VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ
DEPARTMENT OF COMPUTER SYSTEMS

INFERENCE NEURONOVÉ SÍTĚ NA ZAŘÍZENÍ ZYNQ

NEURAL NETWORK INFERENCE ON THE ZYNQ

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

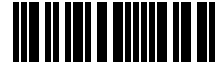
AUTOR PRÁCE — **FILIP MASÁR**
AUTHOR

VEDOUCÍ PRÁCE — **Ing. VOJTĚCH MRÁZEK, Ph.D.**
SUPERVISOR

BRNO 2024

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce



156344

Ústav: Ústav počítačových systémů (UPSY)
Student: **Masár Filip**
Program: Informační technologie
Název: **Inference neuronové sítě na zařízení ZYNQ**
Kategorie: Návrh číslicových systémů
Akademický rok: 2023/24

Zadání:

1. Seznamte se s možnostmi akcelerace inference neuronové sítě na zařízeních FPGA.
2. Seznamte se s tématem odolnosti proti poruchám.
3. Zpracujte studii na výše uvedená témata.
4. Vyberte vhodný existující akcelerátor inference a transformujte jej na čip FPGA.
5. Navrhněte a implementujte rozšíření akcelerátoru pro simulaci jedné vybrané chyby čipu.
6. Vyhodnoťte vlastnosti a chování implementovaného akcelerátoru.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Mrázek Vojtěch, Ing., Ph.D.**
Vedoucí ústavu: Sekanina Lukáš, prof. Ing., Ph.D.
Datum zadání: 1.11.2023
Termín pro odevzdání: 9.5.2024
Datum schválení: 25.4.2024

Abstrakt

Neuronové sítě jsou stále populárnější. Inference se v dnešní době provádí nejen na výkonných grafických kartách, ale také na vestavěných systémech s nízkou spotřebou. Tato bakalářská práce zkoumá způsoby testování odolnosti systému proti poruchám na hardwarovém akcelerátoru neuronových sítí. Navrhuje použití FPGA pro zvýšení rychlosti experimentů s odolností proti poruchám. K dosažení tohoto cíle byl použit open-source akcelerátor NVDLA, který byl upraven tak, aby podporoval injektování poruch. Pro demonstraci navrhovaného řešení byla vypracována jednoduchá analýza odolnosti vůči poruchám pomocí sítě ResNet-18.

Abstract

Neural networks are becoming increasingly popular. Inference is now performed not only on high-end GPUs, but also on low-power embedded systems. This bachelor's thesis explores ways to test fault tolerance on the hardware accelerator of neural networks. It propose the use of FPGAs to increase the performance of fault tolerance experiments. To achieve this goal, an open-source accelerator NVDLA was used and modified to support fault injection. Furthermore, an analysis of the fault tolerance of ResNet-18 is presented to demonstrate the proposed solution.

Klíčová slova

Neuronová síť, FPGA, NVDLA, injekce poruch, odolnost proti poruchám, vestavěný systém

Keywords

Neural network, FPGA, NVDLA, fault injection, fault tolerance, embedded system

Citace

MASÁR, Filip. *Inference neuronové sítě na zařízení ZYNQ*. Brno, 2024. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Vojtěch Mrázek, Ph.D.

Inference neuronové sítě na zařízení ZYNQ

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vojtěcha Mrázka, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

....
Filip Masár
1. května 2024

Poděkování

Na tomto místě bych rád poděkoval vedoucímu mé práce, panu Ing. Vojtěchu Mrázkovi, Ph.D., za vstřícnou spolupráci, odborné rady a cenné připomínky. Dále bych chtěl poděkovat své rodině za podporu.

Obsah

1 Úvod **3**

2 Akcelerace neuronových sítí na FPGA **5**
2.1 Obecný princip akcelerátorů . . . 6
2.2 Konkrétní akcelerátory . . . 7

3 Odolnost proti poruchám **12**
3.1 Klasifikace poruch . . . 12
3.2 Injekce poruch . . . 13
3.3 Testování odolnosti akcelerátorů neuronových sítí . . . 14
3.4 Použití aproximačních obvodů . . . 15

4 Návrh řešení pro injekci poruch **16**
4.1 Výběr akcelerátoru . . . 16
4.2 Výběr konfigurace NVDLA . . . 16
4.3 Výběr hardwaru . . . 17
4.4 Návrh injekce poruch . . . 17
4.5 Způsob inference neuronové sítě . . . 20
4.6 Architektura . . . 22
4.7 Výběr neuronové sítě pro testování . . . 22

5 Implementace hardwaru **24**
5.1 Generování RTL popisu . . . 24
5.2 Úprava RTL popisu pro FPGA . . . 25
5.3 Rozšíření RTL popisu o injekci poruch . . . 25
5.4 Vytvoření projektu ve Vivado Design Suite . . . 26

6 Implementace softwaru **32**
6.1 PetaLinux . . . 32
6.2 Příprava microSD karty . . . 36
6.3 NVDLA UMD . . . 37
6.4 Tengine . . . 37
6.5 Aplikace pro injekci poruch . . . 38

7 Experimenty **39**
7.1 Vyhodnocení architektury a parametrů . . . 39
7.2 Injekce poruch . . . 42
7.3 Dopad postupného sčítání v MAC jednotkách . . . 52

8 Závěr **53**

Literatura **55**

A Obsah DVD **58**

Kapitola 1

Úvod

V dnešní době je používání neuronových sítí stále populárnější než v minulosti. Inference neuronových sítí je možné provádět i na vestavěných systémech s nízkou spotřebou energie. Provést ji přímo v procesoru vestavěného systému by však byla velmi náročná úloha, jak z hlediska energetické náročnosti, tak časové náročnosti. To je způsobeno tím, že CPU je univerzální výpočetní jednotkou, která je příliš složitá a energeticky náročná pro jednoduché aritmetické operace během inferencí. Proto se využívají různé metody akcelerace, ať už pomocí speciálních periferií CPU nebo speciálních obvodů.

To umožňuje ještě větší rozšíření neuronových sítí, protože není nutné používat pouze výkonné a energeticky náročné grafické karty. V některých oblastech, kde nacházejí neuronové sítě své využití, není nutné řešit odolnost systému vůči poruchám. Při selhání nedochází k výrazným chybám, a proto není nutné tuto problematiku tolik řešit. Jako příklad může sloužit neuronová síť v mobilním telefonu, která je určena k vylepšování pořízených fotografií. Nicméně existují oblasti, kde je odolnost proti poruchám kritická a selhání takového systému může mít velmi nepříjemné následky. Jako příklad můžeme uvést automobilový průmysl a autonomně řízená auta. Vzhledem k nutnosti zachovat validní výsledky je potřeba řešit odolnost proti poruchám u neuronových sítí a hardwaru provádějícího jejich inferenci.

Testování odolnosti systému proti poruchám se provádí pomocí zanášení umělých poruch do systému a pozorováním jeho chování. Toto zanášení poruch se nazývá injekce poruch. Injektovat poruchy do reálného hardwaru bývá velmi složité, až nemožné. Proto se často používají simulace, které výrazně zjednodušují proces injekce poruch. V simulaci lze snadno modifikovat všechny komponenty akcelerátoru, a tudíž je možné injektovat poruchy i do částí, které jsou v reálném hardwaru nedostupné. Nevýhodou simulací je však jejich pomalost. Při simulaci na úrovni logických obvodů (RTL) je rychlost výrazně nižší než u reálného akcelerátoru. Proto existují i jiné metody simulace, jako je například vytvoření modelu akcelerátoru, který je jednodušší na simulaci než RTL, nebo simulace pouze části akcelerátoru. Nevýhodou těchto přístupů je nižší přesnost výsledků, jelikož se nejedná o reálnou implementaci akcelerátoru.

Cílem této práce je rozšíření akcelerátoru neuronových sítí o podporu injekce poruch a jeho následná realizace pomocí programovatelného hradlového pole (FPGA). Představené řešení využívá open-source akcelerátoru NVIDIA Deep Learning Accelerator (NVDLA). Použití akcelerátoru na FPGA má oproti simulaci výhodu ve vyšší rychlosti. Další výhodou je možnost zkoumat chování celého akcelerátoru.

Hlavním přínosem představeného řešení v této práci je rychlejší vyhodnocení odolnosti vůči poruchám ve srovnání s pracemi, které k těmto účelům využívají simulaci.

Následující kapitola 2 se věnuje metodám akcelerace neuronových sítí na FPGA. Kapitola 3 obsahuje základní pojmy z oblasti odolnosti systému proti poruchám a popisuje metody injekce poruch. Kapitola 4 se zabývá návrhem rozšíření akcelerátoru neuronových sítí o injekci poruch. V kapitolách 5 a 6 je popsán proces implementace hardwaru a softwaru. Kapitola 7 poté obsahuje vyhodnocení výsledného systému a popis experimentů s injekcí poruch do systému.

Kapitola 2

Akcelerace neuronových sítí na FPGA

S přibývajícím výkonem počítačů již není inference neuronových sítí (neural networks) doménou pouze velmi výkonných CPU a GPU, ale i energeticky nenáročných vestavěných systémů. Z důvodu snížení energetické náročnosti výpočtů, které na CPU vestavěných systémů nejsou efektivní, mají tyto systémy různé způsoby akcelerace pro neuronové sítě. Při inferenci neuronových sítí dochází k velkému množství výpočtů. Tyto výpočty je možné v rámci jedné vrstvy neuronové sítě provádět paralelně, čehož využívá většina způsobů akcelerací.

Mezi nejjednodušší způsoby patří například podpora v CPU pro tzv. single instruction, multiple data (SIMD), což slouží pro paralelní zpracování více dat naráz pomocí jedné instrukce. U architektury ARM, která se často používá ve vestavěných systémech, se jedná konkrétně o rozšíření Neon. V procesorech se rozšíření Neon využívá převážně k urychlení zpracování mediálních dat, ale lze ho použít mimo jiné i k urychlení inference neuronových sítí. Jedná se tedy o způsob akcelerace spojený s CPU.

Efektivnějším řešením je použití speciálních akcelerátorů určených přímo na akceleraci neuronových sítí. Akcelerátor se může nacházet na stejném čipu s procesorem, jako tomu je například u Nvidia Jetson [21]. Nebo může být na vlastním čipu, jako tomu je například u Edge TPU od firmy Google [6], který se k CPU připojuje pomocí PCIe nebo USB sběrnice.

Akcelerace může probíhat plně samostatně, kdy se akcelerátoru předají data k neuronové síti a vstupní data. Akcelerátor poté provede celou inferenci neuronové sítě a vrátí až výsledný výstup. Druhou možností je částečná akcelerace, kdy se akcelerátor používá k urychlení pouze určitých operací, například k vypočítání konvolučních vrstev, a mezivýsledky se pak přenášejí do paměti.

Akcelerátory jsou často vytvořeny jako již hotové čipy (ASIC) a jejich implementace na FPGA je možná pouze v případě zveřejnění zdrojových kódů (např. NVDLA [20]) nebo architektury (např. Eyeriss [4]), kterou si je nutné implementovat samostatně. Existují ovšem i akcelerátory navržené přímo pro FPGA (např. DnnWeaver [25]). Nevýhodou implementace na FPGA může být nižší frekvence a s tím související delší doba trvání inference. Naopak výhodou může být konfigurovatelnost FPGA, díky které se může akcelerátor přizpůsobit dané neuronové síti.

2.1 Obecný princip akcelerátorů

Většina akcelerátorů neuronových sítí je založena na principech pocházejících z paralelního návrhu výpočetních systémů. Hlavní princip je založen na homogenní síti výpočetních jednotek (systolic array). Výpočetní jednotky jsou jednoduché hardwarové entity, specializované na určitou výpočetní operaci, kterou je potřeba v systému opakovaně provádět a je možné ji provádět paralelně. Konfigurovatelnost jednotek se liší.

Nejjednodušší variantou jsou fixní jednotky, které provádí stále stejnou operaci, v tomto případě multiply-accumulate (volně přeložitelné jako násobení a sečtení, zkráceně MAC). Tento typ jednotek je typicky používán v hardwaru. Druhou možností jsou konfigurovatelné jednotky, které umí provádět více různých operací. Nevýhodou je větší komplexnost jednotky, která se odráží na její velikosti a spotřebě. Z tohoto důvodu se typicky nepoužívají v hardwarové implementaci.

Pro akceleraci neuronových sítí se používají právě nekonfigurovatelné jednotky provádějící MAC operace. Architektury akcelerátorů založených na tomto principu se dají dále dělit na další podtypy.

- **Návrh na základě vrstev:** Pro každý typ vrstvy neuronové sítě (konvoluční, ReLU atd.) je v hardwaru vytvořený speciální blok, starající se o provedení dané vrstvy. Základem akcelerátoru je kontrolér, který se stará o komunikaci s pamětí a dostává informace o jednotlivých vrstvách neuronové sítě. Podle toho aktivuje příslušný modul v hardwaru pro danou vrstvu a předá mu potřebná data a převezme výsledky. Tím dojde k dokončení jedné vrstvy neuronové sítě a přejde se na vykonání další vrstvy. Proces se opakuje, dokud nedojde k dokončení všech vrstev neuronové sítě. Jedna z možných optimalizací pro tento typ architektury je zmenšení počtu přístupů do paměti, která bývá obvykle problémem z důvodu dlouhého přístupového času a energetické náročnosti. Kontrolér tak může umět nastavit předávání dat mezi jednotlivými moduly přímo, bez zbytečných přístupů do paměti. Další možností je vytvoření pokročilé paměťové hierarchie přidáním rychlejší paměti (SRAM) v případě, kdy je hlavní paměť pomalejší (DRAM nebo SDRAM). Rychlejší paměť bývá menší než hlavní paměť, z důvodu její náročnosti na plochu na čipu, takže se používá jako vyrovnávací na ukládání mezivýsledků a načítání dat s opakovaným přístupem. Výhodou těchto akcelerátorů bývá obvykle jejich univerzálnost a konfigurovatelnost. Umožňují vyhodnotit libovolnou neuronovou síť, která se skládá z podporovaných vrstev. Architektura akcelerátorů na základě vrstev se často používá na výrobu ASIC, právě díky své univerzálnosti. Příkladem těchto akcelerátorů je třeba NVDLA [20] nebo Eyeriss [4].

- **Převod celé neuronové sítě na hardware:** Při vytváření akcelerátoru je potřeba předem znát neuronovou síť, která se má akcelerovat. Jednotlivé vrstvy neuronové sítě se převedou do hardwaru ve stejné podobě (vynechají-li se optimalizace), podle specifikace neuronové sítě na vstupu. Dojde tedy k vytvoření kompletní neuronové sítě v hardwaru. Předávání dat mezi vrstvami je díky tomu přímé, bez potřeby ukládání do externí paměti. Po převodu vznikne velký akcelerátor, podle velikosti zadané neuronové sítě. Akcelerátor potřebuje přístup k paměti na začátku inference, pro načtení vstupních dat, a na konci, kdy dojde k uložení výsledků. Váhy pro jednotlivé vrstvy mohou být přímo uloženy v hardwaru, nebo nastavitelné, kde v takovém případě je ještě potřeba přístup do paměti při inicializaci vah v hardwaru před začátkem inference. Nevýhodou akcelerátoru je jeho velikost, která je odvozena od velikosti neuronové sítě a nelze ji tedy změnit jinak, než změnou neuronové sítě. Další nevýho-

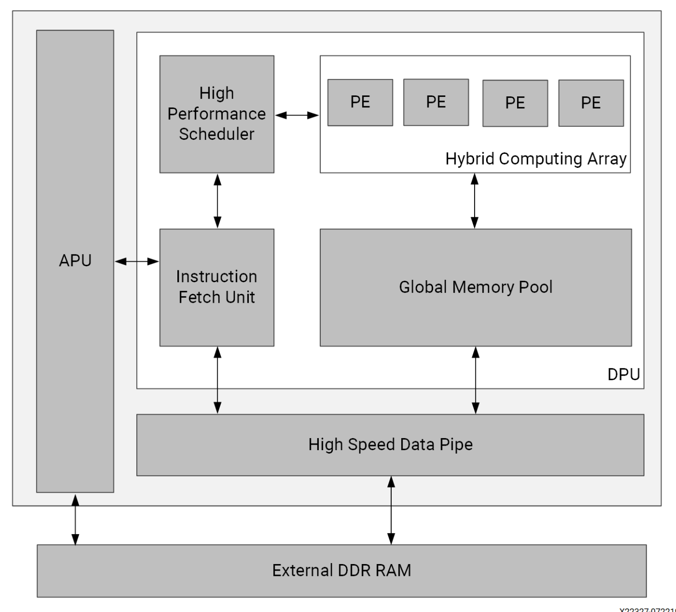
dou je limitace na danou neuronovou síť. Jediné, co může akcelerátor podporovat, je změna vah pro vrstvy. Hlavní výhodou je zvýšená rychlost, díky malému počtu přístupů do paměti. Tento typ akcelerátoru se spíše hodí pro FPGA, kde je možné měnit konfiguraci, než pro výrobu ASIC. Při použití na FPGA je ovšem problém s velikostí celého akcelerátoru. Tento typ akcelerátoru se proto většinou nepoužívá. Příkladem může být akcelerátor z práce [28].

2.2 Konkrétní akcelerátory

Konkrétních akcelerátorů použitelných na FPGA je celá řada. Proto zde bude uveden pouze krátký přehled několika populárních akcelerátorů, ze kterých se vybíralo při návrhu v kapitole 4.1. Všechny uvedené akcelerátory jsou založeny na návrhu na základě vrstev.

