nemůžou být libovolně změněny pokud již detektor běží. Aktuálně jsou podporovány pouze dvě vstupní metody, a to *UART* a *SPI*. Pro potřeby propojení simulátoru a detektoru se více osvědčil UART i přes jeho nižší teoretickou propustnost, a to především z toho důvodu, že je možné jej připojit přímo k PS části celého systému, kdežto SPI musí s procesorem komunikovat ještě přes mezičlen microblaze a zpomaluje se tak celkový přenos.

9.2 ÚPRAVA DAT

U některých vstupních dat může být, pro lepší funkčnost detekce anomálií, provést úpravu dat vyhlazením pomocí metody coarse graining[7] dle

$$y_{cg}(k) = \frac{1}{2r+1} \sum_{i=-r}^{2r+1} y(k-i). \tag{9.1}$$

V této metodě se nastavuje z kolika hodnot r (radius) okolo hodnoty $y(k)$ se vypočte aritmetický průměr, který se dosadí za hodnotu $y(k)$. Je ovšem důležité nepřehánět s

7 Coarse Graining – Metoda klouzavého průměru

parametrem r. Doporučené hodnoty dle testování v této práci jsou pro $r \in (0, 10)$, kde $r = 0$ znamená zachování původních hodnot. Tato úprava se nastavuje pouze v konfiguračním souboru a nelze ji během běhu detektoru měnit.

Dále je data nutné vždy normalizovat dle MinMax či Z-Score. Jedním z hlavních důvodů je, aby se srovnaly vlivy jednotlivých vstupních veličin tak, aby vstupní veličiny pohybující se v různých řádech jednotek byly na stejném rozmezí. Obecně se v této práci více osvědčila metoda MinMax, jelikož je možné jí jednoduše nastavit dle rozsahů senzorů a stále zajistit, že se všechny vstupní hodnoty budou pohybovat na intervalu $(0, 1)$.

9.3 STABILITA GRAFICKÉHO ROZHRANÍ V JUPYTERU

Program je velmi stabilní, dodržuje-li uživatel tato pravidla:

- Zadává hodnoty do textových polí ve správném formátu. Ačkoliv je valná většina polí vybavena kontrolou vstupu a zakázáním všech nepovolených hodnot, některá vstupní pole, jako například *Alphas*, pro potřeby LNU tuto kontrolu nemají vzhledem k principu jejich fungování.
- Dodržuje kauzalitu. Toto je důležité především pro dávkově zpracovávaná data, a to i přesto, že program je vybaven řadou ochran proti kumulovaným požadavkům. Nicméně je pravděpodobné, že program obsahuje v tomto směru celou řadu bugů[8] a uživatel by tak kupříkladu neměl žádat program o výsledky výpočtu, který ještě neproběhl.
- Nekliká velmi rychle na tlačítko *live simulation*. Program je navržen tak, aby fungoval plynule a řada výpočtů je akcelerována pomocí čipu FPGA a rychlé přepínání simulace může vést až k pádu celého programu.

Bohužel se v této práci nepodařilo odstranit dva známé bugy (jejichž původ je zřejmě již v původních knihovnách Pythonu). Jedná se o bug v grafickém zobrazení průběhu monitorování na živých datech. Graf má interaktivní lištu s nástroji, se kterými je možné manipulovat s grafem. Pokud se ovšem zastaví simulace, přiblíží se některý z grafů a simulace se opětovně spustí dojde velice často k zamrznutí celého programu. V méně častém případě dojde k resetování nastavení zobrazení, ale bez pádu programu. Je proto nezbytné, aby uživatel po přiblížení a před opětovným spuštěním vždy klikl na *domeček* a nechal tak

8 Bug – chyba v počítačovém programu

program nastavit výchozí zobrazení, které je stabilní. Další bug vzniká na komunikačním rozhraní SPI. Pokud je na lince dlouhou dobu klid (déle než 10min), přestane program na daném portu poslouchat a jakákoliv budoucí data nebudou zaznamenána. Bug má původ zřejmě v nativní knihovně pythonu s ohledem na některý z time-outů, a i přes značné úsilí k vyřešení tohoto problému se toto bohužel nepovedlo. Proto je nezbytné při využití SPI k přenosu dat začít data posílat dostatečně rychle, nebo lépe využít UART a usnadnit si tak práci.

10 TESTOVÁNÍ PLATFORMY S NAVRŽENÝMI A IMPLEMENTOVANÝMI ALGORITMY

V této kapitole je ověřena funkčnost vyvinutého softwaru a jeho možnosti, jak na umělých datech ze zde vytvořeného simulátoru, tak je ověřena funkčnost dávkového zpracování a vyhodnocení reálných dat od společnosti Robert Bosch, s.r.o. v konzolovém módu. Dále je zde provedeno porovnání výpočetní rychlosti mezi čistě CPU přístupem a akcelerací některých výpočtů detekčního softwaru na čipu FPGA. Toto porovnání je vyhodnoceno pro jednoduchost na základě rychlosti nových simulačních snímků na vývojové desce PYNQ-Z1.

10.1 ONLINE ZPRACOVÁVANÁ DATA ZE SIMULÁTORU

Simulátor dynamického systému byl pro účely této práce nastaven přesně tak, jak je prezentován v kapitole 7.1 na Obr. 16. Jedná se tedy o simulaci dynamického systému druhého řádu, který je buzen obdélníkovým signálem o $\Omega_1 = 0.3$ a $\Omega_2 = 0.5$, kde horní limit buzení je 10 a spodní limit buzení je -3. PID regulátor byl také nastaven obdobně, tedy pro proporcionální složku $K_p = 5.4$, integrační složku $K_i = 2.9$ a derivační složku $K_d = 5.1$. Hodnoty konstant pro PID regulátor byly nastaveny na základě experimentální metody, a ačkoliv by bylo zřejmě možné dosáhnout přesnější regulace, není to pro účely této práce podstatné a ani nikterak důležité. Regulace dynamického systému zde byla zavedena především proto, aby byl lépe reprezentován možný skutečný systém a simulátor tak lépe popisoval možný reálný průběh. Opět je ovšem nutné podotknout, že nastavení a práce s PID regulátorem není předmětem této práce a je zde využit čistě k demonstračním účelům. Signál

jako takový je vždy zatížen alespoň drobným šumem, kde odstup signál-šum je roven 90dB, pokud není uvedeno jinak.

10.1.1 PŘÍPRAVA DETEKTORU ANOMÁLIÍ

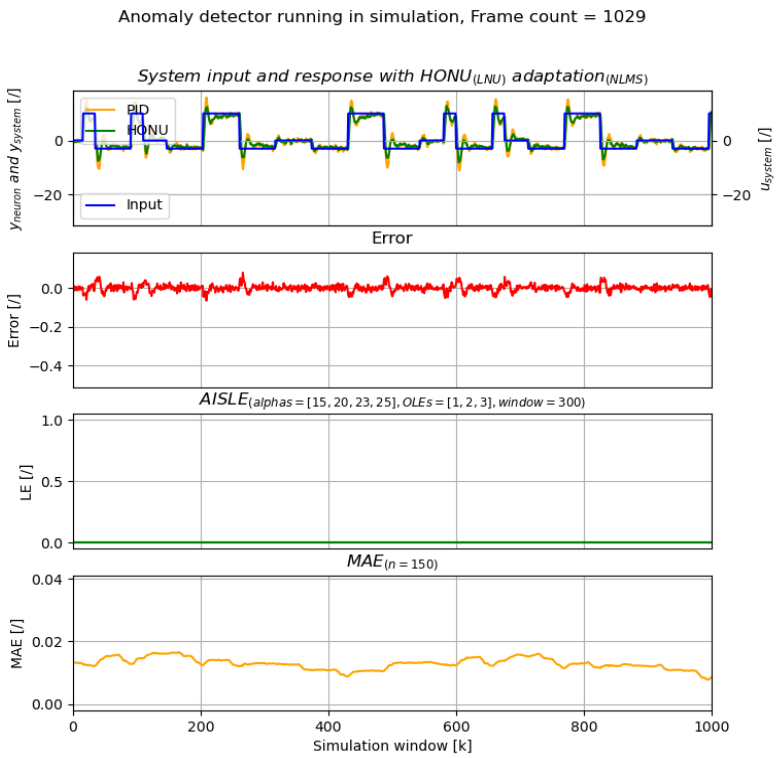
Nejprve ze všeho se musí detektor správně natrénovat na chování daného systému. Vzhledem k tomu, že je známo, že sledovaný systém je druhého řádu, lze očekávat přesnější aproximace pomocí neuronu typu QNU, avšak i přesto je vhodné vždy ověřit, zda nebude postačovat aproximace pomocí LNU vzhledem k podstatně jednodušším výpočtům a díky tomu vyšší teoretické výpočetní rychlosti společně s obvykle lepší robustností celého řešení. Všechna zde prezentovaná řešení využívají automatické ukončení učení na základě $\mathbf{MAE}$, jak je uvedeno v 6.2, jelikož se tato metoda ukázala jako nejvhodnější pro daný systém. Automatické ukončení adaptace je dále určeno jako

$$adapt_{end} = \begin{cases} True & \text{if } \sum_{k=0}^{N} \mathbf{MAE}_{hist} < goal \\ False & \text{else} \end{cases}. \tag{10.1}$$

K tomuto vyhodnocení bylo přikročeno především proto, aby oblasti signálu okolo nulové hodnoty předběžně neukončily adaptaci a bylo tak co nejvíce zajištěno nejlepší možné natrénovaní neuronu. Počet historických hodnot $\mathbf{MAE}$ byl na základě testování zvolen jako 150, což bylo postačující pro všechny zde prováděné experimenty. Nejprve byla provedena aproximace pomocí neuronu typu LNU. Tento neuron byl na základě experimentů nastaven na hodnoty dle Tab. 14 a výsledky jeho aproximace jsou uvedeny v Obr. 35.

Tab. 14: Nastavení LNU pro aproximaci simulovaného dynamického systému adaptační metodou NLMS. Hodnoty byly určeny experimentální metodou.

y_{hist}	u_{hist}	μ	ε	$\mathbf{MAE}_{target}$
5	5	0.3	1.0	1.8

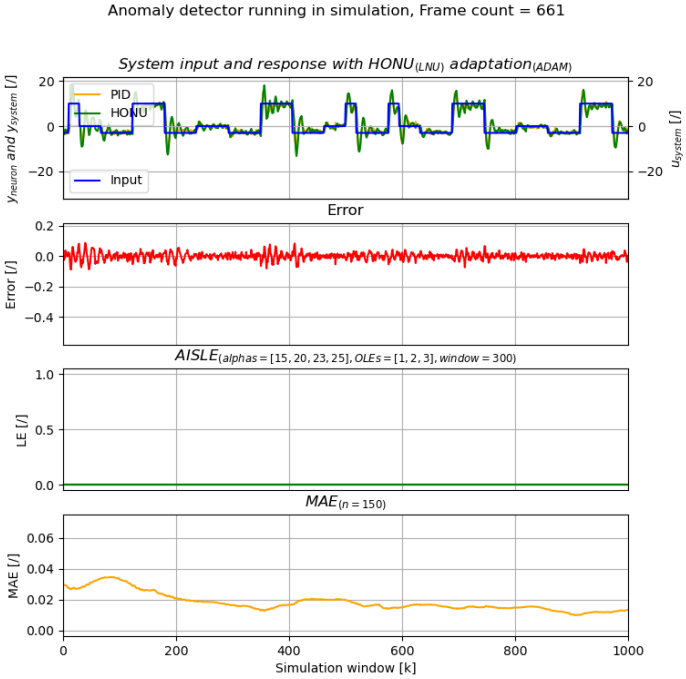


Obr. 35: Aproximace simulovaného dynamického systému pomocí neuronu typu LNU nastaveného dle Tab. 14. LNU dokázalo relativně dobře aproximovat požadovaný systém v rámci jeho možností. Bohužel však nedokázalo dostatečně věrohodně podchytit reakci systému na skokovou změnu buzení a ani při dalším učení nedojde ke zlepšení této aproximace. Od neuronu QNU jsou v tomto ohledu očekávány lepší výsledky, jak je názorně ukázáno v Obr. 37.

Je patrné, že základní chování systému dokáže neuron relativně dobře aproximovat. Avšak má problémy v oblasti skokových změn budícího signálu, kde simulovaný systém má relativně znatelný překmit, který není neuronem typu LNU podchycen a je zde spíše tlumen. Tato aproximace je také nejlepší možná, jaká se s tímto neuronem podařila dosáhnout a ani další učení již přesnost nezlepšilo. Hodnota ukončení automatického učení tak nemůže být dle metodiky popsané výše o moc nižší než 1.8, jelikož by nedošlo k jejímu dosažení a systém by se tak nepřestal adaptovat. Učící metoda a její parametry byly zvoleny na základě experimentální metody, kde se přístup NLMS ukázal jako nejlepší i přes delší dobu adaptace, která k dosažení požadované chyby trvala 1029 simulačních snímků, což odpovídá 9261 updatům vah. Pro porovnání je na Obr. 36 uvedena adaptace pomocí algoritmu Adam nastaveného obdobně jako NLMS dle Tab. 15.

Tab. 15: Nastavení LNU pro aproximaci simulovaného dynamického systému adaptační metodou Adam. Hodnoty byly určeny experimentální metodou.

y_{hist}	u_{hist}	μ	β_1	β_2	$\mathbf{MAE}_{target}$
5	5	0.01	0.9	0.99	1.8

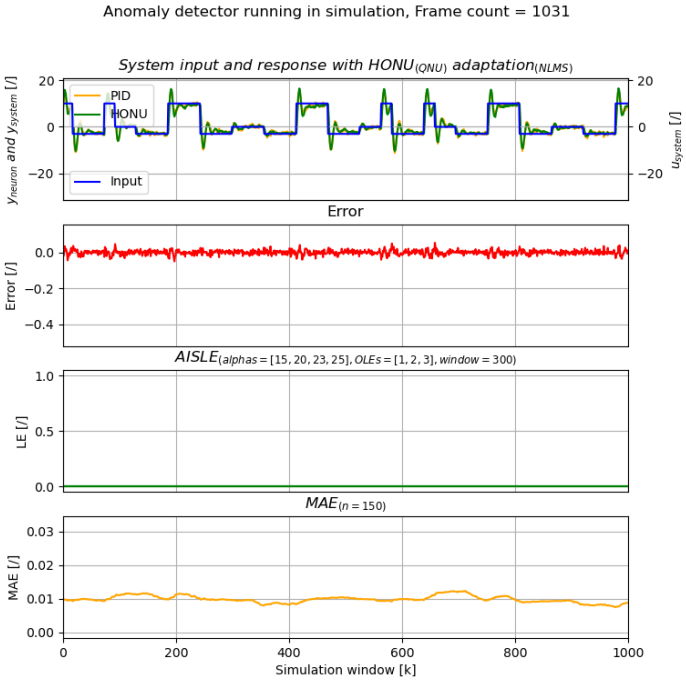


Obr. 36: Aproximace simulovaného dynamického systému pomocí neuronu typu LNU nastaveného dle Tab. 15. Adaptační algoritmus Adam dokázal oproti NLMS lépe aproximovat reakce systému na skok vstupního buzení. Bohužel kromě toho ale výstup neuronu trpí značnými oscilacemi. Trénování jako takové bylo rychlejší o 368 snímků oproti NLMS k dosažení stejného hodnotícího kritéria, avšak podávané výsledky nejsou vhodné k použití takto natrénovaného neuronu k detekcím anomálií.

Algoritmus Adam oproti NLMS dosahuje rychleji cílové chyby, přesně o 368 snímků, což odpovídá 3312 updatům vah. Tento algoritmus také mnohem lépe aproximuje reakce systému na skokovou změnu vstupního buzení a nedochází zde k jejich utlumení, jako je tomu v případě algoritmu NLMS. Bohužel ovšem výstup z neuronu trpí na značné oscilace, a je tak ne příliš vhodný k použití pro detekci anomálií. I přes značné úsilí a vynaložený čas se nepodařilo nalézt alternativní nastavení hyperparametrů algoritmu Adam, které by tento problém vyřešilo. Kromě toho, řešení v podobě neuronu typu QNU s adaptací algoritmem NLSM nastaveného dle parametrů uvedených v Tab. 16, dosahovalo velmi dobrých výsledků, jak je dobře vidět na Obr. 37. Nebyl tak vynakládán další čas na optimalizaci aproximace pomocí neuronu typu LNU.

Tab. 16: Nastavení QNU pro aproximaci simulovaného dynamického systému adaptační metodou NLMS. Hodnoty byly určeny experimentální metodou.

y_{hist}	u_{hist}	μ	ε	$\mathbf{MAE}_{target}$
7	3	0.2	1.0	1.3



Obr. 37: Aproximace simulovaného dynamického systému pomocí neuronu typu QNU nastaveného dle Tab. 16. QNU dokáže chování dynamického systému aproximovat velice dobře a oproti LNU se jedná o velké zlepšení v oblasti přesnosti. I přes vyšší výpočetní náročnost bude tedy vhodné využít tento typ neuronu v aplikacích detekcí anomálií. Pomocí tohoto neuronu lze spolehlivě nastavit hodnotu automatického ukončení učení na 1.3 oproti 1.8 u neuronu LNU pro metodiku hodnocení chyby popsanou výše.

Neuron typu QNU dokáže aproximovat chování simulovaného dynamického systému podstatně lépe než tomu bylo v případě neuronu typu LNU. Není zde již tlumení reakce systému na skokovou změnu buzení nýbrž je tento skok věrohodně aproximován. Tato aproximace je také nejlepší možná, jaké se zde podařilo dosáhnout. Hodnota ukončení automatického učení tak nemůže být dle metodiky popsané výše o moc nižší než 1.3, jelikož by nedošlo k jejímu dosažení a detektor by se nepřestal adaptovat. Učící metoda a její parametry byly opět zvoleny na základě experimentální metody, kde se opět přístup NLMS ukázal jako nejlepší. V případě dříve uvedeného algoritmu Adam sice učení neuronu QNU probíhalo opět rychleji než tomu bylo v případě NLMS, avšak takto adaptovaný neuron trpěl na kmitání podstatně více než neuron typu LNU a nebylo by tedy vhodné jej použít.

Pro účely detekce anomálií je v této práci tedy využit neuron typu QNU adaptovaný algoritmem NLMS s automatickým ukončením adaptace pro chybu menší než 1.3 dle metodiky popsané výše. Nastavení vstupů a hyperparametrů adaptačního algoritmu je provedeno přesně dle Tab. 16. Nastavení parametrů detektoru jako takového ať již pro náhlé anomálie detekované pomocí AISLE, tak dlouhodobé anomálie detekované například pomocí MAE jsou nastaveny a otestovány v následující kapitole.

Zde použité nastavení je uloženo ve složce sample_config v souboru config_QNU_NLMS.py, (kap. 12). Po překopírování tohoto konfiguračního souboru a přejmenování pouze na config.py lze zde nastavený detektor spustit pomocí uživatelského prostředí, které bylo vyvinuto v Jupyter Notebooku.