2.2.1 AMD Deep Learning Processor Unit

Základním přístupem pro akceleraci neuronových sítí v FPGA doporučovaným výrobcem AMD je použití AMD Deep Learning Processor Unit (DPU) [1]. Důvodem doporučování je kompatibilita se softwarem AMD Vitis AI, který je vyvinut právě za účelem inference neuronových sítí na FPGA od firmy AMD. AMD DPU je určen pro Zynq 7000 SoC a Zynq UltraScale+ MPSoC a je zveřejněn jako IP jádro (IP core) pro Vivado Design Suite. Pro akcelerátor existuje několik architektur, a každou je možné do jisté míry konfigurovat. Všechny architektury používají pro výpočty pole výpočetních prvků, nazvaných Processing Engine. Výhodou akcelerátoru je tedy kompletní podpora ze strany výrobce FPGA a stále udržovaný vývoj, jak samotného akcelerátoru, tak softwaru kolem vývoje a nasazení. Zdrojové kódy jsou ovšem publikované jako šifrovaný RTL popis. To znamená, že je není možné přečíst a ani jakkoliv upravovat nad rámec povolené konfigurace.



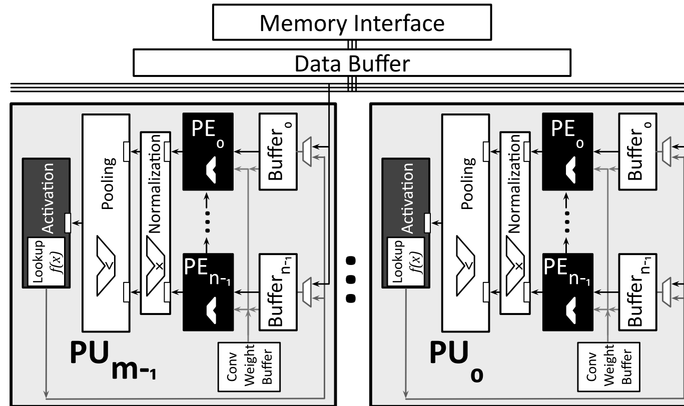
Obrázek 2.1: Architektura DPU akcelerátoru. Převzato z [1].

2.2.2 DnnWeaver

DnnWeaver [25] je projekt zaměřený na převod modelu neuronové sítě do FPGA pro akceleraci výpočtů. Cílem DnnWeaver bylo vytvořit nástroj, který provede převod automatizovaně s využitím co nejvíce dostupných prostředků na daném FPGA. Převod probíhá ze specifikace neuronové sítě v Caffe [9]. Výsledný akcelerátor je tedy vázaný na konkrétní neuronovou síť.

Architektura pro vygenerované akcelerátory (obrázek 2.2) je postavena na výpočetních jednotkách (Processing Units, PUs), kde každá z nich obsahuje několik výpočetních prvků (Processing Engines, PEs). PU obsahuje vyrovnávací paměti pro každou PE, které jsou mapovány do bloků paměti v FPGA, zvaných BRAM. PE potom obsahují ALU jednotku, která podporuje násobení a je mapována do fyzických ALU jednotek na FPGA (DSP). Na základě těchto pravidel je určen maximální počet PU a PE jednotek pro dané FPGA v době generování akcelerátoru.

K frameworku jsou zveřejněny zdrojové kódy a ukázky použití.



Obrázek 2.2: Architektura DnnWeaver akcelerátoru. Převzato z [25].

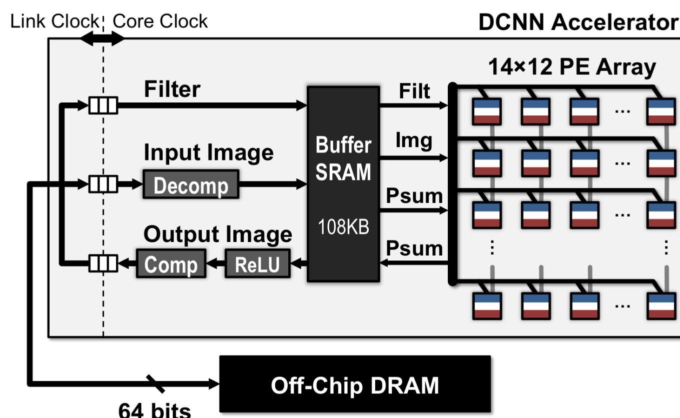
2.2.3 Eyeriss

Eyeriss [4] je energeticky efektivní akcelerátor konvolučních neuronových sítí. Práce se zabývá návrhem akcelerátoru a jeho realizací pomocí ASIC. Pro výpočty v akcelerátoru slouží pole 168 PE. Díky tomuto návrhu se jedná o univerzální akcelerátor, který lze použít bez úprav na akceleraci různých konvolučních neuronových sítí. Autoři uvádějí, že pro AlexNet je spotřeba pouze 278 mW při rychlosti 35 snímků za sekundu. Architektura akcelerátoru je zobrazena na obrázku 2.3. Nejedná se o akcelerátor určený pro FPGA, ale je možné vytvořit implementaci nebo použít již hotovou implementaci architektury a použít ji na FPGA. K akcelerátoru není zveřejněn RTL popis, ale existují implementace popsané architektury se zveřejněným RTL popisem[1].

2.2.4 NVIDIA Deep Learning Accelerator

NVIDIA Deep Learning Accelerator (NVDLA) [20] je open source akcelerátor od firmy NVIDIA, který byl použit například v jednodeskových počítačích Jetson AGX Xavier a Jetson

[1] https://github.com/SingularityKChen/dl_accelerator



Obrázek 2.3: Architektura Eyeriss akcelerátoru. Převzato z [4].

Xavier NX [21]. Jedná se o univerzální konfigurovatelný akcelerátor primárně určený na výrobu ASIC s akcelerátorem. Po drobných úpravách je ovšem možné jeho použití na FPGA.

Akcelerátor je postaven na modulární architektuře s následujícími komponentami:

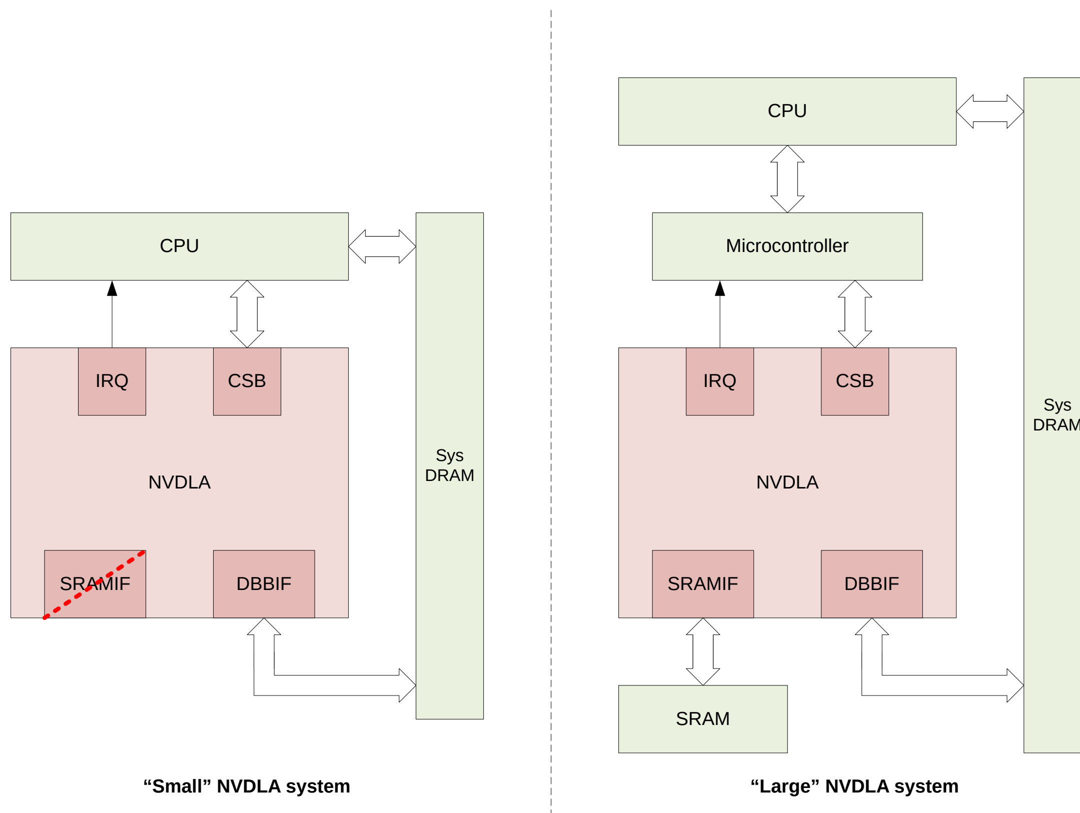
- **Convolution Core:** Slouží k výpočtům konvolučního jádra na základě vstupních dat a vah.
- **Single Data Processor:** Používá se k aplikaci lineárních a nelineárních funkcí jako například ReLU a sigmoid funkce.
- **Planar Data Processor:** Stará se o sdružovací funkce.
- **Channel Data Processor:** Obstarává normalizační funkci (local response normalization, zkratka LRN).
- **Dedicated Memory and Data Reshape Engines:** Provádí transformaci formátu datu (rozdělování, slučování a podobně).

NVDLA má dvě základní konfigurace systému, malý („small“) systém a velký („large“) systém. Hlavní rozdíly jsou v tom, že „small“ systém je určen převážně pro malé vestavěné systémy a internet věcí (Internet of Things, zkratka IoT). Je tedy zjednodušen o určité funkce, jako například vlastní vyrovnávací paměť nebo mikrokontrolér, starající se o řízení akcelerátoru. Na druhou stranu „large“ systém je určen pro výkonnější IoT zařízení, kde je větší důraz na výkon výsledného zařízení. Architektura obou systémů je na obrázku 2.4. Zkrácené porovnání některých parametrů z výchozí konfigurace systémů je uvedeno v tabulce 2.1.

Jak je patrné z obrázku 2.4, NVDLA obsahuje tři hlavní rozhraní:

- **Configuration Space Bus (CSB) rozhraní:** Konfigurační synchronní sběrnice určená pro komunikaci s řídícím procesorem. Sběrnice implementuje velmi jednoduchý protokol, který je možné použít s jinými sběrnicemi, například ABP.
- **Rozhraní pro přerušení:** Jednobitová sběrnice sloužící k vyvolání přerušení v řídícím procesoru.

- **Data Backbone (DBB) rozhraní:** Vysokorychlostní sběrnice pro připojení hlavní systémové paměti. Jedná se o implementaci podobnou AXI protokolu s možností přímého použití s AXI-4 rozhraním.



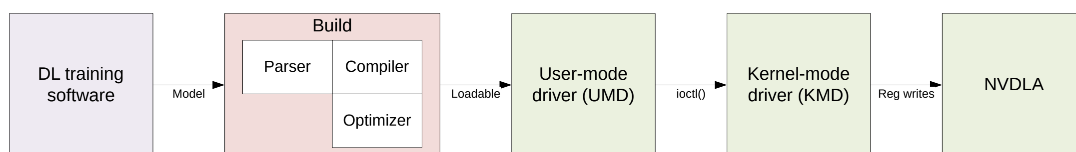
Obrázek 2.4: Porovnání dvou možných konfigurací NVDLA. Převzato z [20].

Vlastnost	**„large" systém**	**„small" systém**
Datový typ	FP16/INT16	INT8
Vyrovnávací SRAM	Ano	Ne
Komprese vah	Ano	Ne
Transformace dat	Ano	Ne
Počet MAC jednotek	16	8
Počet násobiček v MAC jednotce	64	8

Tabulka 2.1: Stručné porovnání výchozích konfigurací systémů NVDLA.

DBB rozhraní má druhou sběrnici, určenou k připojení volitelné vyrovnávací paměti SRAM na čipu, která poskytuje vyšší propustnost a nižší latenci přístupu než hlavní systémová paměť.

Součástí řešení je také softwarová část. Tato část zahrnuje kompilátor neuronových sítí z Caffe [9] na formát NVDLA Loadable. Dále ovladač v uživatelském prostoru, User-mode driver (UMD) a ovladač v prostoru jádra, Kernel-mode driver (KMD). Datový tok softwarové části je naznačen na obrázku 2.5.



Obrázek 2.5: Datový tok softwarové části NVDLA. Převzato z [20].

Kapitola 3

Odolnost proti poruchám

Řešení odolnosti systému proti poruchám je důležité pro bezpečnost daného systému. Na každý systém se kladou jiné nároky na bezpečnost, a podle toho je i potřeba zajistit příslušnou odolnost systému proti poruchám. Definice a pojmy použité v této kapitole pochází ze zdrojů [3, 10, 11]. Na začátku je potřeba upřesnit používané pojmy pro popis jednotlivých jevů:

- **Porucha** (fault) je fyzický defekt, nedokonalost nebo závada vyskytující se v nějaké komponentě.
- **Chyba** (error) je odchylka od přesnosti nebo správnosti způsobená projevem poruchy.
- **Selhání** (failure) je neprovedení očekávané činnosti kvůli výskytu chyby.

Měření odolnosti systému není možné jakkoliv určit pomocí jedné univerzální metody. Používá se proto několik různých metrik, jako například spolehlivost, dostupnost, bezpečnost, střední doba mezi poruchami (Mean Time Between Failures, zkratka MTBF) a latence při poruše.

Poruchy mohou být buď v hardwaru nebo v softwaru. Jelikož náplní této práce je simulace poruch v hardwarové části akcelerátoru, zaměřím se v textu na popis poruch v hardwaru.

3.1 Klasifikace poruch

Poruchy je možné rozdělit do různých kategorií podle několika aspektů. Jednou z možností je rozdělení podle doby trvání na:

- **Trvalé poruchy** (permanent faults): Nacházejí se v systému celou dobu a jsou neměnné. Příkladem trvalé poruchy může být vyhořelý rezistor.
- **Přechodné poruchy** (transient faults): Ovlivňují systém nějakou dobu a poté zmizí. Příkladem může být elektromagnetické rušení.
- **Občasné poruchy** (intermittent faults): Oscilují mezi ovlivňováním systému a stavem bez ovlivňování. Příkladem může být špatný kontakt v konektoru.

Další možností rozdělení je podle příznivosti:

- **Příznivé poruchy** (benign fault): Způsobují, že celá jednotka přestane fungovat.

- **Zákeřné poruchy** (malicious fault): Vytvářejí věrohodné výsledky, ale jiné než má jednotka správně vracet.

3.2 Injekce poruch

Injekce poruch (fault injection) je definována jako technika ověřování spolehlivosti, která je založená na realizaci řízených experimentů, při nichž je pozorováno chování systému v době výskytu poruchy.

Vzhledem k tomu, že se poruchy běžně v systému nevyskytují, dochází při sledování systému k injekci uměle vytvořených chyb, a tím k výraznému urychlení experimentů. Injekci lze provádět jak v simulaci, tak i na reálném hardwaru. Podle toho se také rozděluje do dvou kategorií.

3.2.1 Injekce poruch na reálném hardwaru

Na reálném hardwaru se provádí injekce poruch pomocí systému, který je vyrobený jako reálný hardware, a používají se techniky pro způsobování umělých poruch v systému. Používá se převážně pro analýzu finálního návrhu.

Příklady výhod:

- Provádění experimentů v reálném čase, což znamená rychlé provedení, a to umožňuje spouštění velkého množství experimentů.
- Možnost detekce reálných poruch ve vyrobeném hardwaru.
- Zjištění chování reálného hardwaru, který je v produkci.
- Vhodnější pro poruchy na nízké úrovni.

Příklady nevýhod:

- Možné riziko poškození hardwaru.
- Některé části nemusí být dostupné k injekci z důvodu vysoké integrace.
- Limitované možnosti injekce poruch podle návrhu hardwaru.
- Špatná přenositelnost na jiný systém.
- Delší čas na přípravu experimentů.

3.2.2 Injekce poruch v simulaci

Injekce chyb v simulaci (simulation-based) probíhá na systému, který je vytvořený jako simulační model (nejčastěji VHDL nebo Verilog). Řešení je vhodnější při analýze odolnosti proti poruchám ve fázi vývoje, protože je jednodušší implementovat případné změny do implementace hardwaru.

Příklady výhod:

- Maximální možnost sledování a ovládání systému.
- Nedochází k nežádoucímu narušení systému.

- Nízká cena, jelikož nevyžaduje žádný speciální hardware.
- Možnost injekce časově závislých poruch, protože je možné provést injekci v nulovém čase (pro simulovaný systém).

Příklady nevýhod:

- Nutnost vytvoření modelu.
- Velká časová náročnost na provedení experimentu.
- Přesnost výsledků záleží na kvalitě použitého modelu.
- Model nemusí obsahovat všechny možné poruchy jako reálný návrh.

3.2.3 Injekce poruch v FPGA

Určitou kombinací obou metod je použití FPGA pro testování hardwaru. Testování poté probíhá v reálném čase (ale třeba na menší frekvenci) bez potřeby vyrábět hardware fyzicky a je možné hardware v FPGA rychle upravovat. Na druhou stranu se přijde o možnost provádět experimenty na nejnižší úrovni, o plné ovládání systému a o možnost injektovat časově závislé poruchy.