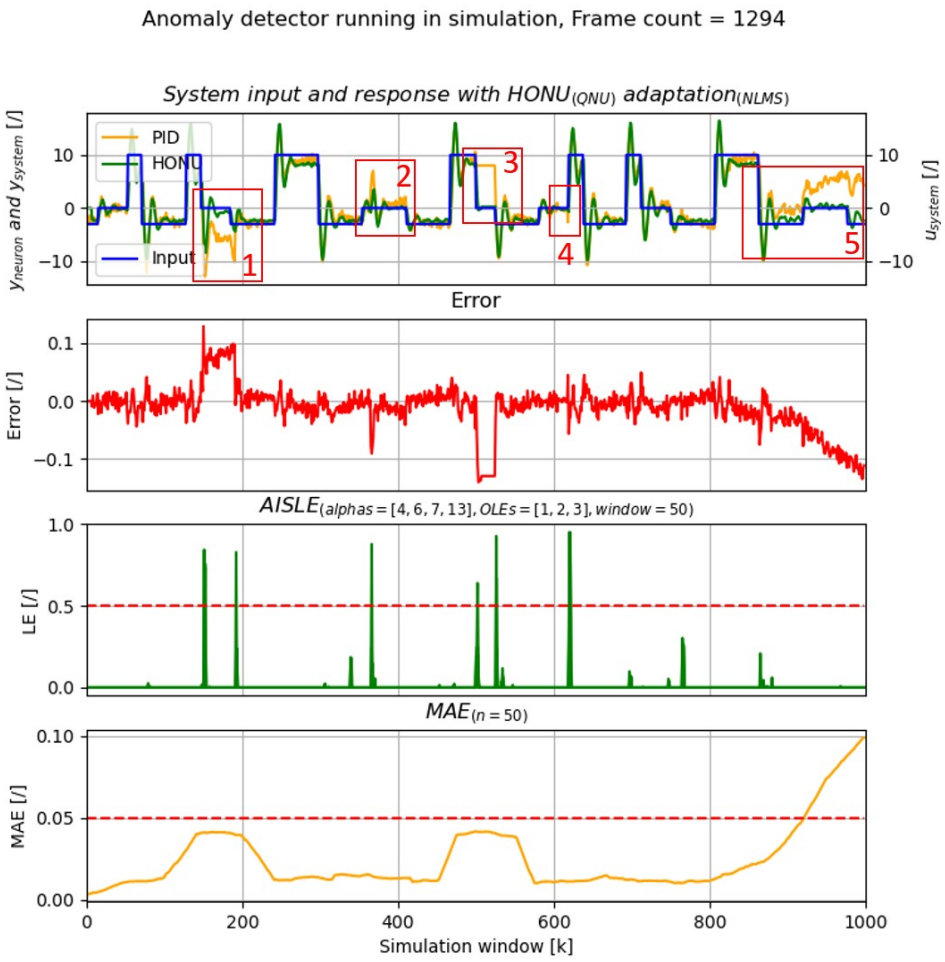
10.1.2 DETEKCE ANOMÁLIÍ

Po nastavení vhodné aproximace a natrénování neuronu je čas na vhodné nastavení detekčních metodik. Toto nastavení je obvykle nutné provést pro každý jeden systém individuálně, jelikož jeho chování i kvalita senzoriky se může lišit od testovaného vzorku a detektor by poté mohl chybně detekovat některé stavy. Vzhledem k využití simulátoru pro účely této práce, je toto do jisté míry usnadněno.

Od vhodně nastavené detekční metody AISLE se očekává, že bude dobře detekovat neočekávaný skok v datech, případný fázový posun vstupního signálu, výpadek signálu senzoru a další neočekávané stavy. Náhodné poruchy se do zde vyvinutého simulátoru musí zanášet skrze příkazový řádek a není k nim vyvinuto grafické rozhraní. Kromě přímého ovlivnění simulace je také možné poruchám nastavit počáteční a konečný simulační snímek, ve kterém bude porucha simulována. Na základě experimentální metody byly detekce pomocí AISLE a detekce dlouhodobých poruch nastaveny dle Tab. 17. Toto nastavení bylo zvoleno tak, aby co nejlépe detekovalo všechny testované anomálie a pokud možno co nejvíce potlačovalo detekce skokových změn budícího signálu, které by se za anomálii z pohledu LE dalo také považovat. Grafické vyhodnocení detektoru je zobrazeno na Obr. 38.

Tab. 17: Zvolené nastavení detekční metody AISLE pro vstupní simulovaný systém a simulované poruchy.

$Alphas$	$OLEs$	$Window_{AISLE}$	$Window_{MAE}$
4, 6, 7, 13	1, 2, 3	50	50



Obr. 38: Testovací sekvence nastaveného detekčního softwaru. Do dat je zaneseno celkem 5 různých poruch, které mají být úspěšně detekovány. První porucha představuje neobvyklý drift výstupu systému s jinak běžným chováním. Druhá porucha představuje náhodné zvýšení proporcionální složky regulátoru. Třetí porucha simuluje zaseknutý senzor na fixní hodnotě. Čtvrtá porucha představuje neobvyklou reakci systému na skokovou změnu vstupu a pátá porucha simuluje pozvolné zvyšování výstupu systému. První čtyři poruchy byly detekovány pomocí AISLE, pátou poruchu se podařilo detekovat pomocí MAE.

Experimentálně nastavený detektor byl schopný detekovat všechny zde testované anomálie pomocí metod AISLE nebo MAE. Anomálie označená číslovkou 1 byla zanesena na interval vzorků 150 – 190 a představuje neobvyklý drift výstupu systému s jinak obvyklým chováním. U této anomálie byl úspěšně detekován jak její počátek, tak konec pomocí detekční metody AISLE, která zde zřetelně překročila nastavenou detekční úroveň. Pro metodu MAE byla tato anomálie stále pod detekční hodnotou.

Další anomálie označená číslovkou 2 byla zanesena na interval 365 – 410 a představuje náhodné zvýšení proporcionální složky regulátoru. Zde byl metodou AISLE detekován pouze

počátek anomálie, nikoliv její konec. Metoda MAE na tuto anomálii prakticky vůbec nereagovala.

Anomálie označená číslovkou 3 byla zanesena na interval vzorků 500 – 525 a přestavuje zaseknutý senzor na fixní hodnotě. Tento druh anomálie byl opět detekován na jejím počátku a konci pomocí metody AISLE, i když je nutné poznamenat, že detekce počátku této anomálie je téměř na nastavené detekční hraně a nebyla tedy detekována dostatečně jednoznačně. Metoda MAE tuto anomálii na nastavené detekční hraně opět nedetekovala a to i přes to, že dle průběhu chyby predikce je velikost chyby v porovnání se zbylým průběhem vskutku enormní.

Anomálie číslo 4 byla zanesena na interval 615 - 620, kde došlo ke změně reakce systému na skokovou změnu. Systém tak na skok vstupního buzení „nahoru“ nejdříve reagoval drobným poklesem a poté se vrátil k jeho běžnému chování. Tato anomálie byla pomocí detekční metody AISLE dobře detekována, zatímco v metodě MAE tato anomálie nebyla zaregistrována vůbec.

Poslední anomálií s číslem 5 je pozvolné zvyšování výstupní hodnoty systému, které má simulovat dlouhodobě trvající chybu. Toto pozvolné zvyšování výstupu systému není metodou AISLE detekováno vůbec, je však dobře detekováno pomocí metody MAE, která je zde užita právě k detekci pozvolně vznikajících anomálií. Anomálie jako taková sice není detekována na jejím počátku, ale je stále detekována dostatečně brzo na to, aby systém včas informoval obsluhu.

Detektor jako takový se na základě zde provedeného testu osvědčil jako spolehlivý. Prakticky nereaguje na běžnou skokovou změnu vstupního signálu, zatímco dokáže spolehlivě odhalit testované anomálie. Pokud by se využívala detekce pouze na základě metody learning entropy, nebylo by zřejmě možné odhalit dlouhodobě trvající anomálie, jak dokázala ověřit i testovaná anomálie číslo 5, na kterou metoda AISLE vůbec nezareagovala. Pro kompletní záruku spolehlivosti by bylo vhodné detektor otestovat i na reálných datech. Tato možnost se bohužel během psaní této práce nenaskytla, a proto se i čtenář musí spokojit pouze se simulovanými daty.

10.1.3 POROVNÁNÍ VÝPOČETNÍCH RYCHLOSTÍ

Výpočetní rychlosti přístupu čistě pomocí CPU a pomocí CPU s FPGA akcelerátorem jsou na následujících řádcích zhodnoceny. Výpočetní rychlost jako taková je vyhodnocena na

základě rychlosti výpočtu jednoho simulačního snímku. Vzhledem k tomu, že neuron QNU s tímto nastavením má 66 vah, je nutné využít maticový akcelerátor 128×128, jehož vývoj je popsán v 8.1.1 a jeho výpočetní rychlost je zhodnocena v 8.2.1. V případě CPU implementace je využito modifikace HONU do dlouhých vektorů, jak bylo vysvětleno v 5.1.1, zatímco v případě FPGA je přístup zachován čistě na maticové úrovni. Pro přehlednost jsou výsledky shrnuty v Tab. 18.

Tab. 18: Výsledky výpočetní rychlosti hardwarového akcelerátoru pro jeden simulační snímek ve vyvinutém detekčním softwaru v porovnání s výpočetní rychlostí čistě na dedikovaném CPU vývojové desky PYNQ-Z1.

Výpočetní časy	CPU	FPGA
Minimální	0.114 $[s]$	0.088 $[s]$
Maximální	0.126 $[s]$	0.103 $[s]$
Průměrný	0.118 $[s]$	0.095 $[s]$

V tabulce je dobře vidět, že výpočetní rychlost na čipu FPGA je sice vyšší, ale nikoliv o tolik, o kolik bylo očekáváno dle 8.2.1. A to i s přehlédnutím k faktu, že jeden snímek nese vždy devět datových záznamů z důvodu optimalizace grafického zobrazení, jak bylo vysvětleno v 9. Hlavním problémem je zde tedy zjevně overhead a logika aplikace jako takové, která je napsána výhradně v Pythonu. Ačkoliv bitstream nahraný na FPGA dokáže výpočet HONU jako takového opravdu zrychlit a výpočetní čas jednoho snímku se skutečně snížil, je zde stále velké omezení ze strany Pythonu, který se stará o chod celé aplikace.

Pokud se k výpočetnímu času přičte ještě čas potřebný ke grafickému vykreslení, jehož průměrná hodnota odpovídá 0.173 $[s]$, dostane se aplikace během externí simulace pouze na 3.42 FPS pro výpočty čistě na CPU a 3.73 FPS pro výpočty s maticovým akcelerátorem. V obou případech se ovšem jedná o celkem nízké hodnoty a detektor na desce PYNQ-Z1 by tak, v aktuálním stavu, zřejmě nebyl aplikovatelný pro celou řadu procesů či neměl vhodné komerční využití.

Lepší funkčnosti by bylo možné dosáhnout jak lepší optimalizací aplikace v Pythonu, tak přepsáním jejích důležitých částí do Cythonu, který by zřejmě přinesl největší zlepšení celkového výkonu. Toto přepsání by však v případě zde vyvinuté aplikace bylo časově velmi náročné a vzhledem k tomu, že nebylo předmětem této práce, nebyl tento směr dále rozvíjen.

Další možnou alternativou zlepšující výkon by bylo prakticky veškeré výpočty a datové manipulace provádět přímo na FPGA a Python využít pouze k přijímání dat a následné

vizualizaci výstupů detektoru. Tato alternativa by při správné implementaci měla být schopná podávat nejlepší výsledky, avšak zcela jistě by byl potřeba podstatně větší čip FPGA a mnohem rozsáhlejší hardwarový design. Návrh takto komplexního hardwarového designu ovšem zcela jistě leží za aktuálními schopnostmi a dovednostmi autora a patří spíše do komerční sféry, kde se nachází i opravdoví odborníci zabývající se touto problematikou.

10.2 DÁVKOVĚ ZPRACOVÁVANÁ REÁLNÁ DATA

Pro dávkově zpracovávaná data, jak již bylo dříve naznačeno, není aktuálně vyvinuto žádné grafické rozhraní a je nutné jej spustit speciálními příkazy, jak je uvedeno v Alg. 14. Tento postup nebyl do 9.1 zahrnut, vzhledem k tomu že se aktuálně jedná pouze o ověření konceptu a tento mód není napojen na celou řadu procesů hlavní aplikace a tudíž nemůže být považován za ekvivalentní s módy ostatními.

```
from detector import Detector
det = Detector(mode='batch', accelerate=Bool)
det.input = 'path://to/files/with/data/in/csv'
det.BatchStart()
```

Alg. 14: Import a spuštění detekčního SW v módu dávkového zpracování dat. Bool v parametru accelerate se nahradí buďto true nebo false v závislosti na tom, zda má byt provedena akcelerace na FPGA či nikoliv. Nastavení cesty k vstupním souborům se provede nastavením atributu input na cestu k souborům. Spuštění dávkové detekce se provede pomocí metody BatchStart(). Po dokončení všech výpočtů se zobrazí okno s výsledky.

Po spuštění provede program analýzu poskytnutých dat. Pokud není atribut *input* specifikován, jak je uvedeno výše, program předpokládá přítomnost složky *data* se soubory typu pickle. Znovu je nutné připomenout, že zde prezentovaná data pochází od průmyslového partnera a proto jsou jako taková zcela anonymizována. K tomu jsou zde také zobrazeny jen vybrané a schválené části. Jakmile bude analýza dokončená, automaticky se zobrazí všechny relevantní grafy.

Cílem zde prezentované metody je správně detekovat anomálie v testu a, ideálně, vytvořit algoritmus pro podporu vyhodnocování, zda výrobek testem prošel či nikoliv, za využití efektivní neuronové architektury a learning entropy dávkovou metodou na embedded zařízení. Tuto metodu by mělo být možno využít jak k vyhodnocení testu pro celý jeden výrobek, tak může významně pomoci dohledat počátek vzniku anomálie v daném testu.

Pro implementaci výpočtu neurálních vah byl použit dávkový algoritmus Levenberg-Marquardt a bylo nalezeno vhodné nastavení jak pro zvolený neuron, tak metodu ABSLE. Pro metodu ABSLE bylo využito dvou různých nastavení citlivosti. První nastavení je využíváno k vyhodnocení celého testu, zda proběhl úspěšně či nikoliv. Poskytuje tedy dobrou robustnost proti náhodným drobným anomáliím a je tak vhodné právě k celkovému vyhodnocení.

Druhé nastavení je již o poznání citlivější a je vhodné k dohledání počátku anomálie, která mohla vést až k neúspěšnosti testu. Použité nastavení zvoleného neuronu je pro přehlednost shrnuto v Tab. 19 a obě nastavení detekční metody ABSLE jsou shrnuty v Tab. 20. Za anomálii jsou považovány hodnoty ABSLE vyšší než 0.5, jak je zobrazeno na Obr. 39.

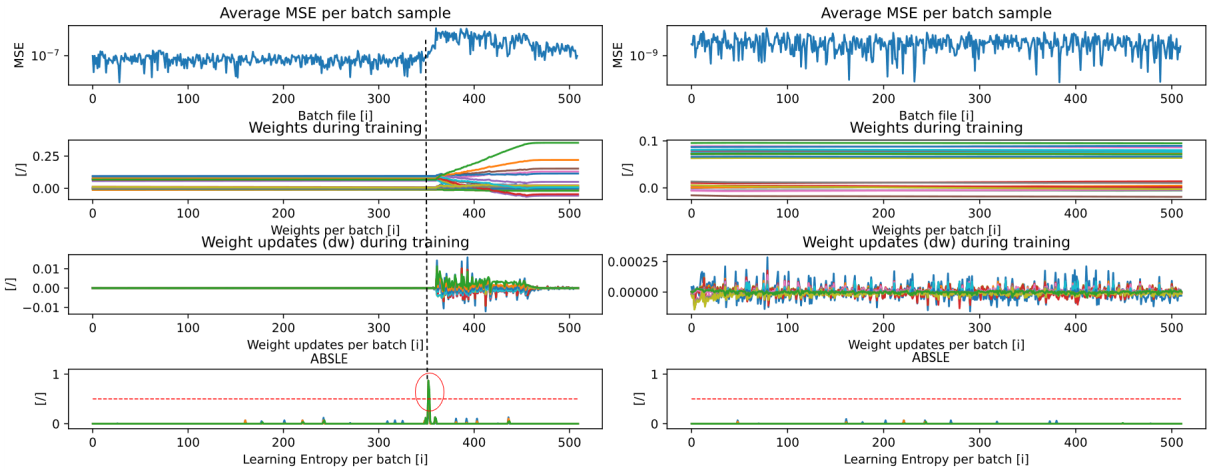
Tab. 19: Nastavení zvoleného typu neuronu pro dávkové zpracování vstupních dat od průmyslového partnera.

$Neuron$	y_{hist}	u_{hist}	μ
LNU	4	3	0.0001

Tab. 20: Nastavení detekčního algoritmu ABSLE o dvou úrovních citlivosti. Nízká citlivost je vhodná především k celkovému vyhodnocení testu zda byl úspěšný či neúspěšný zatím co vysoká citlivost je vhodná k odhalení možného počátku neúspěšnosti testu.

$Alphas_{low\ sensitivity}$	$Alphas_{high\ sensitivity}$	$OLEs$	LE_{window}
[20, 24, 32]	[5, 10, 14]	[1, 2, 3]	10

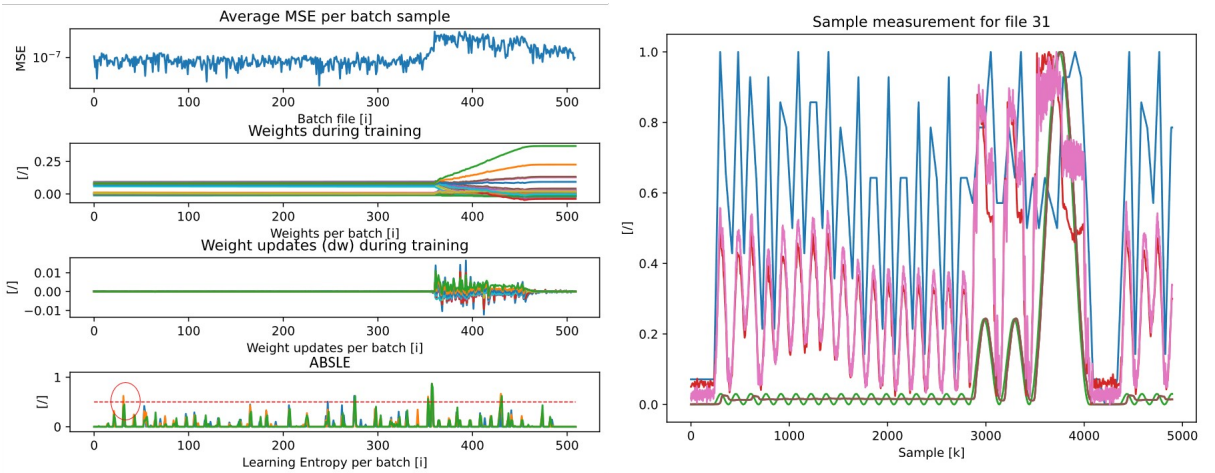
Na Obr. 39 jsou zobrazeny průběhy dávkového učení pro dvě testovací sady dat téhož výrobku, kde průběh učení vlevo odpovídá testu neúspěšnému a průběh učení vpravo odpovídá testu úspěšnému. Dále je vidět na průběhu vah, že u obou testovacích sad proběhla selekce závislých vstupů a málo či nezávislé veličiny byly potlačeny.



Obr. 39: Průběhy učení LNU na testovacích datech. Vlevo test, který skončil neúspěšně, divergence testu byla algoritmem LE detekována okamžitě s mírným časovým předstihem. Vpravo test, který skončil úspěšně.

Detekce neúspěšného testu pomocí ABSLE proběhla již v dávce 352, zatímco v datech na průběhu MSE je znatelně vidět až od dávky 361. Je zde tedy detekována se značným předstihem a pokud by byla tato detekce nasazena na živá data, bylo by možné test zastavit s uspokojivým předstihem. V případě úspěšného testu byly hodnoty ABSLE velice nízké s velkým odstupem od nastavené detekční hodnoty a lze tedy tato nastavení pro tyto data považovat za spolehlivé.

K detekci možného počátku anomálie je nutné zvýšit citlivost na hodnoty uvedené v Tab. 20. S vyšší citlivostí dostaneme průběh ABSLE zobrazený na Obr. 40. První dávka, která překročí nastavenou detekční hodnotu je dávka 31 a je patrné, že již na samotném začátku testu přestal výstup systému sledovat požadovanou hodnotu pro všechny menší pohyby.



Obr. 40: Detekce anomálie na počátku testu. Test následně vyšel negativně.

Jak se ukázalo, tuto metodu lze vhodně aplikovat i na reálná data s dobrými výsledky detekce. Tuto koncepční část zde vyvinutého SW je možné využít jak čistě na zařízení s CPU, tak lze využít i akceleraci zde vyvinutých bitstreamů pro FPGA. Porovnání jednotlivých výpočetních rychlostí je shrnuto v následující kapitole.

Hlavním omezením tohoto konceptu je aktuálně nutnost nakopírovat veškerá data do paměti embedded zařízení, jelikož není vyvinuta metoda přístupu do externího či cloudového úložiště. Ukládání dat bývá v každé firmě velmi individuální záležitost a jakékoliv zde navržené řešení by nemuselo být vyhovující. Proto nebyl vývoj této části zde navrženého softwaru hlouběji proveden a v případě jeho nutnosti jej lze relativně snadno dodělat díky modulárnímu konceptu celého programu.