3.2.4 Další možnosti rozdělení

Další možností, jak rozdělit metody injekce poruch, jsou podle přístupu na:

- **Invazivní**: Metoda po sobě zanechává stopu v systému. Může například dojít ke zvýšení latence, a tím k problémům s časováním.
- **Neinvazivní**: Nedojde k zanechání žádné stopy kromě samotné injekce poruchy.

Při injekci poruch pomocí VHDL a Verilog se ještě rozděluje podle použité techniky.

- **Technika sabotérů** (Saboteurs Technique): Injekce poruchy je realizována pomocí speciální komponenty, která přeruší ovlivněný signál a nahradí ho injektovanou hodnotou. Komponenta navíc může poskytovat originální hodnotu do řídícího systému, pro další zpracování. Nevýhodou jsou přidané kontrolní signály pro komponentu.
- **Technika mutantů** (Mutants Technique): Komponenta pro injekci poruch nahrazuje původní komponentu novou komponentou. Nová komponenta se ve výchozím stavu chová stejně jako původní a až po aktivaci se změní její fungování na injektování poruchy.

3.3 Testování odolnosti akcelerátorů neuronových sítí

Při testování odolnosti systému proti poruchám je potřeba provést velké množství experimentů s injekcí poruch. Počet potřebných experimentů se odvíjí od komplexnosti daného systému. Systémy s neuronovými sítěmi vyžadují pro rozsáhlou analýzu odolnosti vůči chybám provést desítky milionů injekcí poruch [8, 24]. I když je v dnešní době možné provádět inferenci neuronových sítí už i na vestavěných systémech, stále se jedná o výpočetně náročnou úlohu. Po přidání výpočetní náročnosti na simulaci takového akcelerátoru na RTL

úrovni se dostáváme do časů, při kterých není možné provést ani částečnou analýzu odolnosti. Existují proto práce, které se na tento problém zaměřují a snaží se nějakými způsoby zmenšit výpočetní náročnost. Konkrétní způsoby zrychlení se liší, ale v základu se většinou jedná o zjednodušení simulačního modelu, při snaze zachovat původní přesnost.

3.3.1 Softwarový model na základě hardwaru

Softwarový model na základě hardwaru pro injekci poruch (software-based hardware-aware fault injection) je jedním z možných způsobů urychlení injekce poruch. Injekce poruch v simulaci je většinou prováděna bez ohledu na reálný hardware nebo RTL návrh. Díky tomu jsou simulace mnohem rychlejší, ale ztrácejí přesnost právě kvůli chybějící vazbě na hardware. Řešením této nepřesnosti se zabývají práce FIdelity [8] a SAFFIRA [27]. Obě práce řeší problém pomocí vytvoření simulačního modelu s ohledem na daný hardware a přináší frameworky pro testování pomocí tohoto konceptu.

FIdelity vytváří simulační model na základě reálného akcelerátoru. Na ověření byl použit akcelerátor NVDLA. Před vytvářením simulačního modelu se prvně analyzuje RTL model reálného akcelerátoru, a tím se zjišťují možná senzitivní místa na poruchy. Poté se vytvoří simulační model s ohledem na slabá místa získaná v předešlém kroku.

SAFFIRA používá simulační model založený na homogenní síti z výpočetních jednotek (systolic array, SA). Simulační model SA je založený na systému rekurentních rovnic (Uniform Recurrent Equation). Tento způsob umožňuje do hodnot v simulovaném modelu injektovat chyby s ohledem na hardware.

Práce se tedy liší ve vazbě výsledného simulačního modelu na hardware.

3.4 Použití aproximačních obvodů

Aproximační aritmetické obvody poskytují podobný výstup jako reálné obvody, ale jsou jednodušší na hardwarovou realizaci. Na zrychlení akcelerátorů neuronových sítí je tedy možné použít upravený akcelerátor, který používá právě aproximační aritmetické obvody. Tím jsou ale do systému zaneseny poruchy vzniklé nepřesnými výpočty, a je potřeba ověřit jejich vliv na celý systém. Poruchy v systému tedy vznikají při návrhu záměrně, s cílem urychlit akcelerátor. Nejedná se tedy o typické testování odolnosti akcelerátoru vůči poruchám, kdy jsou chyby vytvářeny uměle, aby došlo k testování odolnosti akcelerátoru při provozu. Cílem je hlavně přinést způsob urychlení akcelerace pro reálné akcelerátory s přijatelnou chybou ve výsledku. Tento koncept je například použit v práci [2].

Práce pracuje s aproximačními obvody z knihovny EvoApproxLib [16]. Simulace výpočtů pomocí aproximací by ovšem byla na CPU nebo GPU pomalá, protože daný hardware nemá podporu aproximovaných výpočtů. Bylo by tedy potřeba provádět celou simulaci aproximačních obvodů. V práci byl proto navržen způsob rychlejšího použití aproximačních obvodů na základě jejich implementace pomocí vyhledávacích tabulek (look-up tables) v paměti GPU s podporou CUDA.

Kapitola 4

Návrh řešení pro injekci poruch

V době vytváření této práce se většina podobných prací zaměřovala na testování odolnosti proti poruchám pouze v simulaci. Práce různými způsoby zjednodušovaly modely akcelerátorů neuronových sítí, aby bylo možné provádět simulace rychleji. Nebyla nalezena žádná práce, která by se zajímala o injekci poruch do reálného hardwaru, ať už ve formě FPGA nebo ASIC. Tento způsob ovšem přináší různé výhody, jako například větší rychlost akcelerátoru, protože vše funguje v reálném čase. Dále umožňuje testovat reálný akcelerátor místo modelu akcelerátoru. Z těchto důvodů bylo rozhodnuto, že se tato práce bude zabývat právě injekcí poruch do reálného akcelerátoru, který bude realizován v FPGA.

Tato kapitola se věnuje návrhu architektury, která umožní dosažení výše popsaného cíle. Na začátku bude potřeba vybrat vhodný akcelerátor a k němu hardware. Poté následuje jednoduchý rozbor fungování akcelerátor se zaměřením na požadovanou část, kam se budou injektovat poruchy. Po analýze této části bude možné navrhnout způsob injekce poruch. Dále ještě bude nutné vyřešit otázku, jak provádět inferenci neuronových sítí na akcelerátoru. Na závěr bude dokončen návrh architektury ve spojení se softwarovým řešením. Po dokončení architektury bude nutné vybrat vhodnou neuronovou síť, která bude použita při testování odolnosti proti poruchám.

4.1 Výběr akcelerátoru

Na začátku práce bylo nutné vybrat vhodný akcelerátor k použití. Jelikož bude potřebné rozšířit zdrojové kódy akcelerátoru o injekci poruch, byl výběr omezen pouze na akcelerátory s otevřeným zdrojovým kódem. Univerzální způsob injekce poruch do akcelerátoru by se obtížně realizoval při dynamickém generování akcelerátoru pro konkrétní neuronovou síť. Výběr se proto ještě zúžil na univerzální akcelerátory. Dynamicky generované akcelerátory jsou navíc myšleny pouze pro FPGA, což by odchýlilo práci od plánovaného záměru injektovat poruchy do akcelerátoru, který je možné vyrobit na čipu. Tímto postupným vyřazováním nakonec zůstal pouze jeden akcelerátor, a to NVDLA.

4.2 Výběr konfigurace NVDLA

Při návrhu bylo potřeba vybrat, která konfigurace NVDLA bude implementována. Pro implementaci výchozí konfigurace „large“ systému na FPGA je potřeba mít FPGA o podobné velikosti jako Virtex UltraScale XCVU440[1], což nespadá do kategorie běžně dostupných

[1] https://github.com/nvdla/hw/issues/110

FPGA a je mimo možnosti zapůjčení z fakulty. Teoreticky by mohlo být možné pozměnit konfiguraci a pokusit se použít nějak omezenou „large“ konfiguraci. Pro práci ovšem nebylo nutné využít zrovna „large“ konfiguraci, takže bylo rozhodnuto použít „small“ konfiguraci NVDLA, která se vejde i na dostupná FPGA.

4.3 Výběr hardwaru

Při výběru hardwaru byly dostupné dvě varianty. První varianta byla AUP PYNQ-Z2 s FPGA Zynq-7000 SoC a druhá varianta byla UltraScale+ MPSoC ZCU104 Evaluation Kit s FPGA Zynq UltraScale+ XCZU7EV. Deska AUP PYNQ-Z2 obsahuje pouze 512 MiB DDR3 RAM, zatím co UltraScale+ MPSoC ZCU104 Evaluation Kit obsahuje 2 GiB DDR4 RAM a rozšiřující SO-DIMM slot pro externí DDR4 RAM.

Použití AUP PYNQ-Z2 nepřipadá v úvahu, vzhledem k tomu, že „small“ konfigurace NVDLA vyžaduje alespoň 256 MiB RAM [14]. Navíc bylo zjištěno, že je potřeba zhruba 94 000 LUT jednotek a AUP PYNQ-Z2 má pouze 53 200 LUT jednotek[2]. Pro řešení byla tedy vybrána deska UltraScale+ MPSoC ZCU104 Evaluation Kit s rozšířením o 4 GiB externí DDR4 RAM do SO-DIMM slotu. Tím bude k dispozici dohromady 6 GiB RAM. NVDLA bude možné přidělit 1 GiB RAM a pro systém zůstane 5 GiB, což bude podstatné při návrhu softwarové části v kapitole 6.1.4.

4.4 Návrh injekce poruch

Porucha může do systému vstoupit několika způsoby. Například může dojít k poškození dat v paměti. Tento druh poruchy lze jednoduše simulovat ze softwaru. Stačí na úrovni jádra operačního systému změnit některé bity v paměti vyhrazené pro akcelerátor. Složitější je injektovat poruchy do výpočetní části akcelerátoru. Tento způsob injekce je nutné provést úpravou samotného hardwaru, aby došlo k simulaci poruchy. Při řešení této práce se rozhodlo zaměřit se právě na injekci poruch do výpočetní části. Před samotným návrhem injekce poruch do NVDLA je ovšem potřebné prvně pochopit princip fungování akcelerátoru.

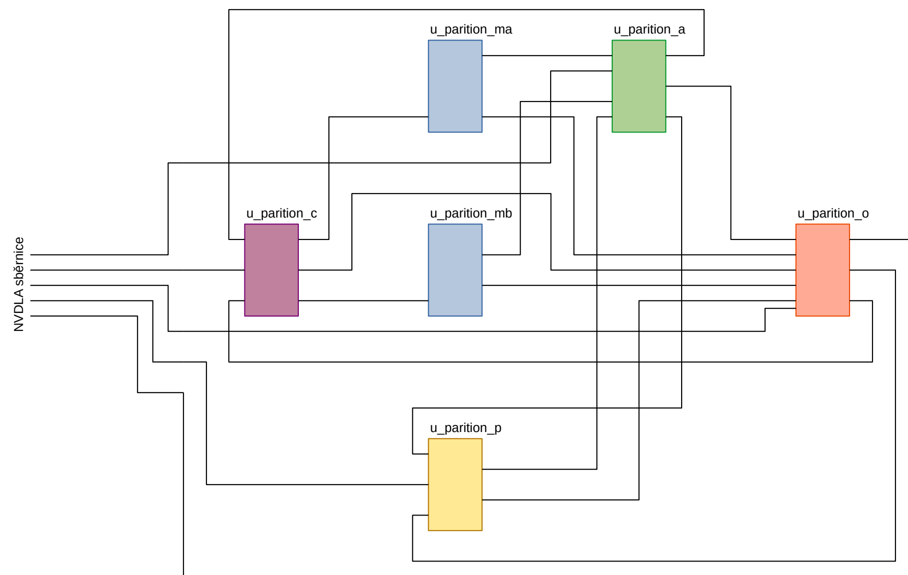
4.4.1 Rozbor NVDLA

Základem NVDLA je 6 oddílů (partitions), z nichž jsou dva stejného typu. Zjednodušené schéma propojení jednotlivých oddílů je na obrázku 4.1.

Jednotlivé oddíly a jejich funkce:

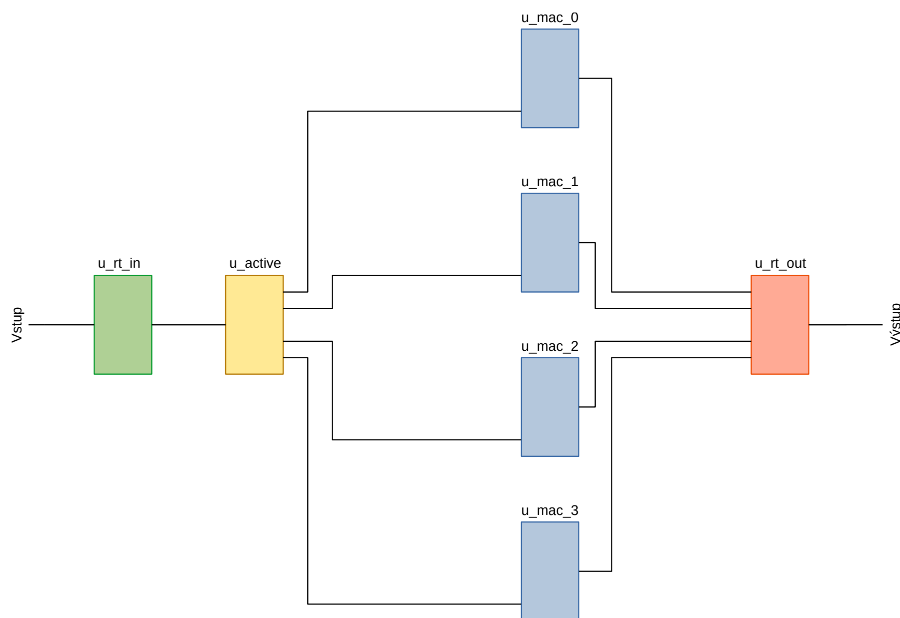
- **partition c:** Spravuje jednotlivé operace konvolučního jádra.
- **partition m:** Oddíl starající se o operace násobení a sčítání.
- **partition p:** Zajišťuje různé lineární a nelineární funkce.
- **partition a:** Sčítá částečné součty z MAC jednotek a posílá výsledky k dalšímu kroku.
- **partition o:** Řídí komunikaci mezi vnitřními oddíly a externím systémem.

[2] https://www.amd.com/en/corporate/university-program/aup-boards/pynq-z2.html



Obrázek 4.1: Zjednodušené schéma propojení jednotlivých oddílů v NVDLA.

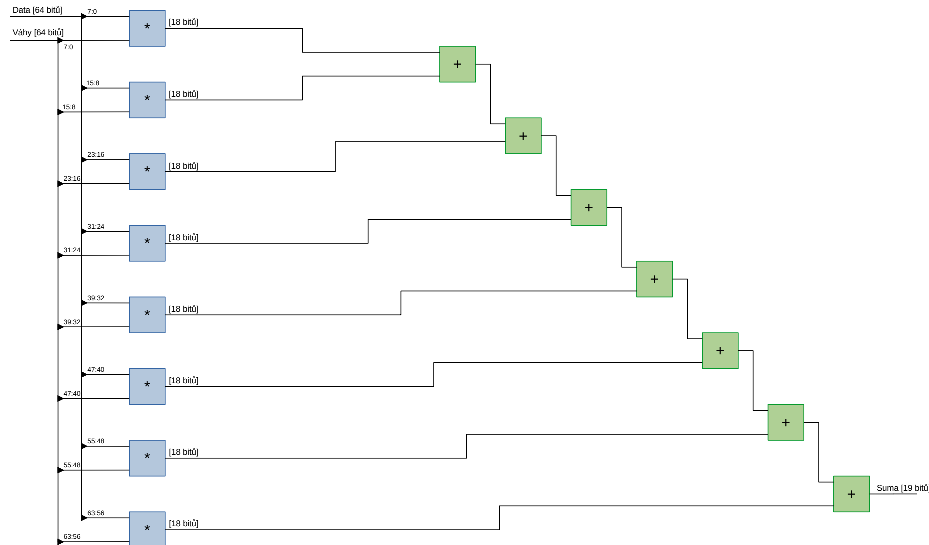
Partition m se nachází v návrhu NVDLA dvakrát a každý oddíl obsahuje modul s MAC jednotkami. Zjednodušené schéma modulu s MAC jednotkami je na obrázku 4.2. V modulu jsou vstupní registry, aktivační blok, čtyři MAC jednotky a výstupní registry. Aktivační blok se stará o přidělování dat a vah jednotlivým MAC jednotkám. Do modulu vstupují pouze data a váhy pro jednu MAC jednotku v daném čase. Aktivační blok tyto data přebírá a ukládá v příslušných registrech pro danou MAC jednotku.



Obrázek 4.2: Zjednodušené schéma modulu s MAC jednotkami v *partition m*.

Uvnitř každé MAC jednotky (obrázek 4.3) je osm 18bitových násobiček a sedm 19bitových sčítaček, počítajících výslednou sumu. Zajímavé je, že sčítačky nejsou ve stromové

struktuře. Součet tedy probíhá postupně a je možné, že dochází ke zhoršení času, a tím ke snížení maximální frekvence. Jako drobné rozšíření práce bude zkoumán dopad tohoto sčítání v kapitole s experimenty 7.3. Celkově se tedy v NVDLA nachází mřížka 8x8 násobiček, které mohou vytvářet chybný výstup.