Zde použité nastavení je uloženo ve složce Batch_Detection v souboru honu_script.py, (kap. 12). Po spuštění detektoru metodou popsanou na začátku této kapitoly dojde k automatickému natrénování HONU na vstupní data a k následnému zobrazení všech relevantních výsledků dávkové detekce.

10.2.1 POROVNÁNÍ VÝPOČETNÍCH RYCHLOSTÍ

Výpočetní rychlosti přístupu čistě pomocí CPU a pomocí CPU s FPGA akcelerátorem jsou na následujících řádcích zhodnoceny. Výpočetní rychlost jako taková je porovnávána jak pro jednu dávku, tak pro celkovou dobu zpracování všech dat, jelikož nezbytný overhead by nemusel být dostatečně dobře promítnut pouze do jedné dávky. Pro všechny testy na hardwaru bylo využito akcelerátoru 128×128, jehož vývoj je popsán v 8.1.1 a jeho výpočetní rychlost jako takového je zhodnocena v 8.2.1. Vzhledem k tomu, že celková délka dat dosahuje pro jeden soubor okolo pěti tisíc hodnot, musí zde vyvinutý software data rozložit na takzvané mini-batche o maximální velikosti odpovídající použitému akcelerátoru, což zde odpovídá délce o hodnotě 128. Pro přehlednost jsou výsledky opět shrnuty v Tab. 21.

Tab. 21: Výsledky výpočetní rychlosti hardwarového akcelerátoru pro reálnou aplikaci dávkového zpracování dat ve vyvinutém softwaru v porovnání s výpočetní rychlostí čistě na dedikovaném CPU vývojové desky PYNQ-Z1.

Průměrné hodnoty	Výpočetní čas CPU	Výpočetní čas FPGA
Jedné dávky	0.918 [s]	0.462 [s]
Všech dat	440.701 [s]	358.197 [s]

Dle očekávání byly výpočty s akcelerací na FPGA celkově rychlejší než tomu bylo v případě výpočtů na CPU a to i přesto, že výpočty na procesoru využívají optimalizace dle 5.1.1. Nicméně, z časů je patrné, že výpočetní akcelerátor jako takový je silně omezován výkonem kódu, který je napsaný v Pythonu. Výpočet jako takový by pro soubor s daty o délce 5000 vzorků měl na základě výsledků Tab. 10 trvat necelých 0,03 sekundy.

Jelikož ovšem musí Python připravovat a konsolidovat všechny mini-batche jedné dávky, je tento výpočetní čas značně prodloužen. Je také nutné přičíst nezbytný overhead softwaru jako takového, načež se teoretické hodnoty od reality opravdu velmi vzdálí. Aby se teoretické a reálné výpočetní časy k sobě alespoň přiblížily, bylo by nutné provést řadu optimalizací na úrovni jazyka Python. Velkým zlepšením napříč celou aplikací by bylo přepsat co největší část kódu do Cythonu a zvýšit tak celkový výkon aplikace při zachování snadného rozšíření funkcionalit pomocí Pythonu.

Pokud by se zde vyvinutý software na platformě PYNQ instaloval k testovacím standům, odkud zde zpracovávaná data pochází, bylo by možné obsluhovat opravdu velké množství strojů jedním zařízením. Jak již bylo vysvětleno v 7.2, testy obvykle trvají dlouhou dobu a výpočetní čas je v tomto ohledu téměř zanedbatelný. Výhodou této instalace by ovšem byla podstatně nižší spotřeba elektrické energie a nižší počáteční náklady než je tomu například v případě plnohodnotného počítače.

10.3 AUTOMATICKÉ UKONČENÍ TRÉNOVÁNÍ

Neurony je možné jednou natrénovat a dále už nepřetrénovat dle nastavené cílové chyby, avšak k správnému fungování je nutné dodržet několik podmínek. Obzvláště pokud se nejedná o data ze simulátoru či je vstup simulátoru často volně měněn nebo silně zašuměn. Mezi tyto nezbytné podmínky patří:

- Krátkodobější predikce (maximálně do 5 vzorků)
- Střední nebo vysoké vyhlazení dat (radius mezi 3 až 10 dle 9.2)
- Vstupní data musí mít přibližně konstantní rozsahy jak během trénování, tak během dalšího chodu programu jinak nebude normalizace dobře fungovat.

Pokud ovšem není možné zajistit všechny výše zmíněné podmínky, může nastat vyšší množství falešných detekcí a detektor jako takový by se mohl stát nespolehlivým. Velkou

výhodou zde ovšem zůstává, že tento výpočet bez následné aktualizace vah je podstatně rychlejší a pro některé systémy tak může být výhodný a žádaný.

10.4 ZHODNOCENÍ NÁROČNOSTI HW AKCELERACE NA FPGA

Hardwarové programování je velmi obtížná a neobyčejně zajímavá problematika, na které aktuálně stojí nejen prakticky celý moderní průmysl, ale i aktuální způsob našeho života. Velké společnosti jako Xilinx, Intel či Altera vynakládají velké množství svých finančních prostředků do vývoje softwarových nástrojů, které mají za cíl usnadnit celý proces hardwarového návrhu a jeho následné implementace a tím dále rozvíjet jeho široké spektrum aplikovatelnosti. I přes všechny tyto nové a sofistikované softwarové nástroje se ovšem stále jedná o velmi komplikovaný a dlouhý proces.

Pokud k této problematice přijde člověk jako autor této práce, tedy člověk zkušený v softwarovém programování a znalý programovacích jazyku jako C, C++ a Python s jen naprosto minimální znalostí v oblasti hardwarového programování, avšak s notnou dávkou nadšení pro tuto problematiku, je sice možné vyvinout hardwarový akcelerátor či jiný design, který bude opravdu fungovat, jeho výkon však nebude nikdy optimální.

Je tedy nutné poznamenat, že marketing a prezentace aktuálních vývojových nástrojů a obtížnost implementací návrhů na FPGA je podávána poněkud zavádějícím způsobem, který neodpovídá skutečným nezbytným znalostem pro úspěšnou implementaci vlastního designu a od nuly až po funkční knihovnu ve vysokém jazyce je opravdu dlouhá cesta. Nicméně pravdou zůstává, že oproti historickým postupům se o vývoj a celkové zjednodušení skutečně jedná a je dost dobře možné, že budoucí softwarové nástroje již těmto marketingovým hláškám budou i odpovídat. Prozatím v cestě bohužel stojí několik překážek, které je nutné překonat. Některé z těchto překážek jsou na následujících řádcích sepsány tak, jak je autor na vlastní kůži pocítil.

Za největší překážku je dle autora považován rozdílný způsob přístupů k programování jako takovému. Tyto rozdíly jsou patrné i dle definic hardwarových jazyků v kapitole 3 a jejich rozdíl je na následujících řádcích více porovnán vzhledem k softwarovému přístupu. Ačkoliv oba mají společný cíl, a to dosáhnout co nejrychlejšího spolehlivého řešení daného problému, oba k tomu musí využít jiný postup.

Běžný softwarový vývojář nemusí prakticky nikdy řešit pojem času uvnitř jeho kódu. Naopak hardwarový vývojář musí na implementaci časového signálu myslet již při samotném

začátku vývoje, jinak se zřejmě nepodaří dosáhnout optimálního implementačního řešení, anebo by cílové řešení ani nemuselo být schopné dosáhnout požadovaného časování daného čipu. Tento pojem času je pro softwarového programátora tedy zcela nový a nelehký na pochopení pro jeho správnou implementaci.

Dále je zde rozdíl v implementaci souběžného zpracování dat. Zatímco softwarový programátor navrhuje sekvenční části kódu, které mohou ale nemusejí být spuštěny paralelně (na vícero výpočetních jádrech) a obvykle řeší synchronizaci pomocí vhodné knihovny, tak hardwarové programování je z principu jeho fungování vždy paralelní a je synchronizováno právě časem, což opět vyžaduje jiný způsob uvažování při řešení dané problematiky a klade tak další nároky na programátory.

Softwarový programátor tedy musí své znalosti jazyka C a C++ upravit vhodně tak, aby byl schopný napsat hardwarově efektivní a přeložitelný kód, na který dále naváže specifické direktivy překladu jak byly popsány v 8.3. Jak již bylo vysvětleno, bez vhodného užití těchto direktiv je zcela nemožné dosáhnout optimálního překladu požadovaného návrhu. Ačkoliv se autor původně domníval, že se bude moci využít k návrhu hardwaru jazyk Python a jeho knihovny myHDL, bohužel rychle zjistil, že se jedná spíše o hudbu budoucnosti. Aktuálně je totiž nutné psát i v Pythonu zcela dle syntaxe VHDL a k tomu zde není možnost využití direktiv překladu jako tomu je v případě vývoje v nástroji Vitis. Tento způsob vývoje je tedy velice komplikovaný a pro nováčka v této oblasti prakticky nereálný.

Za další velkou překážku autor považuje skutečnost, že existuje jen velmi malé množství open-source IP bloků, které je možné použít přímo pro daný projekt nebo se pomocí nich učit způsob hardwarového programování obdobně, jako je tomu v případě vývoje a učení se softwarovému kódu. Toto je však bohužel dáno jak komplikovaností vývoje hardwarového návrhu, který firmy obvykle nechtějí dávat na internet zdarma, tak skutečností, že tyto návrhy jsou a často musí být velice specifické pro daný čip a jeho zdroje čímž je jeho znovupoužitelnost minimální.

Při porovnání se softwarovým kódem, kterému takřka nezáleží na tom, na jakém hardwaru běží, je toto zcela zásadní rozdíl a zároveň i překážka pro větší rozšíření open-source komunity. Je sice pravda, že vývojové desky typu PYNQ byly vyvinuty právě s ohledem na rozšíření vývojářské komunity a zvýšení tak počtu open-source IP bloků díky jejich komunitnímu webu, kde vývojáři mohou sdílet své projekty. Prozatím se ovšem jedná o velmi malou a mladou komunitu a valná většina zde zveřejněných řešení neposkytuje své

zdrojové kódy a tudíž není možné dané uveřejněné řešení upravit či jej nelze využít k téměř jakékoliv formě výuky.

Rozvoji v této oblasti by podle autora mohl pomoci vznik výukových webů obdobně, jako je tomu pro softwarové jazyky. Avšak opět se jedná o nákladný a dlouhý projekt, který je s aktuálním rozpoložením trhu prakticky nereálný pro jeho nízkou návratovou hodnotu.

Za poslední velkou zde uvedenou překážku považuje autor celkovou časovou náročnost implementace řešení na čip FPGA. Oproti vývoji softwaru se jedná opravdu o extrémní rozdíl. U hardwarového designu není obvykle komplikovanost v řádcích kódu, ale je zde celkem běžné, že doladění překladu k maximálnímu využití zdrojů čipu a vyvážení priorit mezi jednotlivými IP bloky a jejich případnými přerušeními může být klidně i den, dva práce, kde na konci bude napsáno jen 50 řádků kódu. U běžného softwarového vývoje je na konci pracovního dne obvykle podstatně větší množství kódu, jehož autor ovšem obvykle nemusí řešit priority mezi jeho jednotlivými částmi, jelikož se o to postará například compiler a nějak to na konci bude fungovat.

Dalším velkým žroutem vývojového času je ladění softwaru. Zatímco softwarový programátoři mohou často využívat k ladění breakpointy a debugger módy, které jim poskytne jejich vývojové prostředí IDE, a celý proces tak značně urychlí, v případě hardwarového programování je sice možné využít sledování časových signálů a jednotlivých logických úrovní v simulaci prováděné pomocí Vivado HLS, avšak vzhledem k tomu, že se jedná pouze o simulaci, musí opravdový hardwarový programátor často použít i osciloskop, logický analyzátor, anebo někdy i multimetr.

Autor si naštěstí pro účely této práce vystačil se sledováním časových signálů v simulaci implementačního softwaru Vivado, které využil pro optimalizaci přesunu dat do paměti BRAM. I tak se ovšem jednalo o necelý týden práce s jen minimálním posunem kupředu, což samozřejmě bylo dáno i autorovou původní neznalostí problematiky a nutností intenzivního zjišťování potřebných informací.

Celkové zhodnocení náročnosti implementace na základě zde odvedené práce je, že díky moderním nástrojům je možné, aby i strojní inženýr zručný v softwarovém programování dokázal navrhnout alespoň nějaký hardwarový design. Tento design nebude ovšem nikdy optimální jako kdyby byl navržen odborníkem. Návrh hardwarového designu je i přes značnou automatizaci na prakticky všech úrovních časově velmi náročný a s největší pravděpodobností by nebyl finančně výhodný pro celou řadu potencionálních aplikací. Vývoj

samotného hardwarového návrhu by totiž nakonec mohl potencionální produkt prodražit na tolik, že by jeho možné výhody, jako jsou rychlost, efektivita a spotřeba byly zcela zastíněny jeho cenou.

Když se ovšem pozornost zaměří mimo komerční sektor třeba na školství, zdá se, že vývojové desky typu PYNQ by mohly být velmi užitečné během výuky na vysokých školách a to nejen na školách elektrotechnických. Autor je přesvědčen o tom, že jeden semestr s odborně vedeným projektem na této desce by poskytl studentům dostatečně dobrou představu a nutný základ pro vlastní vývoj základního hardwarového návrhu a dále jim tak rozšířil představu o tom, jak hardware, který řídí jejich stroje a systémy vlastně funguje. Díky těmto znalostem by při jejich dalším profesním životě dokázali vhodně přihlédnout k hardwaru jako takovému a navrhovat tak lepší zařízení než doposud.

11 ZÁVĚR A BUDOUCÍ APLIKACE

Cílem této práce bylo provést studii dostupných nízko nákladových HW řešení s open-source SW nástroji pro výkonné výpočty se strojovým učením v reálném čase a následnou vlastní implementací na problematiku detekce anomálií s využitím jedné ze zvolených platforem.

Zvolenou platformou pro účely této práce byla vývojová deska PYNQ-Z1 s čipem FPGA. Tato deska poskytuje široké možnosti pro potenciální budoucí aplikace a autora nalákal i marketing, který prezentoval takové usnadnění HW programování, že by jej měl bez problémů zvládnou i běžný softwarový programátor. Toto se do jisté míry i potvrdilo avšak komplikovanost celého vývoje je opravdu velká i přes tuto dnes již rozsáhlou automatizaci. Je však nutné podotknout, že bez této automatizace by nebylo zcela jistě vůbec možné, aby člověk bez dostatečné znalosti problematiky hardwarového programování dokázal alespoň nějaký design sám vytvořit a v tomto ohledu tedy i marketing do jisté míry svoje sliby splnil.

Pro zvolenou platformu s čipem FPGA byl navržen a naprogramován hardwarový akcelerátor maticových výpočtů. Vzhledem k povaze hardwaru byly vyvinuty akcelerátory o čtyřech rozměrech, a to 128x128, 64x64, 32x32 a 16x16, kde od každého z nich bylo očekáváno zrychlení výpočetní rychlosti, poněvadž byl vždy kladen důraz na maximální využití bloků DSP neboli „násobiček“. Teoretická rychlost se na základě simulací skutečně vždy podstatně zlepšila a pro akcelerátor 16x16 dosahovala dokonce $2.2\ \mu s$, avšak autor

narazil na hardwarové omezení, které se zde bohužel nepovedlo vyřešit. Toto omezení spočívá v přenosu dat mezi PS a PL a zásadně snižuje maximální reálnou výpočetní rychlost, která pro akcelerátor 16x16 odpovídá výpočetní rychlosti $318\ \mu s$. Podrobnější prozkoumání tohoto problému bylo zdokumentováno v 8.2. Tyto akcelerátory pak byly integrovány do detekčního softwaru za účelem zrychlení vhodných maticových výpočtů.

Dále byly v této práci vyvinuty dva softwary v jazyce Python s grafickým rozhraním v Jupyter notebooku. Prvním vyvinutým softwarem je simulátor dynamických systémů. Tento simulátor dovoluje budit požadovaný systém třemi různými typy signálu kde každý z nich je možné téměř libovolně nastavit přímo v grafickém rozhraní. Kromě toho je tento simulátor, nad rámec zadání této práce, vybaven i možností PID regulace simulovaného systému a dále se tak rozšiřují jeho možnosti. Simulátor jako takový může běžet v lokálním módu, kde jiný proces na stejném zařízení získává jeho data, anebo může být spuštěn v takzvaném externím módu a data odesílat pomocí UART či SPI. Ačkoliv bylo externí odesílání dat testováno pouze v kombinaci Raspberry a PYNQ, software je možné využít i pro jiné embedded systémy po náležité změně konfiguračního souboru. Simulátor jako takový je popsán v kapitole 7.1.

Druhým softwarem je detektor anomálií. Uživatelské prostředí tohoto detektoru bylo navrženo tak, aby uživatel mohl co nejsnáze měnit všechna důležitá nastavení. Pro neuron lze tak nastavit například jeho typ a počet jeho historických vstupů nebo zvolit jeho adaptační metodu. Pro detekci lze přímo v uživatelském prostředí nastavovat citlivost metody LE nebo měnit nastavení metody vyhodnocování chyby z dlouhodobého hlediska. Nad rámec zadání této práce byla ještě vyvinuta metodika dávkového vyhodnocování chyb v případě potřeby zpracování většího množství datových souborů z testů daného výrobku. Tato dávková metoda není vybavena grafickým rozhraním. Vzhledem k tomu, že detektor musí komunikovat se simulátorem, tak je také dle očekávání vybaven komunikačními rozhraními UART a SPI.

Pomocí zde vyvinutého softwaru lze dosáhnout uspokojivých výsledků detekce, jak je uvedeno v 10.1 a 10.2, nicméně je potřeba, aby uživatel aplikace správně naladil všechny nezbytné parametry pro detekci neobvyklých stavů a postupoval při tom tak, jak bylo naznačeno v kapitolách 6, 9 a 10. Ačkoliv ve finále hardwarová akcelerace maticových výpočtů nepřinesla zásadní rozdíl ve výkonu aplikace, stále vždy zvyšovala její celkový výkon, ať již pro online data vyhodnocována v 10.1.3 nebo pro dávkově zpracovávaná data diskutovaná v 10.2.1. Při zapnuté hardwarové akceleraci je nutné pamatovat na vhodný počet historických hodnot. Pokud by jejich počty na úrovni vah přesáhly celkový počet 128, pak

není možné hardwarovou akceleraci využít a software musí jít cestou výpočtů na CPU, které by byly zřejmě dosti pomalé.

Všechny části zde vyvinutého softwaru v jazyce Python jsou modulární a navrženy tak, aby se dle potřeby daly modifikovat. Je tedy možné aplikaci do budoucna rozšiřovat a to jak z hlediska nových algoritmů učení a nových metod vyhodnocování chyb detekce, tak z hlediska hardwarové akcelerace. Relativně snadno tedy může být do hlavního softwaru implementována například celá metoda hardwarově akcelerované learning entropy, pokud se jí samozřejmě povede také vhodně hardwarově naprogramovat.

V navazujících pracích by bylo zajímavé zjistit, pro jakou konfiguraci metody AISLE by bylo možné využít plně čip FPGA právě na desce PYNQ-Z1 a jaký přínos by ve finále tato implementace představovala oproti zde vyvinutému maticovému akcelerátoru. Vzhledem k dostupným hardwarovým zdrojům na tomto čipu je velice nepravděpodobné, že by se celé jádro detekčního softwaru dokázalo na tento čip vejít, avšak v budoucnu budou s největší pravděpodobností cenově dostupné i podstatně větší čipy, na které by se již mohlo povést vtěsnat všechny potřebné části detekčního softwaru a opravdu dosáhnout i megaherzových výpočetních rychlostí. Vývoj hardwarového designu se všemi nezbytnými částmi jako jsou neuron, adaptační algoritmus a algoritmus learning entropy na čipu FPGA s propojením na aplikaci napsanou ve vysokoúrovňovém jazyce by sice byla velká výzva, avšak dle názoru autora by toto mohl být velmi vhodný projekt pro univerzitní prostředí, na jehož konci by mohl vzniknout skutečný hardwarový detektor s velmi vysokou elektrickou a výpočetní efektivitou.

Zdroje

[1] GUPTA, Madan M., Liang JIN a Noriyasu HOMMA. *Static and dynamic neural networks: from fundamentals to advanced theory*. New York: Wiley, 2003. ISBN 0-471-21948-7.