Obrázek 4.3: Zjednodušené schéma vnitřní části modulu MAC jednotky.

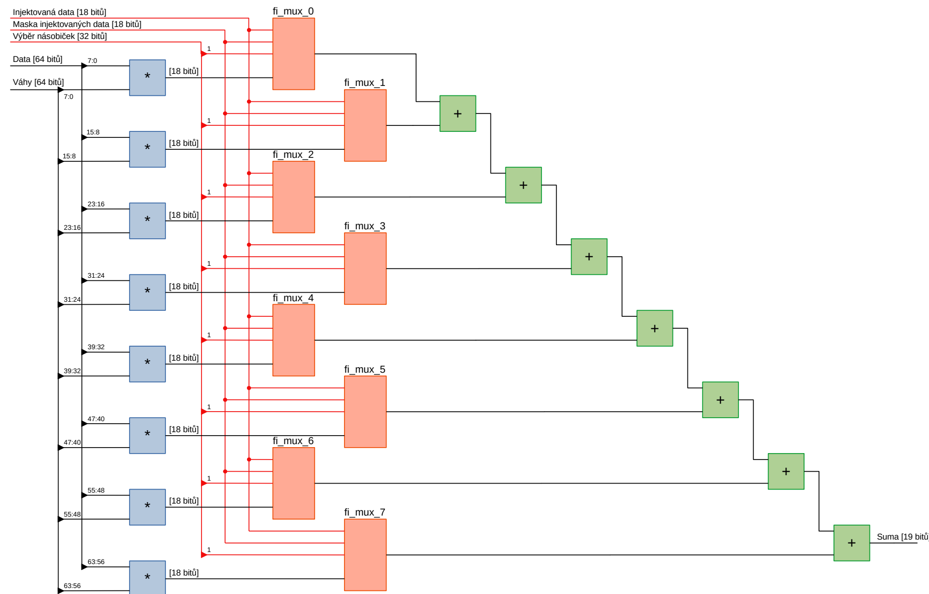
4.4.2 Rozšíření o injekci poruch

Rozhodl jsem se, že injekce poruch bude probíhat pomocí úpravy RTL kódu NVDLA, tzn. invazivní metodou. Vhodnějším řešením by bylo testovat odolnost vůči poruchám bez zásahu do testovaného hardwaru, tzn. neinvazivní metodou. Toho by šlo dosáhnout například pomocí modifikace FPGA bitstreamu [13]. Takový způsob řešení by byl ovšem časově náročnější, protože by bylo nutné pro každou testovanou poruchu vytvořit nový bitstream, a ten následně nahrát do FPGA. Mnou navržené řešení je schopné dynamicky měnit injektované poruchy za běhu systému (mělo by být možné měnit poruchy s každým tikem hodinového signálu). Dalším důvodem pro zvolení úpravy RTL kódu je fakt, že modifikace bitstreamu není jednoduchá záležitost a provedení takového způsobu injekce poruch by překročilo rozsah této práce.

Navrhl jsem, že se bude porucha vkládat pomocí injektoru za násobičkou. Injektor bude realizován modulem vytvořeným pro tyto účely. Za každou násobičkou se bude nacházet tento modul a bude přerušovat spojení mezi násobičkou a sčítačkou. Modul bude umožňovat nahrazení vybraných bitů na konkrétní hodnotu, tzn. bude simulovat zkrat jednotlivých bitů do logické 1 nebo 0. Modul by ovšem po úpravě mohl podporovat i jiné ovlivnění výstupu násobičky, jako například invertování hodnoty jednotlivých bitů. Hodnotu pro každý bit bude možné vybrat nezávisle na ostatních bitech pomocí injektované 18bitové hodnoty. Modul tedy bude umět simulovat trvalé, přechodné a občasné poruchy. Po další úpravě by mohl podporovat i čtení původní hodnoty.

Upravené schéma s moduly pro injekci poruch je na obrázku 4.4. Kvůli zmenšení sběrnice pro injekci dat budou injektovaná data a maska napojeny na společnou sběrnici. Bude tedy

možné zvolit, která násobička bude ovlivněna, ale nebude možné zvolit způsob ovlivnění pro každou násobičku zvlášť.



Obrázek 4.4: Zjednodušené schéma vnitřní části modulu MAC jednotky s rozšířením o injekci poruch.

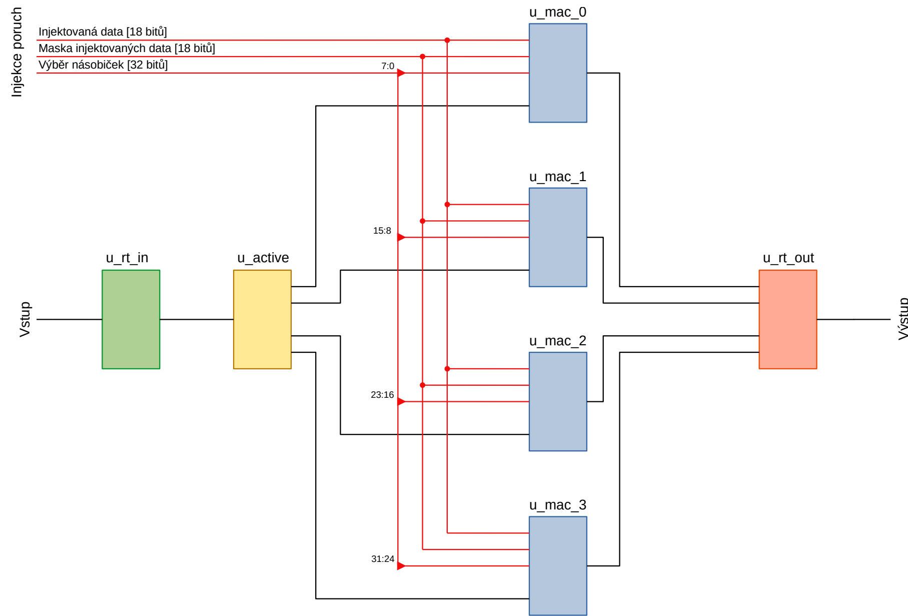
Řízení injektovaných hodnot bude probíhat pomocí externích komponent mimo NVDLA akcelerátor. Z toho důvodu je nutné všechny sběrnice pro injekci dat vyvést mimo akcelerátor. Nejprve dojde k vyvedení sběrnic z MAC jednotek (obrázek 4.5). Sběrnice pro injektovaná data a masku budou sdíleny pro všechny MAC jednotky a sběrnice pro výběr násobiček budou spojeny do jedné větší sběrnice. Tím se dostanou do *partition ma* a *partition mb*, ze kterých budou vyvedeny do nejvyšší části NVDLA. V nejvyšší části NVDLA (obrázek 4.6) dojde opět ke sdílení sběrnic pro data a masku z *partition ma* a *partition mb*. Sběrnice pro výběr násobiček nebudou spojeny, aby nevznikla příliš velká sběrnice.

Aby bylo možné poruchy dynamicky měnit ze softwaru, budou sběrnice pro injekci poruch připojeny k procesoru. Bylo by i možné připojit datovou sběrnici například ke generátoru pseudonáhodných čísel a měnit tak injektované hodnoty v průběhu inference neuronové sítě. Měnit hodnoty během inference půjde do jisté míry i s navrženým řešením, ale čas změn bude závislý na frekvenci procesoru a samotné rychlosti předání dat do modulu. Kvůli nejistotě, jak dlouho by trvala změna poruch, se budou poruchy nastavovat pouze před začátkem inference s malou časovou prodlevou, aby bylo jisté, že došlo k jejich propagaci.

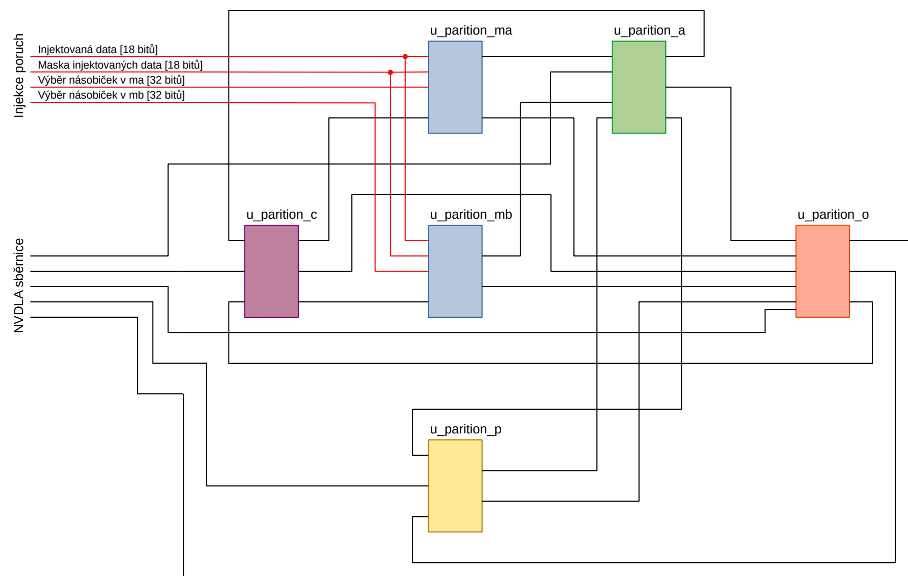
4.5 Způsob inference neuronové sítě

Existují dvě možnosti, jak spustit inferenci neuronové sítě na NVDLA:

1. **Software od NVDLA:** Jak již bylo částečně popsáno v teoretické části (kapitola 2.2.4), NVDLA implementace obsahuje i softwarové řešení pro spouštění neuronových sítí. Nejprve je potřeba převést neuronovou síť na formát NVDLA Loadable pomocí kompilátoru. NVDLA kompilátor podporuje pouze Caffe modely. V plánech je sice



Obrázek 4.5: Zjednodušené schéma modulu s MAC jednotkami v *partition m* s rozšířením o injekci poruch.



Obrázek 4.6: Zjednodušené schéma propojení jednotlivých oddílů v NVDLA s rozšířením o injekci poruch.

podpora ONNX[3], ale projekt byl opuštěn dřív, než došlo k implementaci. Dalším krokem je spouštění NVDLA Loadable pomocí User Mode Driver (UMD), což je ovladač v uživatelském prostoru, který komunikuje s Kernel Mode Driver (KMD) v prostoru jádra. Jelikož „small“ konfigurace podporuje pouze celočíselnou aritmetiku, je potřeba mít k neuronové síti kalibrační tabulku ke kvantizaci. Oficiálním způsobem pro vy-

[3] https://github.com/nvdla/sw/blob/master/Roadmap.md

generování těchto dat je pomocí použití NVIDIA TensorRT, což je SDK určené pro inferenci neuronových sítí na grafických kartách od firmy NVIDIA. Je tedy potřeba mít k dispozici grafickou kartu NVIDIA podporující TensorRT, jinak není možné vytvořit si vlastní NVDLA Loadable pro celočíselnou aritmetiku.

2. **Tengine:** Jedná se o open source framework vytvořený OPEN AI LAB, určený na rychlé a efektivní nasazení neuronových sítí na vestavěných zařízeních [22]. Tengine má vlastní formát neuronových sítí, do kterého se dají zkonvertovat formáty z jiných frameworků jako například TensorFlow nebo Caffe. V případě nasazení s NVDLA je tento formát na začátku převeden na NVDLA Loadable pomocí NVDLA kompilátoru a až poté je neuronová síť spuštěna pomocí NVDLA UMD. Jedná se tedy o nadstavbu nad oficiálním řešením od firmy NVIDIA. Tengine navíc umožňuje neuronovou síť při převodu optimalizovat, aby došlo ke zrychlení vyhodnocení. Další užitečnou součástí frameworku Tengine je hotový nástroj na kvantizaci neuronové sítě, což umožňuje jednoduchý převod z aritmetiky s plovoucí desetinnou čárkou na celočíselnou aritmetiku. Kvantizace probíhá na procesoru a jediná limitace je architektura procesoru, a to přesně x86 (a x86-64). Je tedy mnohem flexibilnější než oficiální řešení od firmy NVIDIA. Další výhodou frameworku Tengine je možnost inference modelu s neuronovou sítí na všech podporovaných platformách bez jakékoliv úpravy.

Pro implementaci jsem se rozhodl použít Tengine, jelikož se jedná o více propracované řešení. Tengine navíc má i vlastní model zoo, což zjednodušuje implementaci, protože nebude potřeba trénovat vlastní neuronovou síť.

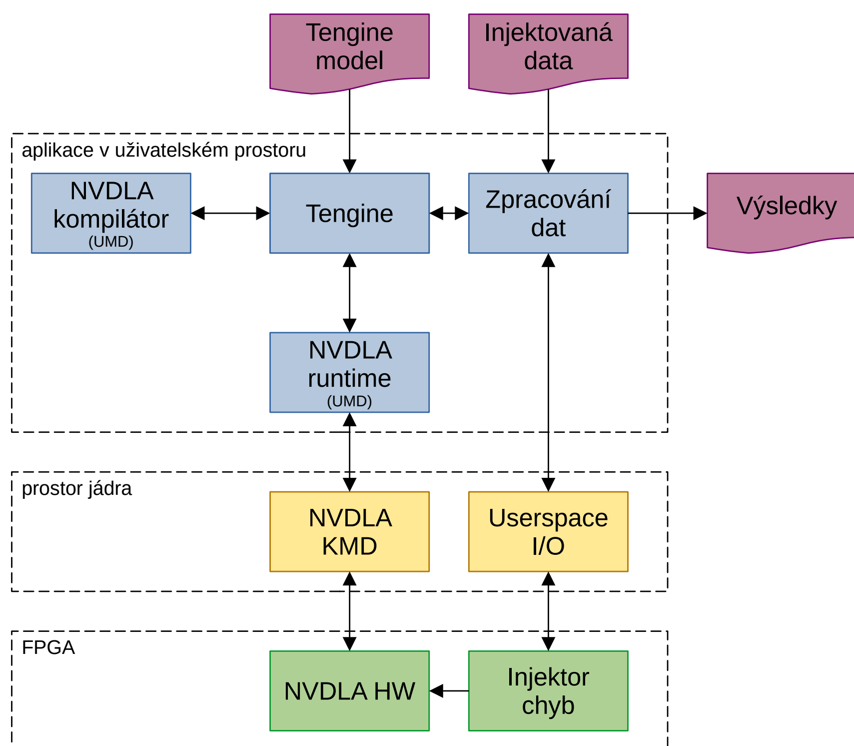
4.6 Architektura

Architektura systému na testování odolnosti NVDLA vůči poruchám se bude skládat z aplikace, ovladačů a hardwaru (obrázek 4.7). Uživatelská aplikace bude obsahovat Tengine, který se bude starat o inferenci neuronové sítě pomocí NVDLA UMD. Dále se zde bude nacházet mnou vytvořená část pro zpracování injektovaných dat, která bude načítat injektovaná data ze souboru. Načtená data budou předávána do ovladače, který se nachází v prostoru jádra (na obrázku 4.7 pojmenovaný jako Userspace I/O). Dále bude komunikovat s frameworkem Tengine a spouštět pomocí něho inferenci po injekci poruch. V prostoru jádra se bude nacházet ovladač NVDLA KMD, který bude zprostředkovávat komunikaci mezi NVDLA UMD a hardwarem NVDLA. Také zde bude již zmíněný ovladač, starající se o předání dat o injektovaných poruchách do hardwaru. V hardwarové části FPGA bude samotný NVDLA akcelerátor a modul pro injekci poruch do akcelerátoru.

4.7 Výběr neuronové sítě pro testování

Jelikož trénování vlastní neuronové sítě není zahrnuto do rozsahu této práce, bude pro zjednodušení použita neuronová síť z Tengine model zoo. Na výběr jsou jenom dva modely neuronových sítí schopné inference na NVDLA (kvantizované na INT8), a to ResNet-18 trénovaný na datasetu CIFAR-10 a YOLOX-nano, nejspíše trénovaný na COCO datasetu.

ResNet [7] je architektura konvoluční neuronové sítě určená pro klasifikaci obrázků. Architektura obsahuje tzv. reziduální spojení, která pomáhají přenášet informace přes různé vrstvy, a tím usnadňují trénování. ResNet-18 je konkrétní implementace této architektury. CIFAR-10 [12] je dataset používaný při testování algoritmů klasifikace obrázků. Dataset



Obrázek 4.7: Architektura systému na testování odolnosti NVDLA vůči poruchám.

obsahuje 60 000 obrázků, 50 000 pro trénování a 10 000 pro testování, o rozměrech 32 na 32 pixelů. Obrázky jsou rozděleny do 10 různých tříd: letadlo, automobil, pták, kočka, jelen, pes, žába, kůň, loď a kamion.

YOLOX [5] je vylepšená verze YOLO [23], což je konvoluční neuronová síť pro detekci a klasifikaci objektů v obraze. Hlavní myšlenkou YOLO je provést detekci a klasifikaci objektů v jednom průchodu. YOLOX-nano je zmenšená varianta YOLOX se vstupním rozlišením obrázku o rozměrech 416 na 416 pixelů. Microsoft COCO [15] je rozsáhlý dataset určený pro trénování a testování neuronových sítí pro detekci a klasifikaci obrázků. Dataset obsahuje 328 000 obrázků, na kterých se nachází 91 typů objektů.