[2] BUKOVSKY, Ivo a Noriyasu HOMMA. An Approach to Stable Gradient-Descent Adaptation of Higher Order Neural Units. *IEEE Transactions on Neural Networks and Learning Systems* [online]. 2016, 1–13. ISSN 2162-237X, 2162-2388. Dostupné z: doi:10.1109/TNNLS.2016.2572310

[3] BUDÍK, Ondřej, Daniel MALACHOV a Ivo BUKOVSKY. *Dávkový Algoritmus Learning Entropy v Softwaru Analýzy Testovacích Dat* [online]. B.m.: CTU in Prague, nedatováno. 28th WORKSHOP OF APPLIED MECHANICS BOOK OF PAPERS. ISBN 78-80-01-06791-8. Dostupné z: https://aleph.cvut.cz/

[4] BUKOVSKY, Ivo. Learning Entropy: Multiscale Measure for Incremental Learning. *Entropy* [online]. 2013, **15**(10), 4159–4187. Dostupné z: doi:10.3390/e15104159

[5] Coral. *Coral* [online]. [vid. 2020-06-24]. Dostupné z: https://coral.ai/

[6] PYNQ - Python productivity for Zynq. *PYNQ - Python productivity for Zynq* [online]. [vid. 2020-09-02]. Dostupné z: http://www.pynq.io/home.html

[7] Tensor Processing Unit (TPU). *Semiconductor Engineering* [online]. [vid. 2020-06-24]. Dostupné z: https://semiengineering.com/knowledge_centers/integrated-circuit/ic-types/processors/tensor-processing-unit-tpu/

[8] TensorFlow models on the Edge TPU. *Coral* [online]. [vid. 2020-10-15]. Dostupné z: https://coral.ai/docs/edgetpu/models-intro/#compatibility-overview

[9] LTD, Arm. Microprocessor Cores and Technology – Arm. *Arm | The Architecture for the Digital World* [online]. [vid. 2020-05-05]. Dostupné z: https://www.arm.com/products/silicon-ip-cpu

[10] ŠŤASTNÝ, Jakub. *FPGA prakticky : realizace číslicových systémů pro programovatelná hradlová pole*. 1. vyd. B.m.: BEN - technická literatura, 2010. ISBN 978-80-7300-261-9.

[11] OKAFOR, Kennedy, Genevra CHINWE a Ogungbenro AKINYELE. *Hardware description language (HDL): An efficient approach to device independent designs for VLSI market segments* [online]. 2011. ISBN 978-1-4673-0758-1. Dostupné z: doi:10.1109/ICASTech.2011.6145181

[12] COELHO, David R. *The VHDL Handbook* [online]. B.m.: Springer US, 1989 [vid. 2021-02-08]. ISBN 978-0-7923-9031-2. Dostupné z: doi:10.1007/978-1-4613-1633-6

[13] GOLSON, Steve. *Oral History of Philip Raymond "Phil" Moorby* [online]. 22. duben 2013 [vid. 2021-02-09]. Dostupné

z: http://archive.computerhistory.org/resources/access/text/2013/11/102746653-05-01-acc.pdf

[14] HW Acceleration. *AI@Edge Community* [online]. [vid. 2020-09-02]. Dostupné z: https://microsoft.github.io/ai-at-edge/docs/hw_acceleration/

[15] x86 computer processors Intel/AMD market share 2012-2019. *Statista* [online]. [vid. 2020-04-27]. Dostupné z: https://www.statista.com/statistics/735904/worldwide-x86-intel-amd-market-share/

[16] *Architektura AMD64* [online]. [vid. 2020-05-04]. Dostupné z: http://noel.feld.cvut.cz/vyu/scs/prezentace2004/Amd64/

[17] *PassMark - CPU Benchmarks - CPU Mega Page - Detailed List of Benchmarked CPUs* [online]. [vid. 2020-05-04]. Dostupné z: https://www.cpubenchmark.net/CPU_mega_page.html

[18] *ARM processor - All industrial manufacturers* [online]. [vid. 2020-05-04]. Dostupné z: https://www.directindustry.com/industrial-manufacturer/arm-processor-77989.html

[19] *Sitara Processors | Industrial Processors| Overview | Processors | TI.com* [online]. [vid. 2020-05-05]. Dostupné z: http://www.ti.com/processors/sitara-arm/overview.html

[20] SoCs, MPSoCs and RFSoCs. *Xilinx* [online]. [vid. 2020-05-05]. Dostupné z: https://www.xilinx.com/products/silicon-devices/soc.html

[21] *OSRTOS* [online]. [vid. 2020-05-05]. Dostupné z: https://www.osrtos.com/

[22] *List of computer hardware manufacturers* [online]. 2020 [vid. 2020-05-06]. Dostupné z: https://en.wikipedia.org/w/index.php?title=List_of_computer_hardware_manufacturers&oldid=954778989

[23] GPU Accelerated Computing with C and C++. *NVIDIA Developer* [online]. 25. listopad 2013 [vid. 2020-05-06]. Dostupné z: https://developer.nvidia.com/how-to-cuda-c-cpp

[24] *AMD Accelerated Parallel Processing OpenCL Programming Guide* [online]. [vid. 2020-05-06]. Dostupné z: http://developer.amd.com/wordpress/media/2013/07/AMD_Accelerated_Parallel_Processing_OpenCL_Programming_Guide-rev-2.7.pdf

[25] ROSS, James A., David A. RICHIE, Song J. PARK, Dale R. SHIRES a Lori L. POLLOCK. A case study of OpenCL on an Android mobile GPU. In: *2014 IEEE High Performance Extreme Computing Conference (HPEC)*: *2014 IEEE High Performance Extreme Computing Conference (HPEC)* [online]. 2014, s. 1–6. Dostupné z: doi:10.1109/HPEC.2014.7040987

[26] List of FPGA Companies. *HardwareBee* [online]. 8. duben 2018 [vid. 2020-06-23]. Dostupné z: https://hardwarebee.com/list-fpga-companies/

[27] Silicon devices. *Xilinx* [online]. [vid. 2020-06-23]. Dostupné z: https://www.xilinx.com/products/silicon-devices.html

[28] *FPGA Programming for the Masses - ACM Queue* [online]. [vid. 2020-06-23]. Dostupné z: https://queue.acm.org/detail.cfm?id=2443836

[29] *Advances in Computers*. B.m.: Academic Press, 1993. ISBN 978-0-08-056669-6.

[30] Edge TPU inferencing overview. *Coral* [online]. [vid. 2020-06-24]. Dostupné z: https://coral.ai/docs/edgetpu/inference/

[31] Difference Between Microprocessor and Microcontroller. *Electronics Hub* [online]. 29. květen 2015 [vid. 2020-09-01]. Dostupné z: https://www.electronicshub.org/difference-between-microprocessor-and-microcontroller/

[32] Approximate Computing for On-Chip AI Acceleration: IBM Research at VLSI. *IBM Research Blog* [online]. 27. červen 2018 [vid. 2020-09-01]. Dostupné z: https://www.ibm.com/blogs/research/2018/06/approximate-computing-ai-acceleration/

[33] STM32F103VC. *STMicroelectronics* [online]. [vid. 2020-09-02]. Dostupné z: https://www.st.com/en/microcontrollers-microprocessors/stm32f103vc.html

[34] Cyclone® IV FPGAs Features - INTEL® FPGA. *Intel* [online]. [vid. 2020-09-02]. Dostupné z: https://www.intel.com/content/www/us/en/products/programmable/fpga/cyclone-iv/features.html

[35] *Arty A7 Reference Manual [Reference.Digilentinc]* [online]. [vid. 2020-09-02]. Dostupné z: https://reference.digilentinc.com/reference/programmable-logic/arty-a7/reference-manual

[36] Welcome to FPGA MicroPython (FμPy). *fupy.github.io* [online]. [vid. 2020-09-02]. Dostupné z: https://fupy.github.io/

[37] *Teach, Learn, and Make with Raspberry Pi – Raspberry Pi* [online]. [vid. 2020-10-01]. Dostupné z: https://www.raspberrypi.org/

[38] Intel® Movidius™ Vision Processing Units (VPUs). *Intel* [online]. [vid. 2020-10-01]. Dostupné z: https://www.intel.com/content/www/us/en/products/processors/movidius-vpu.html

[39] Jetson Nano Developer Kit. *NVIDIA Developer* [online]. 6. březen 2019 [vid. 2020-10-05]. Dostupné z: https://developer.nvidia.com/embedded/jetson-nano-developer-kit

[40] RESEARCHER, Javier BonillaResearcher at CIEMAT-PSAPhD in Computer Science / Solar Thermal Energy. Coral USB Accelerator, TensorFlow Lite C++ API & Raspberry Pi for Edge TPU object detection. *Mechatronics Blog* [online]. 24. červen 2019 [vid. 2020-06-24]. Dostupné z: https://mechatronicsblog.com/coral-usb-accelerator-tensorflow-lite-c-api-raspberry-pi-for-edge-tpu-object-detection/

[41] *Google Coral Edge TPU Incompatible with Raspbian Buster - Raspberry Pi Forums* [online]. [vid. 2020-06-24]. Dostupné z: https://www.raspberrypi.org/forums/viewtopic.php?t=244021

[42] *Hardware Requirements (Running Linux)* [online]. [vid. 2020-10-05]. Dostupné z: https://docstore.mik.ua/orelly/linux/run/ch01_09.htm

[43] *3.4. Meeting Minimum Hardware Requirements* [online]. [vid. 2020-10-05]. Dostupné z: https://www.debian.org/releases/jessie/mips/ch03s04.html.en

[44] *Project Jupyter* [online]. [vid. 2020-10-07]. Dostupné z: https://www.jupyter.org

[45] *Home :: OpenCores* [online]. [vid. 2020-10-07]. Dostupné z: https://opencores.org/

[46] *LibreCores · Free and Open Source Digital Hardware* [online]. [vid. 2020-10-07]. Dostupné z: https://www.librecores.org/

[47] *The Free and Open Source Silicon Foundation* [online]. [vid. 2020-10-07]. Dostupné z: https://fossi-foundation.org/

[48] An in-depth look at Google's first Tensor Processing Unit (TPU) | Google Cloud Big Data and Machine Learning Blog. *Google Cloud* [online]. [vid. 2018-06-19]. Dostupné z: https://cloud.google.com/blog/big-data/2017/05/an-in-depth-look-at-googles-first-tensor-processing-unit-tpu

[49] *JAX Quickstart — JAX documentation* [online]. [vid. 2020-10-16]. Dostupné z: https://jax.readthedocs.io/en/latest/notebooks/quickstart.html#Multiplying-Matrices

[50] Vitis High-Level Synthesis. *Xilinx* [online]. [vid. 2021-02-08]. Dostupné z: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[51] RUSHTON, Andrew. *VHDL for Logic Synthesis*. B.m.: John Wiley & Sons, 2011. ISBN 978-0-470-97797-2.