Rozhodl jsem se použít neuronovou síť ResNet pro testování odolnosti NVDLA vůči poruchám. Důvodem pro toto rozhodnutí je menší výpočetní náročnost neuronové sítě a jednodušší vyhodnocení celkové přesnosti. Pokud by byla zvolena síť YOLOX-nano, mohlo by dojít k příliš dlouhému čekání na výsledky experimentů. Dokonce i při použití ResNet-18 lze očekávat potřebu několika hodin na vyhodnocení většího množství injektovaných poruch.

Kapitola 5

Implementace hardwaru

Při návrhu hardwaru byl použit operační systém Ubuntu 20.04.6 LTS a návrh byl realizován v softwaru Vivado Design Suite 2022.1. Verze Ubuntu byla zvolena podle nejnovější podporované verze pro danou verzi softwaru Vivado Design Suite [32].

5.1 Generování RTL popisu

Zdrojové kódy popisu hardwaru NVDLA jsou zveřejněné v GitHub repozitáři [18]. Nejedná se o hotový RTL popis, ale o model pro vygenerování RTL popisu na základě vybrané konfigurace.

Repozitář obsahuje tři větve. Výchozí větev s názvem `nvdlav1` obsahuje nekonfigurovatelnou verzi „full-precision" NVDLA. Jedná se o verzi, která má 2048 8bitových MAC jednotek a bere se jako stabilní vydaná verze. Další větví je `master`, která je myšlena jako rozšíření výchozí `nvdlav1` větve o nové funkce s možností konfigurace. Poslední větví je `nv_small`, která obsahuje RTL modely pro „small" konfiguraci. Pro tuto práci je použita právě „small" konfigurace z této větve.

Konfigurace NVDLA pro generování je uložena ve složce `spec/defs/`. Konfigurace musí mít stejný název jako název projektu, který se vybere později při generování, a má přidanou koncovku `.spec`. Při této práci byl použit výchozí konfigurační soubor s názvem `nv_small.spec`, který je určený pro „small" konfiguraci NVDLA (detaily byly uvedeny v kapitole 2.2.4).

Generování RTL popisu je stručně popsané v dokumentaci k NVDLA [20]. Výchozí verze použitých nástrojů jsou popsané v souboru `Makefile`. Z experimentování s generací se dospělo k závěru, že generování není závislé na konkrétních verzích nástrojů a lze použít i jiné verze. Také není potřeba zadávat všechny uvedené nástroje. Minimálně to platí pro vygenerování RTL popisu NVDLA ve Verilogu, který je potřeba pro pozdější implementaci ve Vivado Design Suite.

Konfigurace cest k nástrojům se provádí pomocí příkazu `make`, který vytvoří konfigurační soubor `tree.make` potřebný k vygenerování RTL popisu. Při generování pro tuto práci byly zadány následující nástroje v dané verzi:

- cpp/gcc/g++ – 12.2.1 20230201
- perl – 5.36.0
- java – 20.0.1 2023-04-18

- python – 3.10.10
- clang – 15.0.7

Verze nástrojů nejsou zvoleny nijak konkrétně, jednalo se o aktuálně nainstalované verze v systému.

Po dokončení konfigurace byl vygenerován RTL popis příkazem:

```
$ ./tools/bin/tmake -build vmod
```

Po dokončení příkazu se vytvořila nová složka s názvem `outdir`, která obsahuje vygenerovaná data. RTL soubory se konkrétně nacházejí ve složce `out/nv_small/vmod`.

5.1.1 Generování pomocí Dockeru

Vzhledem k závislostem mezi různými knihovnami a potřebnou konfigurací před generováním, může být jednodušší použít pro generování RTL popisu připravený Docker kontejner `farhanaslam/nvdla`[1]. V kontejneru jsou již obsaženy všechny potřebné nástroje pro generování a je dokončena konfigurace. Po spuštění kontejneru tedy stačí spustit tmake pro vygenerování RTL popisu. Kontejner také navíc obsahuje nastavené prostředí pro emulaci NVDLA hardwaru.

5.2 Úprava RTL popisu pro FPGA

Jak již bylo zmíněno, NVDLA je původně navržen pro výrobu ASIC, tzn. že obsahuje popis všech potřebných komponent, včetně interních RAM. Při použití na FPGA jsou ovšem tyto popisy mapovány do LUT jednotek namísto do BRAM jednotek, čímž dochází ke zbytečně velkému vytížení LUT jednotek. Nejjednodušším způsobem, jak namapovat interní RAM do BRAM jednotek v FPGA, je odstranit složku `synth` s jejich popisem nacházející se v adresáři `out/nv_small/vmod/rams` [14].

5.3 Rozšíření RTL popisu o injekci poruch

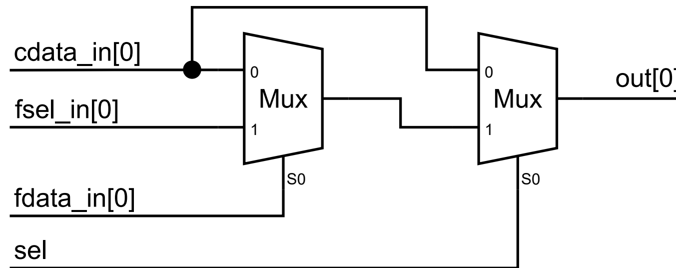
Jak bylo již zmíněno v kapitole 4.4.2 s návrhem, bylo rozhodnuto o tom, že se bude provádět injekce poruch do NVDLA pomocí úpravy RTL popisu. Injektovaná data se vkládají mezi násobičku a sčítačku v MAC jednotce. Modul, který obsahuje popis propojení násobiček a sčítaček se jmenuje `NV_NVDLA_CMAC_CORE_mac` a nachází se v souboru se stejným jménem ve složce `vmod/nvdla/cmac`. Do modulu vstupuje 64bitová (8*8) sběrnice s daty a 64bitová sběrnice s váhami. Dále se zde nachází osm 18bitových násobiček a sedm 19bitových sčítaček. Výstupem je výsledná 19bitová suma. Nachází se zde ještě signály pro validaci vstupních dat. Modul obsahuje i další signály, ale ty nejsou pro injekci poruch důležité. Po vynásobení váhy se vstupními daty ještě probíhá bitový součin (AND) s bitem pro validaci, rozšířeným na 18 bitů. Validované hodnoty jsou poté sečteny pomocí série sčítaček do výsledné sumy.

Aby bylo možné injektovat poruchy, byl vytvořen modul `fault_injection_mux` v souboru se stejným názvem. Modul potřebuje jako vstup 18bitovou hodnotu z násobičky `cdata_in` a jako výstup vrací upravenou hodnotu `out`, která se následně validuje a předává do série sčítaček. Pokud není modul aktivován, dochází k pouhému předání vstupní

[1] https://hub.docker.com/r/farhanaslam/nvdla

hodnoty na výstup bez úpravy. Modul dále potřebuje jako vstup 18bitovou sběrnici s injektovanými daty `fdata_in`, 18bitovou sběrnici s bitovou maskou pro injektovaná data `fsel_in` a 1bitovou hodnotu `sel`, která aktivuje modul.

Samotný modul poté obsahuje dva multiplexory pro každý bit vstupních dat (obrázek 5.1). První určuje, jestli se pro daný bit použije injektovaná hodnota, nebo původní výstup z násobičky. Pro výběr slouží bitová maska na injekci dat. Pomocí druhého multiplexoru se potom vybírá, zdali se bude aplikovat injekce dat z prvního multiplexoru, nebo původní hodnota z násobičky. Výběr výstupní hodnoty řídí bit, který určuje aktivaci modulu.



Obrázek 5.1: Schéma zapojení multiplexorů pro jeden bit v modulu injekce poruch.

V každé MAC jednotce se nachází osm těchto modulů. Jak bylo řečeno již v návrhu, vstupní sběrnice pro injekci jsou mezi všemi moduly sdílené. Akorát bit určující aktivaci je pro každý modul zvlášť. Všechny sběrnice jsou vyvedeny ven z NVDLA a vznikají tak ve výsledku 4 sběrnice pro ovládání injekce poruch. 18bitová sběrnice `fi_mux_fdata_in` s injektovanými daty, 18bitová sběrnice `fi_mux_fsel_in` s maskou injektovaných dat, 32bitová sběrnice `fi_mux_sel_a` s výběrem aktivovaných modulů z *partition ma* a 32bitová sběrnice `fi_mux_sel_b` s výběrem aktivovaných modulů z *partition mb*. Sběrnice pro výběr aktivovaných modulů nebyly při výstupu z *partition ma* a *partition mb* spojeny, protože budou v blokovém návrhu napojeny na IP core, který podporuje maximálně 32bitovou sběrnici.

5.4 Vytvoření projektu ve Vivado Design Suite

Projekt vytvořený ve Vivado Design Suite pro tuto práci je určen pro `Zynq UltraScale+ ZCU104 Evaluation Board`. Při vytváření byl přidán obsah složky `out/nv_small/vmod` s upraveným kódem pro podporu injekce poruch a constraints soubor pro použitou desku. Constraints soubor bylo nutné přidat z důvodu použití externího slotu pro SO-DIMM DDR4 paměť. Bez použití paměťového modulu je teoreticky možné constraints soubor vynechat, protože nejsou použity žádné IO piny FPGA čipu.

5.4.1 Verilog definice

Pro zprovoznění NVDLA na FPGA bylo potřeba nastavit určité Verilog definice (ve Vivado Design Suite pojmenované jako Defines ve Verilog Options dialogu). Definice jsou převzaté z komentáře[2] na GitHubu a obsahují například vypnutí tzv. clock gating a power gating, zkrácení cest a zlepšení časování.

`VLIB_BYPASS_POWER_CG`

[2] https://github.com/nvdla/hw/issues/110#issuecomment-744455880

```
NV_FPGA_FIFOGEN
FIFOGEN_MASTER_CLK_GATING_DISABLED
FPGA = 1
SYNTHESIS
DISABLE_TESTPOINTS
NV_SYNTHESIS
DESIGNWARE_NOEXIST = 1
RAM_DISABLE_POWER_GATING_FPGA
```

5.4.2 Wrapper pro nvdla_top

Jak bylo již popsáno v kapitole 2.2.4, NVDLA používá pro komunikaci s procesorem vlastní CSB rozhraní, které je potřeba převést na rozhraní podporované procesorem, v tomto případě AXI. Zdrojové kódy NVDLA obsahují modul, který převádí CSB na APB rozhraní, pro které je ve Vivado Design Suite dostupný IP core pro převod na AXI rozhraní.

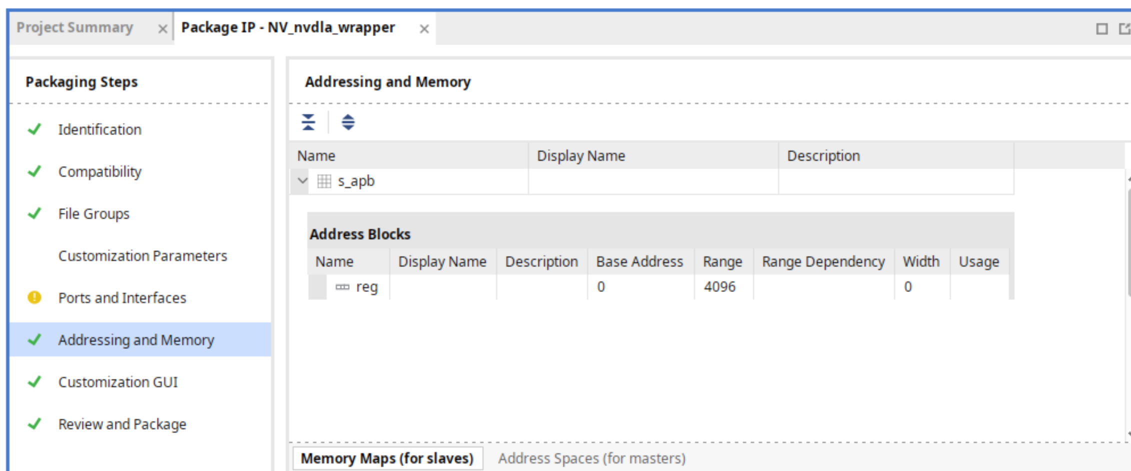
Bylo tedy potřeba použít wrapper pro `nvdla_top`, který bude obsahovat tento převodník. Tím se dosáhlo toho, že výsledný modul vystavuje APB rozhraní, se kterým se už potom dalo dále pracovat. Wrapper dále nastavuje některé vstupy `nvdla_top` modulu (např. `tmc2slcg_disable_clock_gating`), které nejsou potřeba z výsledného modulu vystavit ven, na konstanty. Wrapper byl převzat z blogového příspěvku [14] a rozšířen o sběrnice pro injekci poruch. Výsledný wrapper má následující porty:

- **core_clk** – Vstup hodinového signálu pro akcelerátor.
- **csb_clk** – Vstup hodinového signálu pro CSB rozhraní.
- **rstn** – Inverzní vstup sloužící k resetu akcelerátoru.
- **csb_rstn** – Inverzní vstup sloužící k resetu CSB převodníku na APB.
- **dla_intr** – Výstup vytvářející přerušení v procesoru.
- Sadu vstupů a výstupů sloužících pro master AXI rozhraní, v dokumentaci označované jako `DBBIF` a v IP balíčku jako `m_axi`.
- Sadu vstupů a výstupů sloužících pro slave APB rozhraní, (interně převedené na `CSB`) a v IP balíčku označované jako `s_apb`.
- **fi_mux_fdata_in** – 18bitová sběrnice pro injektovaná data.
- **fi_mux_fsel_in** – 18bitová sběrnice pro bitovou masku injektovaných dat.
- **fi_mux_sel_a** – 32bitová sběrnice pro výběr injektovaných násobiček *partition ma*.
- **fi_mux_sel_b** – 32bitová sběrnice pro výběr injektovaných násobiček *partition mb*.

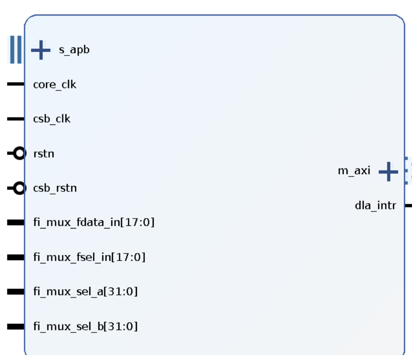
5.4.3 IP balíček s NVDLA

Aby bylo možné použít NVDLA v blokovém návrhu, bylo potřeba vytvořit vlastní IP balíček.

V portech a rozhraních bylo potřeba asociovat hodinový signál `core_clk` s rozhraním `m_axi`. Dále bylo potřeba vytvořit mapování paměti pro `s_apb` rozhraní s adresovým blokem `reg` o velikost 4096 (obrázek 5.2). Vzhled výsledného IP balíčku s NVDLA a jeho porty je na obrázku 5.3.



Obrázek 5.2: Mapování paměti v IP balíčku.



Obrázek 5.3: Výsledný IP balíček s NVDLA.

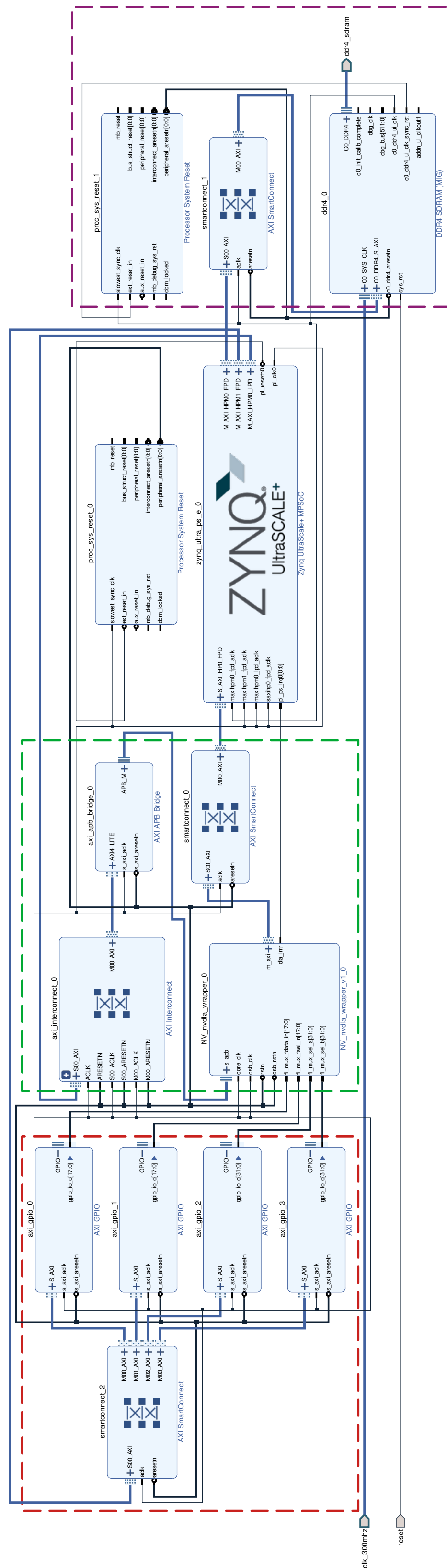
5.4.4 Blokový návrh

Po vytvoření IP balíčku s NVDLA bylo potřeba vytvořit blokový návrh, který spojí NVDLA s procesorem. Dále bylo potřeba napojit do procesoru sběrnice pro injekci poruch do NVDLA. Podstatnou částí bylo i propojit procesor s externí DDR4 SDRAM pamětí, aby ji bylo možné využívat v operačním systému. Schéma výsledného blokového návrhu je uvedeno v obrázku 5.4.

Středem celého blokového návrhu je IP core `Zynq UltraScale+ MPSoC` pro samotný procesor obsažený v FPGA a jsou k němu připojeny všechny ostatní komponenty.

Externí DDR4 paměť – fialová sekce

Aby bylo možné používat DDR4 SDRAM paměť připojenou do rozšiřujícího SO-DIMM slotu, návrh obsahuje IP core `DDR4 SDRAM (MIG)`, který slouží k těmto účelům. `DDR4 SDRAM (MIG)` je napojený přes `AXI SmartConnect` do `Zynq UltraScale+ MPSoC` na jednu z master AXI sběrnic, přesněji `M_AXI_HPM0_FPD`. Bloky sloužící k připojení DDR4 SDRAM paměti mají vlastní `Processor System Reset`. Nastavení bloku `DDR4 SDRAM (MIG)` mohlo zůstat ve výchozím nastavení, díky použití obdobné paměti jako doporučované výchozí paměti pro Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit.



Obrázek 5.4: Schéma výsledného blokového návrhu.

NVDLA – zelená sekce

Dále se v blokovém návrhu nachází NV_nvdla_wrapper_v1_0, který obsahuje samotný NVDLA akcelerátor. Jeho master AXI sběrnice m_axi je napojená do Zynq UltraScale+ MPSoC slave AXI sběrnice S_AXI_HP0_FPD přes AXI SmartConnect. Slave APB sběrnice je napojená do AXI APB Bridge, který má napojenou slave AXI4-Lite sběrnici do AXI Interconnect a až ten je následně připojený do Zynq UltraScale+ MPSoC master AXI sběrnice M_AXI_HPM0_LPD. V použité verzi Vivado Design Suite je možné nahradit AXI Interconnect za AXI SmartConnect. Jediný důvod k použití AXI Interconnect je ten, že v době návrhu nebylo jisté, jestli nebude potřeba použít starší verzi Vivado Design Suite kvůli kompatibilitě se starší verzí PetaLinux Tools. A starší verze AXI SmartConnect nejsou doporučené k používání s AXI4-Lite sběrnicí, protože neumožňují optimalizaci použité plochy [30].

Správa injekce poruch – červená sekce

Další významnou částí blokového návrhu jsou čtyři AXI GPIO, které jsou napojené na sběrnice v NV_nvdla_wrapper_v1_0, použité k injekci poruch. Všechny čtyři AXI GPIO jsou napojeny do společného AXI SmartConnect, který je následně zapojen do M_AXI_HPM1_FPD, master AXI sběrnice v Zynq UltraScale+ MPSoC. Takto je zajištěna možnost ovládání injekce poruch z procesoru. Všechny AXI GPIO jsou nastavené na pouze výstupní a jejich velikost je nastavena na stejnou velikost se sběrnicí, na kterou jsou napojeny.

AXI GPIO také využívají AXI4-Lite sběrnici, stejně jako AXI APB Bridge, ale jsou napojeny přes AXI SmartConnect, protože byly přidány v pozdější fázi, kdy byla potvrzena základní funkčnost návrhu i s novou verzí Vivado Design Suite. AXI Interconnect u spojení s AXI APB Bridge už ovšem nebyl nahrazen z důvodu zachování ověřeného návrhu, který byl akorát rozšířen o injekci poruch.

Časování

V časování Zynq UltraScale+ MPSoC byl nastaven hodinový signál PL0, použitý pro obvody v FPGA, na frekvenci 187,5 MHz, což byla nejvyšší možná frekvence, při které neselhalo časování v implementaci. Frekvence by mohla být ještě o něco vyšší, ale při použití výchozího zdroje časování IOPLL je další možná frekvence až 214 MHz, při které už selhává časování. V kapitole 7.3 jsou uvedeny podrobnosti ohledně maximální frekvence a úpravy zdrojových kódů NVDLA, pro její zvýšení.

Mapování adresového prostoru

Po vytvoření blokového návrhu bylo potřeba vytvořit mapování adres pro jednotlivé adresové prostory. Z velké části se podařilo softwaru vytvořit mapování automaticky a bylo akorát potřeba ověřit (a popřípadě opravit) rozsahy namapovaných oblastí. Je nutné dodržet mapování do správného adresového rozsahu daného AXI rozhraním, na které jsou komponenty připojeny. Výsledné mapování adres je zobrazeno na obrázku 5.5.

U /NV_nvdla_wrapper_0/m_axi bylo potřeba namapovat oblast do segmentu s RAM pamětí. Pro tento účel byla zvolena interní paměť o velikosti 2 GiB.

Velikost rozsahu paměti pro všechny AXI GPIO byla ponechána na výchozích 64 KiB, ale v případě potřeby může být snížena dle konkrétních potřeb bloků.

Diagram | Address Editor | Address Map

Assigned (9) | Unassigned (0) | Excluded (0) | Hide All

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
Network 0					
/NV_nvdla_wrapper_0					
/NV_nvdla_wrapper_0/m_axi (64 address bits : 16E)					
/zynq_ultra_ps_e_0/SAXIGP2	S_AXI_HP0_FPD	HP0_DDR_LOW	0x0000_0000_0000_0000	2G	0x0000_0000_7FFF_FFFF
/zynq_ultra_ps_e_0/SAXIGP2	S_AXI_HP0_FPD	HP0_QSPI	0x0000_0000_C000_0000	512M	0x0000_0000_DFFF_FFFF
/zynq_ultra_ps_e_0/SAXIGP2	S_AXI_HP0_FPD	HP0_LPS_OCM	0x0000_0000_FF00_0000	16M	0x0000_0000_FFFF_FFFF
Network 1					
/zynq_ultra_ps_e_0					
/zynq_ultra_ps_e_0/Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G] ,0x0080000000 [512M] ,0x00B0000000 [256M] ,					
/axi_gpio_0/S_AXI	S_AXI	Reg	0x00_B000_0000	64K	0x00_B000_FFFF
/axi_gpio_1/S_AXI	S_AXI	Reg	0x00_B001_0000	64K	0x00_B001_FFFF
/axi_gpio_2/S_AXI	S_AXI	Reg	0x00_B002_0000	64K	0x00_B002_FFFF
/axi_gpio_3/S_AXI	S_AXI	Reg	0x00_B003_0000	64K	0x00_B003_FFFF
/ddr4_0/C0_DDR4_MEMORY_MAP	C0_DDR4_S_AXI	C0_DDR4_ADDRESS_BLOCK	0x04_0000_0000	4G	0x04_FFFF_FFFF
/NV_nvdla_wrapper_0/s_apb	s_apb	reg	0x00_8000_0000	64K	0x00_8000_FFFF

Obrázek 5.5: Editor adres s výsledným mapováním.

U `/ddr4_0/C0_DDR4_MEMORY_MAP` bylo nutné ověřit velikost rozsahu s reálnou velikostí připojené externí DDR4 SDRAM. Pokud by velikost neseděla, znamenalo by to nejspíše chybu v nastavení `DDR4 SDRAM (MIG)` v blokovém návrhu.

Pro `/NV_nvdla_wrapper_0/s_apb` bylo potřeba dodržet stejnou velikost rozsahu, jako byla uvedena při vytváření IP balíčku, tzn. 64 KiB. Tímto byly dodrženy všechny potřebné náležitosti na mapování adres.

Na závěr ještě bylo potřeba vytvořit wrapper pro blokový návrh, který zapouzdřil celé řešení a byl nastaven jako hlavní soubor v návrhu kódu pro FPGA.

5.4.5 Syntéza, implementace a export bitstreamu

Poslední fází implementace hardwaru bylo spuštění syntézy a implementace návrhu. Při spuštění na procesoru AMD Ryzen 7 7700, s použitím všech 16 logických jader, zabrala syntéza zhruba 6 minut a implementace 15 minut a bylo potřeba alespoň 23 GiB volné paměti před spuštěním. Poté byl vygenerován bitstream a starý formát popisu hardwaru, který se v novějších verzí Vivado Design Suite generuje příkazem:

```
$ write_hwdef -force -file ./nvdla_zcu104.hdf
```

Kapitola 6

Implementace softwaru

Pro přípravu softwaru byl použit operační systém Ubuntu 18.04.6 LTS. Pro vytvoření Linuxového kernelu byl použit software PetaLinux Tools 2019.1. Verze PetaLinux Tools byla vybrána kvůli verzi Linuxového kernelu 4.19 [31]. Jedná se o nejnovější kernel, se kterým lze (po drobné úpravě) používat ovladač KMD pro NVDLA [14]. Daná verze Ubuntu byla zvolena kvůli oficiální podpoře PetaLinux Tools 2019.1 [29]. Zdrojové kódy softwaru k NVDLA byly použity z oficiálního repozitáře publikovaného na webu GitHub [19].

6.1 PetaLinux

Aby bylo možné spustit operační systém na `Zynq UltraScale+ MPSoC`, je potřeba použít upravený Linux kernel s podporou AMD FPGA. Tento Linux kernel je možné vytvořit pomocí nástroje PetaLinux Tools, který slouží k úpravě a vytvoření distribuce Linuxu zvané PetaLinux. Cílem tedy je vytvořit Linuxový kernel pomocí PetaLinux Tools s potřebnými ovladači, včetně ovladače NVDLA KMD, a se Device tree, který obsahuje veškerý přidaný hardware (NVDLA, RAM paměť a AXI GPIO). Dále byl nahrazený kořenový souborový systém z jiné distribuce, aby došlo k přidání správce balíčků a bylo možné vyvíjet software přímo na zařízení. Poslední krok implementace bylo zprovoznění upravené verze frameworku Tengine s podporou injekce poruch.

6.1.1 Vytvoření projektu

Na začátku bylo potřeba vytvořit PetaLinux projekt. Při vytváření je potřeba specifikovat šablonu (`--template`), která je určená pro daný typ procesoru, v tomto případě `zynqMP`. Dále je potřeba uvést název projektu (`--name`), který může být libovolný. Ukázka použitého příkazu:

```
$ petalinux-create -t project --template zynqMP --name smalldla_zcu104
```

Po vytvoření projektu je nutné importovat soubor s popisem hardwaru vygenerovaný v softwaru Vivado Design Suite. Při importování dojde k nastavení projektu podle specifikace hardwaru. Nastavení může zahrnovat FSBL konfiguraci, U-Boot nastavení, nastavení Linux kernelu a konfiguraci zařízení v Linux device tree. Příkaz pro konfiguraci se spouští ve složce s projektem a jako parametr se uvádí cesta ke složce s popisem hardwaru. Ukázka příkazu:

```
$ petalinux-config --get-hw-description=~/nvdla_hw/
```

kde složka `~/nvdla_hw/` obsahuje vygenerovaný HDF soubor z kapitoly 5.4.5.

Po načtení popisu hardwaru dojde k zobrazení menu s konfigurací systému. Zde bylo potřeba nastavit v sekci `DTG Settings` atribut `MACHINE_NAME` na `zcu104-revc`, aby došlo ke specifikaci konkrétního hardwaru (Zynq UltraScale+ MPSoC ZCU104 Evaluation Kit). Dále bylo změněno nastavení kořenového souborového systému na microSD kartu pomocí atributu `Root filesystem type` v sekci `Image Packaging Configuration`.

6.1.2 Příprava Linux kernelu

Dalším krokem po konfiguraci systému je konfigurace Linuxového kernelu v menu, které se spouští příkazem:

```
$ petalinux-config -c kernel
```

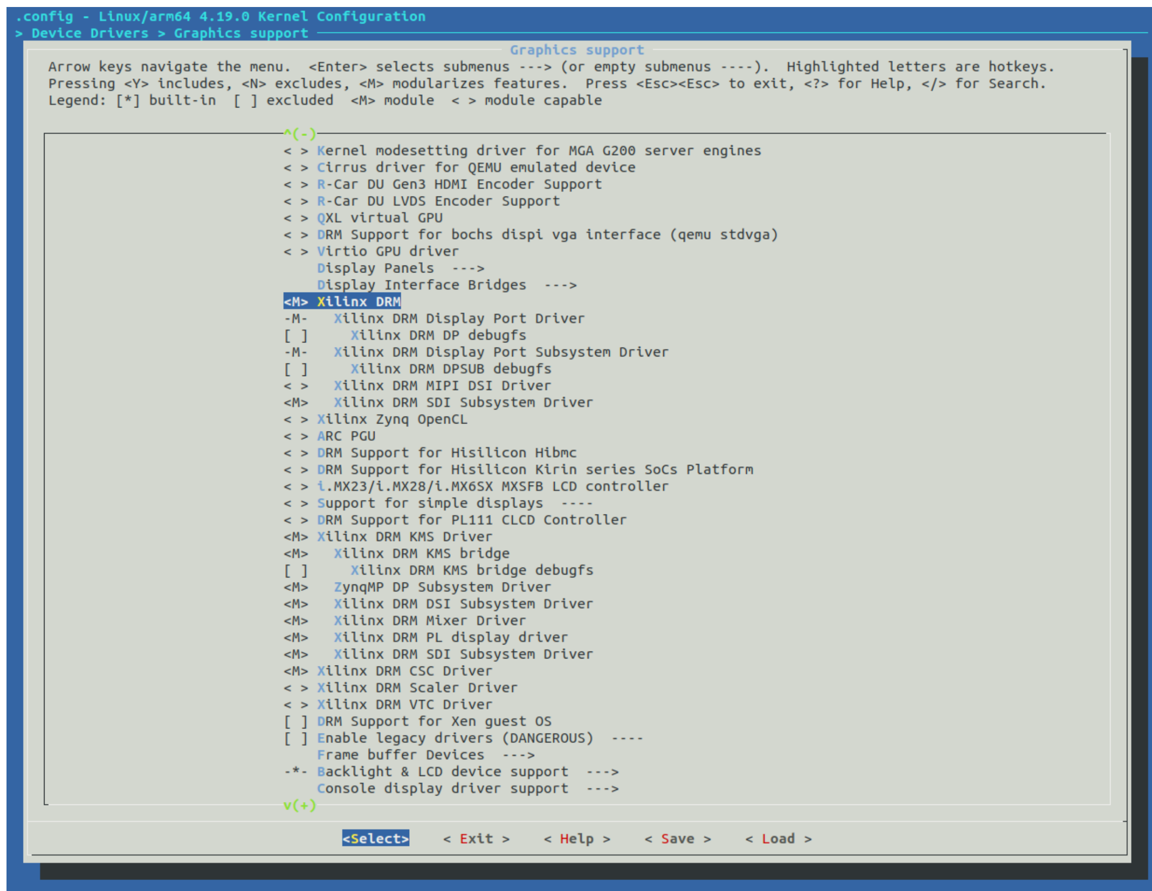
V konfiguraci byly provedeny následující změny:

1. **Použití microSD karty pro kořenový souborový systém:** Výsledný kořenový souborový systém pro Linux má být na microSD kartě, aby jej bylo možné upravovat. Ve výchozím stavu je ovšem kořenový souborový systém uložen v souboru (initramfs), který se po spuštění nahraje do RAM disku. V RAM disku je sice možné provádět změny, ale všechny úpravy se po vypnutí systému ztratí. Za normálních okolností slouží RAM disk pouze k načtení minimálního prostředí s potřebnými nástroji a skripty, které se postarají o připojení hlavního souborového systému, v klasickém počítači např. SSD disk. Chování v PetaLinuxu bylo ovšem jiné, došlo k načtení initramfs, který obsahoval celý souborový systém vytvořený při generování a nedošlo již k připojení oddílu na microSD kartě jako kořenového adresáře. Nejjednodušším řešením bylo vypnout podporu RAM disku v sekci `General setup` pomocí atributu `Initial RAM filesystem and RAM disk (initramfs/initrd) support`.

2. **Kompilace ovladačů DRM jako modulů:** Ve výchozím stavu jsou ovladače DRM kompilované jako součást Linux kernelu (označené hvězdičkou). S tímto výchozím nastavením se ovšem nepodařilo načíst za běhu systému modul s ovladačem NVDLA KMD. Načtení modulu vždy skončilo s chybou „failed to register drm device“. Po nastavení kompilace ovladačů DRM jako modulů (označení písmenem M) již došlo ke správnému načtení modulu s ovladačem NVDLA KMD. Nastavení ovladačů DRM se nachází v sekci `Device Drivers` a následně v sekci `Graphics support`. Konkrétně použité nastavení jednotlivých ovladačů DRM je na obrázku 6.1.

3. **Kompilace ovladače PHY s kernelem:** Při kompilaci ovladačů DRM jako modulu se nepodařilo zkompilovat ovladač Xilinx ZynqMP PHY jako modul, což je výchozí nastavení. Nepodařilo se zjistit, z jakého důvodu je tato kombinace problémová. Při kompilaci ovladače Xilinx ZynqMP PHY jako součást kernelu se problém vyřešil a kompilace proběhla úspěšně. Nastavení ovladače se nachází v sekci `Device Drivers` a následně v sekci `PHY Subsystem` pod názvem `Xilinx ZynqMP PHY driver`.