[52] KIVELÄ, Ville a Jukka LAHTI. *DESIGN, IMPLEMENTATION AND VERIFICATION OF A DIGITAL PREDISTORTER OVERDRIVE PROTECTION MODULE* [online]. Finland, 2016 [vid. 2021-02-10]. Faculty of Information Technology and Electrical Engineering, University of Oulu, Oulu, Finland. Dostupné z: http://jultika.oulu.fi/files/nbnfioulu-201606092483.pdf

[53] BUDÍK, Ondřej. *Supervizorované algoritmy strojového učení pro analýzu průmyslových dat* [online]. B.m., 2019 [vid. 2020-10-06]. České vysoké učení technické v Praze. Vypočetní a informační centrum. Dostupné z: https://dspace.cvut.cz/handle/10467/83624

[54] SCHRAUDOLPH, Nic a Fred CUMMINS. Linear Neural Networks. *Introduction to Neural Networks* [online]. 2006 [vid. 2019-03-29]. Dostupné z: https://cnl.salk.edu/~schraudo/teach/NNcourse/linear2.html

[55] MANDIC, Danilo P. A Generalized Normalized Gradient Descent Algorithm. *IEEE SIGNAL PROCESSING LETTERS*. 2004, **11**(2), 4.

[56] MORÉ, Jorge J. The Levenberg-Marquardt algorithm: Implementation and theory. In: G. A. WATSON, ed. *Numerical Analysis* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 1978 [vid. 2019-05-14], s. 105–116. ISBN 978-3-540-08538-6. Dostupné z: doi:10.1007/BFb0067700

[57] *Hessian matrix of scalar function - MATLAB hessian* [online]. [vid. 2019-05-14]. Dostupné z: https://www.mathworks.com/help/symbolic/hessian.html#buiej1q-2

[58] LEVENBERG, KENNETH. A METHOD FOR THE SOLUTION OF CERTAIN NON-LINEAR PROBLEMS IN LEAST SQUARES. *Quarterly of Applied Mathematics*. 1944, **2**(2), 164–168. ISSN 0033-569X.

[59] CEJNEK, Matouš a Ivo BUKOVSKY. *Novelty detection via linear adaptive filters* [online]. Prague, Czech Republic, 2020 [vid. 2021-04-15]. DOCTORAL THESIS. Czech Technical University in Prague. Dostupné z: https://dspace.cvut.cz/bitstream/handle/10467/93682/F2-D-2020-Cejnek-Matous-thesis_final.pdf?sequence=-1&isAllowed=y

[60] KINGMA, Diederik P. a Jimmy BA. Adam: A Method for Stochastic Optimization [online]. 2014 [vid. 2019-06-11]. Dostupné z: https://arxiv.org/abs/1412.6980v9

[61] *scipy.integrate.ode — SciPy v1.6.2 Reference Guide* [online]. [vid. 2021-04-20]. Dostupné z: https://docs.scipy.org/doc/scipy/reference/generated/scipy.integrate.ode.html

[62] BAGNI, Daniele, A Di FRESCO, J NOGUERA a F M VALLINA. A Zynq Accelerator for Floating Point Matrix Multiplication Designed with Vivado HLS. 2016, 34.

[63] *HLS Pragmas* [online]. [vid. 2021-04-28]. Dostupné z: https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html

12 PŘÍLOHA

Seznam přílohy:

1. Instrukce k instalaci prostředí pro spuštění mé aplikace na PC a PYNQ-Z1
2. Instrukce k zobrazení vzorových aplikací
3. Zdrojový kód
4. CD s prací v elektronické podobě, funkcemi a obrázky

12.1 INSTALACE PROSTŘEDÍ

Postup instalace:

1. Instalace Pythonu 3.8 (testováno na 3.8.5) pomocí služby Anaconda z https://www.anaconda.com/
   - Nainstalovat instalační balíček pro vhodný OS dle jeho instrukcí.
   - aktualizace Python nástrojů příkazem v Anaconda Prompt:
   - **conda update --all**
2. Instalace nezbytných balíčků příkazem v Anaconda prompt v následujícím pořadí:
   - **conda install -c anaconda scipy**
3. Instalace vývojového prostředí Vivado 2020.2 a Vitis HLS 2020.2 ze stránek společnosti Xilinx https://www.xilinx.com/support/download.html
   - Stáhnout vhodný Vivado Design suit dle OS.
   - Během instalace zvolit WebPack, jelikož se jedná o freeware.
   - Přihlásit se ke svému Xilinx účtu či se registrovat a odsouhlasit licenční podmínky.
   - Dokončit instalaci.
4. Příprava systému pro PYNQ-Z1 ze stránek výrobce http://www.pynq.io/board.html
   - Stáhnout obraz pro desku PYNQ-Z1.
   - Pomocí balenaEtcher nahrát obraz na SD kartu (min. 8GB).

- Nastavit **JP4** na pozici SD, vložit SD kartu, nastavit **JP5** na pozici USB, připojit USB napájení, připojit Ethernet, zapnout desku. Jakmile bude systém plně připraven, rozsvítí se dioda **DONE**.
- Připojit se k desce na PC zadáním adresy **pynq:9090/** do internetového prohlížeče.
- Zadat výchozí uživatelský jméno **xilinx** a heslo **xilinx** a přihlásit se do Jupyteru.

12.2 INSTRUKCE K ZOBRAZENÍ VZOROVÝCH APLIKACÍ

Celá struktura programu je náležitě rozřazena do vhodně pojmenovaných složek. Pokud by byl požadavek na simulaci jiného dynamického systému, je nutné upravit funkci „system“ uvnitř modulu simulator dle vhodné ODE syntaxe pro knihovnu Scipy. Vstupní buzení je možné snadno měnit pomocí grafického rozhraní či přímo uvnitř konfiguračního souboru. Pokud bude požadavek na jiná batchově zpracovávaná data, je nezbytné tato data nahradit uvnitř složky batch_data s obdobnou strukturou.

K zobrazení zde užitých grafických reprezentací slouží především metody programu v detector.py. Výstupní zobrazení je vždy automaticky nastaveno pomocí vložení požadovaného konfiguračního souboru do složky s programem a jeho následným spuštěním. Připravené konfigurační soubory jsou k dispozici ve složce sample_configs s vhodným komentářem v docstringu celého souboru.

12.3 ZDROJOVÉ KÓDY

V mé diplomové práci využívám 4 vyvinuté knihovny s vlastními skripty.

Seznam skriptů:

1. Hlavní knihovna
    - Logika programu se nachází v souboru detector.py. V tomto souboru je propojen jak simulátor, tak všechny detekční metody a neuronové sítě typu HONU. Pokud bude tento soubor spuštěn jako skript, bude spuštěna simulace dynamického systému, který je definován v simulator.py dle nastavení uvedených v config.py. **Různé, zde představené implementace, lze simulovat nahrazováním konfiguračního souboru ze složky**

sample_configs. Pokud bude skript spuštěn na desce PYNQ-Z1 je nutné změnit inicializační mód z *„local“* na *„fpga“*.

- Grafické rozhraní v notebooku Jupyter lze spustit ze souboru Detector GUI.ipynb po spuštění zde připraveného kódu proběhne celá inicializace plně automaticky a uživatelsky přívětivě.

2. Přídavné moduly
   - nn_accel – Obsahuje všechny nezbytné součásti k objektové implementaci metodik neuronů HONU. Hlavní skript modulu, honu.py obsahuje rodičovskou třídu pro všechny ostatní deriváty metody HONU. Adaptace vah probíhá pomocí třídy HONUOptimizer jejíž metody jsou vhodně měněny dle nastavení detektoru.
   - system_simulator – Obsahuje třídu a nezbytné funkce k simulaci dynamického systému dle požadovaných parametrů. Třída simulátoru jako taková obsahuje i nastavení grafického rozložení pro vizualizaci v Jupyter notebooku, je-li v tomto režimu spuštěn.
   - Bitstreams – Obsahuje bitstreamy a jejich správce nahrávané na čip FPGA, které obsahují hardwarovou akceleraci maticového násobiče. Tyto bistreamy lze nahrát pouze na čip FPGA osazený na desce PYNZ-Z1. Pokud hlavní skript běží na desce PYNQ-Z1 a během inicializace bylo zvoleno *„fpga“*, bude automaticky zvoleno, zda může být využita výpočetní akcelerace a také dojde automatickému omezení velikosti vstupních dat.
   - Batch_detection – Obsahuje metodiky a scripty potřebné k provedení dávkovému zpracování dat detekční metodikou ABSLE. Samostatně je tento modul nespustitelný a vyžaduje alespoň základní informace z hlavního detekčního modulu.
3. Dodatečné skripty
   - Jupyter notebooky uvnitř složek každého z bitstreamů slouží k ověření výpočetní rychlosti jednotlivých implementací. Vzhledem k tomu, že numpy využívá funkce cachování výsledků, nebylo možné použít magické

funkce typu %timeit a na místo toho jsou zde tyto měřící funkce implementovány vlastním způsobem.

- Jupyter notebooky končící na GUI, které jsou ve složkách detektoru a simulátoru obsahují připravené inicializační skripty k jednotlivým módům a stačí je tedy pouze spustit.
- Složka sample_configs obsahuje různé konfigurační nastavení pro zde použité implementační metody. Jejich nakopírováním do kořenového adresáře a spuštěním programu v detector.py dojde k demonstraci zde prezentovaných výsledků.

Okomentování kódu

- Všechny hlavní skripty jsou důkladně okomentovány, aby bylo snadné pochopit všechny jejich kroky. Každá třída a funkce jakož to i modul mají svůj vlastní docstring a je tedy možné využívat i nápovědy v prostředích IDE.
- Jupyter notebooky jsou okomentovány pouze do té míry, aby bylo jasné co se v dané části ukázky děje a obvykle zde již není detailně vysvětlován kód jako takový.