4. **Kompilace ovladače Userspace I/O:** Aby bylo možné ovládat z Linuxu `AXI GPIO` moduly, použité v FPGA pro injekci poruch, bylo potřeba pro ně použít nějaký ovladač. Ideální řešení by bylo napsat vlastní ovladač, který by interně spojil všechny `AXI GPIO` moduly a aplikacím poskytl jednotné rozhraní pro injekci poruch. Rozhraní ovladače by mohlo vytvářet vyšší úroveň abstrakce (jako např. poskytnout přístupy pro injekci jednotlivých násobiček, které by byly dále rozděleny podle MAC



Obrázek 6.1: Konfigurace ovladačů DRM v nastavení Linux kernelu.

jednotek, ve kterých se nacházejí), místo několika sběrnic, které nejsou nijak popsané, a tím zpřehlednit injekci poruch. Psaní vlastního ovladače by ale bylo časově náročné a zvýšilo by to úroveň komplexnosti řešení. Z tohoto důvodu bylo zvoleno mnohem jednodušší řešení v podobě již existujícího I/O ovladače do uživatelského rozhraní. Pro kompilaci ovladače Userspace I/O je potřeba nastavit dva ovladače `Userspace I/O platform driver with generic IRQ handling` a `Userspace platform driver with generic irq and dynamic memory`, nacházející se v sekci `Device Drivers` a následně v sekci `Userspace I/O drivers`, na kompilované jako součást Linux kernelu. Ovladač za běhu vytvoří v adresáři `/dev` soubory `uio`, které budou napojeny na jednotlivé hardwarové komponenty podle specifikace v Device tree.

6.1.3 Příprava modulu s ovladačem NVDLA KMD

Aby byl ovladač NVDLA KMD zkompilován při kompilaci PetaLinuxu, bylo potřeba pro něj vytvořit PetaLinux modul. Při vytváření modulu je potřeba specifikovat název (`--name`) modulu a povolit modul (`--enable`). Ukázka použitého příkazu:

```
$ petalinux-create -t modules --name opendla --enable
```

Moduly se vytváří ve složce `project-spec/meta-user/recipes-modules/<název>`.

Zdrojové soubory ovladače KMD byly nakopírovány do `files` složky, nacházející se v souborech pro opendla modul. Zároveň byla při kopírování zrušena adresářová struktura zdrojových kódů, aby došlo ke zjednodušení Makefilu a `.bb` souboru.

Dále bylo potřeba upravit `nvdla_gem.c` (původně v adresáři `port/linux`):

1. **Adress pro NVDLA:** Při volání funkce `dma_declare_coherent_memory` jsou adresy a rozsahy paměti rezervované pro NVDLA akcelerátor uvedeny jako magické hodnoty a nejsou čteny z Device tree. Bylo tedy nutné tyto adresy upravit. Pro NVDLA akcelerátor byla později v Device tree vyhrazena spodní polovina (1 GiB) interní RAM paměti. Úprava kódu tedy vycházela ze zamýšlené úpravy Device tree.

2. **Kompatibilita s Linuxem 4.19:** Funkce `dma_declare_coherent_memory`[1] byla ve verzi Linuxového kernelu 4.19 upravena vůči původní verzi 4.13.3. Bylo tedy potřeba odstranit zastaralý příznak `DMA_MEMORY_MAP` a upravit podmínku, která kontrolovala úspěšné vykonání funkce.

V `opendla.h` bylo potřeba definovat, že se používá `nv_small` konfiguraci NVDLA pomocí přidání makra `#define DLA_2_CONFIG`.

Na závěr se do `Makefile` souboru přidaly všechny objekty generované z kódu a upravil se `.bb`, aby obsahoval všechny přidané soubory a upravený `COPYING` soubor s licencí.

6.1.4 Úprava Device tree

Do Device tree bylo potřeba dopsat přidané komponenty z implementace hardwaru (kapitola 5). Adresové prostory jednotlivých komponent byly použity z mapování adres v blokovém návrhu (kapitola 5.4.4). Jediná změna byla u mapování rezervované části paměti pro NVDLA akcelerátor. Tomu byl v blokovém návrh ponechán rozsah adres na celou interní paměť, aby bylo možné při implementaci softwaru vybrat požadovanou velikost rezervované paměti. NVDLA akcelerátoru by mělo stačit vyhradit 256 MiB paměti [14]. V této práci ale bylo pro NVDLA raději vyhrazeno místo v paměti o velikosti 1 GiB. Díky externí paměti nebylo potřeba šetřit kapacitu pro Linux a mohla tak být zarezervována větší paměť, aby se předešlo možným problémům s nedostatkem paměti pro akcelerátor.

6.1.5 Sestavení a zabalení projektu

Po přidání a nakonfigurování všeho potřebného bylo možné projekt sestavit a následně zabalit. Příkaz pro sestavení:

```
$ petalinux-build
```

Při zabalení projektu bylo potřeba specifikovat cestu k bitstreamu pomocí `--fpga`. Příkaz pro zabalení projektu:

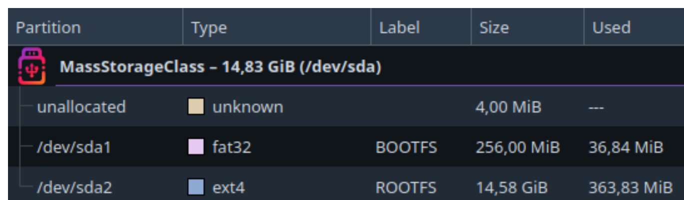
```
$ petalinux-package --boot --fsbl images/linux/zynqmp_fsbl.elf \
--fpga ./nvdla_zcu104.bit --u-boot
```

Všechny sestavené soubory se nachází ve složce `images/linux`.

[1] https://www.kernel.org/doc/html/v4.19/driver-api/infrastructure.html

6.2 Příprava microSD karty

Při přípravě microSD karty bylo potřeba vytvořit dva oddíly. První oddíl souborového systému FAT32 o velikosti 256 MiB a se 4 MiB prázdného místa před oddílem (od začátku microSD karty). Velikost prvního oddílu se může lišit, ale musí být alespoň 60 MiB [29]. Popisek oddílu je `BOOTFS`, jelikož se zde budou nacházet soubory potřebné pro bootování zařízení. Druhý oddíl souborového systému Ext4, zabírající zbytek celé microSD karty. Druhý oddíl má popisek `ROOTFS` a je určený pro kořenový souborový systém. Výsledné rozdělení microSD karty je na obrázku 6.2.



Partition	Type	Label	Size	Used
MassStorageClass – 14,83 GiB (/dev/sda)				
unallocated	unknown		4,00 MiB	---
/dev/sda1	fat32	BOOTFS	256,00 MiB	36,84 MiB
/dev/sda2	ext4	ROOTFS	14,58 GiB	363,83 MiB

Obrázek 6.2: Vytvořené oddíly na microSD kartě.

Na oddíl `BOOTFS` byly nahrané soubory `BOOT.BIN` a `image.ub`, ze složky se sestavenými soubory PetaLinuxu. Soubor `BOOT.BIN` je bootovací obraz, který obsahuje first stage bootloader (FSBL), FPGA bitstream a U-Boot. Druhý soubor `image.ub` obsahuje Linux kernel a Device Tree Blob, zkompilovaný Device tree.

6.2.1 Alternativní kořenový souborový systém

Hlavní myšlenka PetaLinux Tools je vytvořit PetaLinux na míru, se vším potřebným (ovladače, aplikace, které mají na zařízení běžet, atd.), který je možný spustit na vestavěném zařízení bez dalších úprav. Proto PetaLinux neobsahuje žádný správce balíčků, který by umožňoval instalovat dodatečný software za běhu systému. Veškerý software je potřeba kompilovat pomocí křížového překladače (cross compiler) nebo je možné software kompilovat přímo na zařízení, při sestavení PetaLinuxu s překladačem. To ale zahrnuje i kompilaci všech potřebných balíčků pro překlad a následných závislostí, což výrazně zvyšuje náročnost. Ani jedno řešení není příliš vhodné pro vývoj a testování na samotném vestavěném zařízení.

Řešením je použít kořenový souborový systém z jiné distribuce Linuxu, která obsahuje správce balíčků, jako například Ubuntu nebo Debian. Linuxový kernel bude zachován z PetaLinuxu, protože je upravený pro konkrétní hardware a kernel distribuovaný s jinou Linuxovou distribucí by nefungoval. Je potřeba si ověřit, že je vybraná verze dané distribuce kompatibilní s verzí kernelu z PetaLinuxu, tzn. 4.19. Pro tuto práci byl použit souborový systém z distribuce Debian ve verzi 10.13. Verze byla vybrána na základě toho, že používá stejnou verzi kernelu.

Souborový systém pro Debian byl použit ze starého archivu Linuxových obrazů od eewiki[2]. Jedná se o minimální instalaci Debian pro architekturu ARM64.

Aplikace obrazu Debianu byla provedena jednoduchým rozbalením archivu na `ROOTFS` oddíl microSD karty. Tímto krokem se ovšem přišlo o zkompilovaný modul s ovladačem NVDLA KMD a všechny další moduly, které se nachází v PetaLinux souborovém systému. Řešením bylo použít moduly z archivu `rootfs.tar.gz`, který obsahuje PetaLinux

[2] `https://rcn-ee.com/rootfs/eewiki/minfs/debian-10.13-minimal-arm64-2022-12-20.tar.xz`

souborový systém. V archivu bylo potřeba najít složku se zkompilovanými moduly, tzn. `/lib/modules/4.19.0-xilinx-v2019.1/` a nakopírovat ji do stejného adresáře na oddílu `ROOTFS` na microSD kartě.

Tímto byla náhrada kořenového souborového systému dokončena a microSD kartu bylo možné vložit do Zynq UltraScale+ MPSoC ZCU104 Evaluation Kitu a nabootovat Debian s upraveným kernelem.

6.3 NVDLA UMD

NVDLA UMD slouží ke komunikaci s ovladačem NVDLA KMD a ke kompilaci neuronových sítí na NVDLA Loadable. Je potřeba mít NVDLA UMD zkompilovaný specifickým způsobem, aby ho bylo možné použít s frameworkem Tengine. Jednou z možností, jak získat NVDLA UMD, je stáhnout si již zkompilované binární soubory[3]. Tím se ovšem ztratí možnost úpravy zdrojových kódů a hlavně kompilátor neuronových sítí je dostupný pouze pro architekturu x86. Bylo tedy potřeba zkompilovat NVDLA UMD ze zdrojových kódů. Kompilace probíhala přímo na zařízení, aby se vyhnulo nepříjemnostem spojeným s křížovou kompilací na jiné architektuře.

Před kompilací runtime prostředí NVDLA UMD bylo potřeba zkompilovat knihovnu jpeg. Runtime prostředí používá knihovnu jpeg verze 6. Po úspěšné kompilaci knihovny jpeg bylo ještě potřeba upravit Makefile pro runtime prostředí kvůli použití ve frameworku Tengine. Makefile obsahuje příznak pro C++ kompilátor o nepoužívání RTTI (Run-Time Type Information), což je ale potřeba pro kompilaci s frameworkem Tengine. RTTI je technika používaná pro informace o datových typech za běhu. Úprava tedy spočívala v nahrazení příznaku na vynucení použití RTTI. Poté již bylo možné zkompilovat runtime prostředí.

Druhou částí NVDLA UMD je kompilátor neuronových sítí. Před kompilací bylo opět potřeba zkompilovat používanou knihovnu, a to přesně Protocol Buffers (protobuf). Kompilace knihovny proběhla ze zdrojových kódů knihovny přiložených ke zdrojovým kódům kompilátoru. Použitá verze knihovny protobuf tedy byla 2.6. Po úspěšné kompilaci bylo opět potřeba upravit Makefile, stejně jako v případě runtime prostředí. Poté již mohlo dojít ke kompilaci kompilátoru.

6.4 Tengine

Zdrojové kódy pro Tengine byly získány z oficiálního repozitáře na GitHubu[4]. Pro kompilaci frameworku Tengine je potřeba mít knihovnu OpenCV. Ze zdrojových kódů bylo zjištěno, že je potřeba verze 4.2. Knihovna tedy byla zkompilována a nainstalována a poté se přešlo ke konfiguraci a kompilaci frameworku Tengine.

Pro zprovoznění podpory NVDLA bylo potřeba nakopírovat hlavičkové soubory ze zdrojových kódů NVDLA UMD do zdrojových kódů frameworku Tengine. Dále bylo potřeba nakopírovat i knihovny vzniklé při kompilaci NVDLA UMD (runtime a kompilátor) a zkompilovanou knihovnu protobuf. Konfigurace frameworku Tengine probíhá pomocí aplikace cmake. Při konfiguraci se musela povolit podpora NVDLA pomocí použití příznaku `TENGINE_ENABLE_OPENDLA=ON` a poté již bylo možné kompilovat jednotlivé ukázkové příklady pro NVDLA. Tím byla kompilace frameworku Tengine dokončena.

[3] https://github.com/nvdla/sw/tree/master/prebuilt/arm64-linux
[4] https://github.com/OAID/Tengine

6.5 Aplikace pro injekci poruch

Aby bylo možné provádět injekci poruch automatizovaně, bylo ještě potřeba vytvořit aplikaci. Jako základ byla použita ukázka z frameworku Tengine pro inferenci neuronových sítí na klasifikaci obrázků, `tm_classification_opendla`. Ukázková aplikace je napsaná v programovacím jazyce C. V ukázkové aplikaci byly provedeny následující úpravy:

- **Vyhodnocení přesnosti pro dataset.** Základní vyhodnocování funguje pouze pro jeden obrázek. Bylo tedy potřeba vyhodnocování rozšířit pro celý testovací dataset. Aplikace byla upravena tak, aby připravila neuronovou síť a poté postupně vyhodnocovala všechny obrázky v datasetu. Po vyhodnocení jednoho obrázku dojde k získání nejvíce pravděpodobné klasifikace a porovnání s reálnou klasifikací obrázku. Tímto způsobem dojde k vyhodnocení celkové přesnosti nad datasetem.

- **Přidáno nastavení injektovaných poruch.** Ovladač Userspace I/O vytvořil čtyři soubory `/dev/uio`, které jsou napojeny na jednotlivé `AXI GPIO` a pomocí kterých se nastavují injektované poruchy do NVDLA. Aplikace tedy byla rozšířena o namapování těchto souborů do svého paměťového prostoru a byl přidán význam k jednotlivým souborům pomocí struktur.

- **Rozšíření o načítání injektovaných dat z CSV souboru.** Aby bylo možné testování automatizovat, byla aplikace rozšířena o čtení zjednodušeného CSV souboru s injektovanými daty. Na každém řádku se nachází data pro jednu injekci poruch. Data obsahují injektované hodnoty, masku pro injekci jednotlivých bitů, výběr násobiček v *partition ma* a výběr násobiček v *partition mb* pro injekci. Na každém řádku jsou tedy čtyři sloupce s daty a další sloupce jsou ignorovány. Aplikace postupně zpracovává data od prvního řádku až do konce.

- **Rozšíření o ukládání výsledků do CSV souboru.** Do aplikace bylo přidáno ukládání výsledků po vyhodnocení přesnosti do zjednodušeného CSV souboru. Výstupní CSV soubor má velmi podobný formát jako vstupní, aby bylo možné použít výstupní soubor v případě potřeby jako vstup. První čtyři sloupce jsou zachovány stejné a na konec jsou přidané další dva sloupce. První přidaný sloupec slouží pro ukládání počtu úspěšně klasifikovaných obrázků a druhý sloupec pro počet špatně klasifikovaných. Na poslední řádek je přidán doplňující zápis. Ten obsahuje z důvodu kompatibility se vstupním CSV souborem hodnoty pro spuštění bez injekce poruch a až poté čtyři přidané sloupce. První přidaný sloupec obsahuje čas, jak dlouho trvalo frameworku Tengine připravit neuronovou síť pro spuštění v milisekundách. Druhý sloupec obsahuje součet všech časů inference jednotlivých obrázků na samotném NVDLA. Ve třetím sloupci je uložen maximální čas inference obrázku a ve čtvrtém minimální.

S těmito úpravami již bylo možné spouštět pomocí aplikace jednotlivé testovací sady použité při provádění experimentů, a výstupní formát následně strojově zpracovat.

Kapitola 7

Experimenty

Tato kapitola obsahuje popis základních vlastností implementovaného NVDLA akcelerátoru, dále jednotlivé experimenty s injekcí poruch do NVDLA a na závěr malé rozšíření. Jak již bylo zmíněno v kapitole 4.7, pro testování odolnosti proti poruchám byla použita neuronová síť ResNet-18 a testovací dataset CIFAR-10.

7.1 Vyhodnocení architektury a parametrů

Před začátkem samotných experimentů s injekcí poruch bylo potřeba zjistit základní vlastnosti vytvořeného akcelerátoru. Byla zjištěna režie vytvořená invazivní rozšíření pro injekci poruch, čas evaluace vybrané neuronové sítě a přesnost bez poruch. Tím zároveň došlo k ověření funkčnosti celého akcelerátoru a implementovaného řešení před začátkem vkládání poruch.

7.1.1 Režie na FPGA

Po implementaci verze bez injekce a verze s injekcí poruch bylo možné zjistit, jak velké vytížení navíc způsobila úprava pro injekci poruch. Údaje o využití jednotlivých částí FPGA, po úspěšně dokončené implementaci (a předcházející syntéze), jsou uvedeny v tabulce 7.1. V prvním sloupci tabulky jsou vyjmenovány jednotlivé části FPGA. Druhý sloupec uvádí dostupný počet jednotlivých částí FPGA. Třetí sloupec obsahuje počet využitých částí bez jakékoliv úpravy NVDLA akcelerátoru. Čtvrtý sloupec již obsahuje hodnoty po rozšíření NVDLA o injekci poruch a nastavení sběrnic pro injekci na konstantní hodnoty (nuly). Pátý sloupec obsahuje vytížení po napojení sběrnic pro injekci do procesoru, aby bylo možné nastavovat injektované poruchy.

Z údajů je vidět, že při porovnání s injekcí poruch konstantních hodnot nedošlo k výraznému zvýšení využití FPGA. Zvedl se počet využitých LUT jednotek o 18 a dokonce došlo k poklesu využitých FF jednotek o 15.

Při porovnání s nastavitelnou injekcí poruch došlo k již vyššímu využití. Bylo potřeba přidat do implementace moduly obstarávající komunikaci s procesorem a následné nastavení injektovaných poruch, takže šlo očekávat větší nárůst. Konkrétně se zvedl počet využitých LUT jednotek o 1643 (0,71 %), LUTRAM o 1, FF o 1433 (0,31 %) a došlo k poklesu využití BUFG jednotek o 1.

Část	Dostupné	Bez poruch	Kon-stantní porucha	Nastavi-telná porucha
LUT	230400	94438	94456	96081
LUTRAM	101760	5112	5112	5113
FF	460800	104732	104717	106150
BRAM	312	91.50	91.50	91.50
DSP	1728	35	35	35
IO	360	119	119	119
BUFG	544	7	7	6
MMCM	8	1	1	1
PLL	16	3	3	3

Tabulka 7.1: Porovnání využití jednotlivých částí FPGA bez a s injekcí poruch.

7.1.2 Čas evaluace

Jeden ze zajímavých údajů při evaluaci neuronových sítí je doba trvání. Tento čas se navíc hodil pro odhad, jak dlouho budou trvat další experimenty. Čas byl měřen při evaluaci neuronové sítě ResNet-18 na testovací části datasetu CIFAR-10, která obsahuje 10 000 obrázků.

Pro porovnání byl použit čas evaluace na procesoru v FPGA, který obsahuje čtyři jádra ARM Cortex-A53 o maximální frekvenci 1,2 GHz. Kdyby se měla neuronová síť evaluovat bez akcelerátoru, muselo by k výpočtu docházet právě na procesoru v FPGA. Tengine umožňuje spustit evaluaci na více vláknech. Byly tedy změřeny časy pro jedno vlákno a čtyři vlákna. Z naměřených dat (tabulka 7.2) je vidět, že použití akcelerátoru je nejrychlejší možné řešení evaluace. Za zmínku také stojí to, že frekvence akcelerátoru byla 187,5 MHz, což je o 1 GHz méně než jádra procesoru. A pro příklad Jetson AGX Xavier má dva NVDLA „full“ akcelerátory s maximální frekvencí 1,4 GHz [21].

Zařízení	Vlákna	Aritmetika	Vyhodnocení datasetu (s)	Vyhodnocení obrázku (ms)
ARM Cortex-A53	1	int8	226,8	22,680
ARM Cortex-A53	4	int8	141,2	14,120
NVDLA „small“	–	int8	45,8	4,585
ARM Cortex-A53	1	float	133,6	13,356
ARM Cortex-A53	4	float	85,4	8,539

Tabulka 7.2: Čas evaluace neuronové sítě ResNet-18 na testovací části datasetu CIFAR-10 (10 000 obrázků).

Při porovnání času ze práce na frameworku SAFFIRA [27] je vidět výrazný nárůst v rychlosti inference. V práci je uvedeno, že 500 inferencí dvou konvolučních vrstev trvalo 10 minut. V práci nejsou uvedeny údaje o velikosti konvolučních vrstev, takže lze provést pouze hrubý odhad. Inference jedné konvoluční vrstvy pomocí frameworku SAFFIRA trvala 600 ms. Neuronová síť použitá v této práci obsahuje 19 konvolučních vrstev. V případě, kdy se zanedbá existence ostatních vrstev, trvalo vyhodnocení jedné konvoluční vrstvy 0,24 ms. To by znamenalo, že mnou představené řešení je 2 500krát rychlejší. Odhadovaná spotřeba

FPGA ve Vivado Design Suite byla 5,9 W. Při inferenci pomocí akcelerátoru byla změřena spotřeba celého vestavěného systému kolem 15,5 W.

Pro zajímavost a otestování schopností frameworku Tengine byly změřeny i časy na procesoru AMD Ryzen 7 7700 a na grafické kartě GeForce RTX 4070, viz tabulka 7.3. Při evaluaci na AMD Ryzen 7 7700 mohlo dojít k použití více vláken. Bylo ovšem zjištěno, že použití větší počet vláken celou evaluaci akorát zpomaluje a časy jsou horší než při použití jednoho vlákna. Při použití 16 vláken na celočíselné aritmetice (int8) byl čas evaluace asi 1,3krát delší a na aritmetice s plovoucí desetinnou čárkou došlo dokonce k nárůstu času 311krát a navíc bylo vytíženo jenom 8 logických jader procesoru namísto očekávaných 16 logických jader procesoru.

Při evaluaci na grafické kartě GeForce RTX 4070 pomocí CUDA došlo k vytížení GPU na pouhých 33 %, ale bylo využito celých 12 GB VRAM, kterými grafická karta disponuje. Evaluace navíc vracela zvláštní výsledky, které byly vždy jiné. Je potřeba tedy časy v tabulce 7.3 brát s velkou rezervou. Lze tedy spíše usoudit, že evaluace pomocí grafické karty ve frameworku Tengine spíše nefunguje, minimálně ne na grafické kartě GeForce RTX 4070.

Zařízení	**Vlákna**	**Aritmetika**	**Vyhodnocení datasetu (s)**	**Vyhodnocení obrázku (ms)**
AMD Ryzen 7 7700	1	int8	115,7	11,574
AMD Ryzen 7 7700	4	int8	56,7	5,646
AMD Ryzen 7 7700	1	float	7,9	0,793
AMD Ryzen 7 7700	4	float	4,5	0,446
GeForce RTX 4070	–	float	5,3	0,532

Tabulka 7.3: Čas evaluace neuronové sítě ResNet-18 na testovací části datasetu CIFAR-10 (10 000 obrázků).

7.1.3 Zmenšení datasetu a přesnost neuronové sítě

Při experimentech s injekcí poruch bylo potřeba spustit velké množství vyhodnocení neuronové sítě. Provádění všech experimentů na plném datasetu CIFAR-10 by trvalo několik desítek hodin. Dataset obsahuje 10 000 obrázků určených pro testování a vyhodnocení přesnosti neuronové sítě. Vyhodnocení této části datasetu pomocí neuronové sítě ResNet-18 na NVDLA trvalo zhruba 46 sekund (tabulka 7.2). Proto vznikla motivace zmenšit dataset, a tím urychlit experimenty. Po možnosti zmenšení datasetu na 10 % původní velikosti by došlo ke zkrácení času, potřebného na vyhodnocení experimentů, na jednotky hodin. Zmenšení datasetu ovšem může mít výrazný dopad na přesnost výsledků, proto bylo potřeba prvně na omezeném vzorku dat zjistit vliv zmenšení datasetu.

V době úvahy o zmenšení datasetu ještě nebyl známý přesný počet injekcí poruch, který bude potřeba provést. Ale v době psaní této práce byl již počet známý. Pro všechny experimenty bylo potřeba provést kolem 15 000 injekcí poruch, což by na plném datasetu trvalo přibližně 190 hodin. Veškeré injekce navíc bylo potřeba opakovat, kvůli nalezení chyby v testovacím datasetu, který snižoval přesnost o zhruba 12 %. Dále došlo ke spuštění několika dalších injekcí poruch ve fázi testování. V případě použití plného datasetu by to tedy reálně znamenalo potřebu více než 400 hodin na získání použitelných dat pro experimentální část práce. Po zmenšení datasetu bylo potřeba pouze zhruba 50 hodin (některé části byly stále vyhodnoceny s plným datasetem, proto nedošlo ke zmenšení času na desetinu).

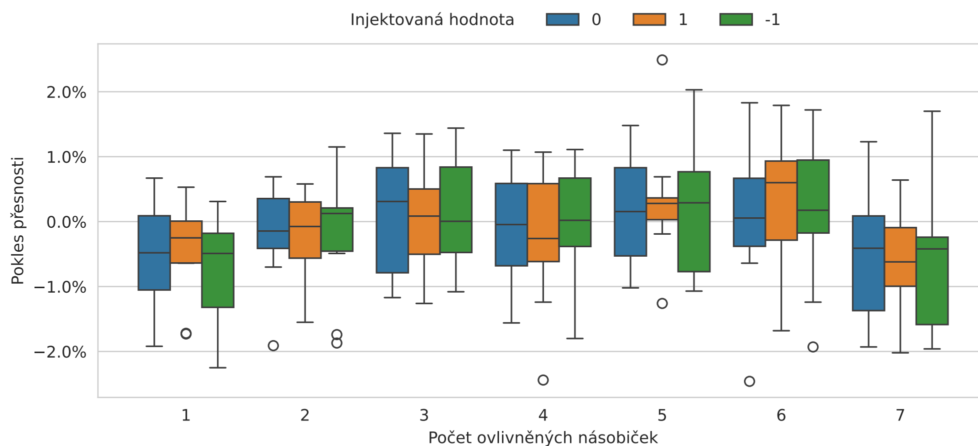
Na začátek byla změřena přesnost neuronové sítě bez injekce poruch na plném datasetu a poté na 10% datasetu (tabulka 7.4), který byl vytvořen použitím prvních 100 obrázků z každé kategorie z plného datasetu. 10% dataset (tzn. 1 000 obrázků) byl vybrán kvůli výraznému snížení časové náročnosti. Z výsledků je patrné, že nedošlo ke skoro žádnému poklesu přesnosti. Při vyhodnocování na procesoru pomocí celočíselné aritmetiky sice došlo ke zvýšení přesnosti o 0,5 %, ale na samotném akcelerátoru nedošlo k žádnému rozdílu.

Zařízení	**Aritmetika**	**Celý dataset**	**10% dataset**
ARM Cortex-A53	float	75,5 %	75,6 %
ARM Cortex-A53	int8	75,6 %	76,1 %
NVDLA „small“	int8	75,1 %	75,1 %

Tabulka 7.4: Přesnost neuronové sítě ResNet-18 na testovací části plného datasetu CIFAR-10 (10 000 obrázků) a 10% datasetu (1 000 obrázků) v závislosti na zařízení a aritmetice.

Dalším krokem bylo zjistit pokles přesnosti při injekci poruch. Pro porovnání byla použita stejná injekce poruch, která sloužila k vyhodnocení prvního experimentu. Podrobnosti o dané injekci poruch jsou tedy popsány v kapitole 7.2.1.

Výsledky pro injekci poruch byly vytvořeny pro plný dataset a pro 10% dataset. Graf s poklesem přesnosti při použití 10% datasetu je uveden na obrázku 7.1 a histogram na obrázku 7.2.



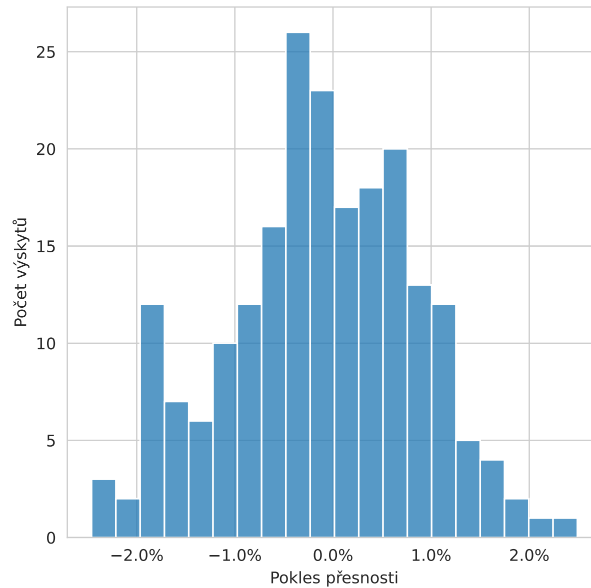
Obrázek 7.1: Pokles přesnosti při použití 10% datasetu při injekci poruch.

Interval spolehlivosti s konfidenční hladinou 95 % pro pokles přesnosti: (-0,26 %, 0,00 %). Z vyhodnocených výsledků je tedy patrné, že nedošlo k výraznému poklesu při použití 10% datasetu, a tudíž došlo k jeho využití při experimentech.

7.2 Injekce poruch

Pro vyhodnocení výsledků při injekci poruch byla použita neuronová síť ResNet-18 natrénovaná na datasetu CIFAR-10. Výsledky byly vyhodnoceny s použitím 10% testovacího datasetu CIFAR-10.

Většina experimentů byla provedena s injektováním hodnot 0, 1 a -1 s maskou označující všech 18 bitů. Hodnota 0 byla vybrána kvůli tomu, že se při injekci nacházejí všechny bity



Obrázek 7.2: Histogram pokles přesnosti při použití 10% datasetu při injekci poruch.

v logické nule. Opakem toho jsou všechny bity v logické jedničce, což znamená hodnotu -1. Hodnota 1 poté byla vybrána jako doplnění předcházejících stavů, kdy je pouze jeden bit (nejméně významný) v logické jedničce a zbytek bitů se nachází v logické nule.

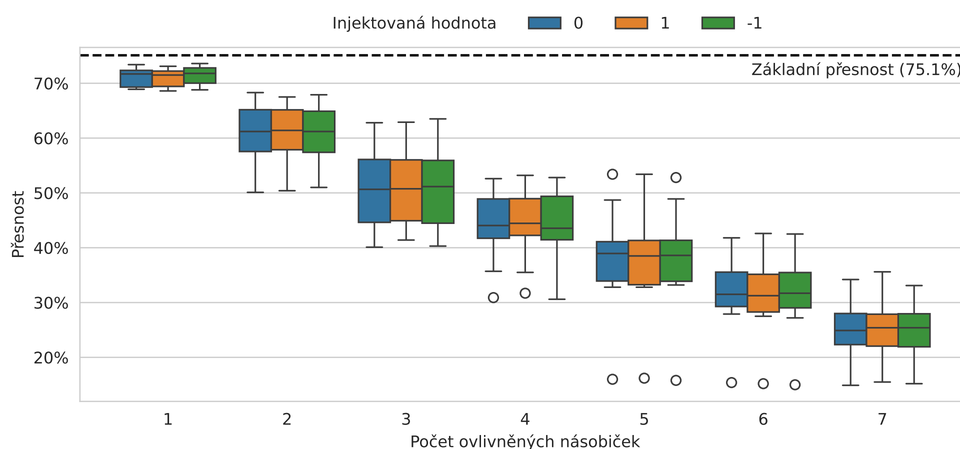
Byly provedeny i dva experimenty, které se zaměřily na injekci jednotlivých bitů bez ovlivňování ostatních. V těchto experimentech se uplatnila bitová maska, kterou vytvořený systém podporuje.

7.2.1 Počet ovlivněných násobiček

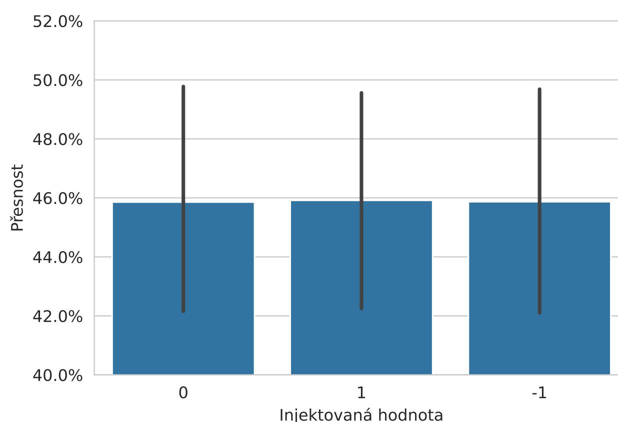
Prvním předpokladem bylo, že bude docházet k postupnému snižování přesnosti při injekci více násobiček naráz. Pro ověření této teorie bylo ovlivňováno několik násobiček. Prvně byla ovlivněna pouze jedna násobička a postupně docházelo ke zvyšování počtu ovlivněných násobiček, až bylo ovlivněno sedm násobiček naráz. Pro každý počet ovlivněných násobiček bylo náhodně vybráno 10 možných kombinací a žádná kombinace se neopakovala. (Kombinace byly náhodně vybrány při generování testovacích dat.) Pro každou kombinaci došlo k injekci tří různých hodnot, a to 0, 1 a -1.

Na výsledcích (obrázek 7.3) je opravdu vidět úbytek přesnosti se zvyšujícím se počtem ovlivněných násobiček. Při injekci jedné násobičky došlo k malému poklesu přesnosti, zatímco u zbytku je už pokles výraznější. Při ovlivnění dvou a tří násobiček je vzájemný pokles přesnosti okolo 10 %. Větší počet ovlivněných násobiček má už vzájemný pokles zhruba o 6 %.

Porovnání injektovaných poruch (obrázek 7.4) ukazuje, že jednotlivé injektované hodnoty skoro nijak nezmění výslednou přesnost.



Obrázek 7.3: Graf přesnosti v závislosti na počtu ovlivněných násobiček.



Obrázek 7.4: Graf porovnání přesnosti mezi jednotlivými injektovanými poruchami. Svislé černé čáry značí interval spolehlivosti.

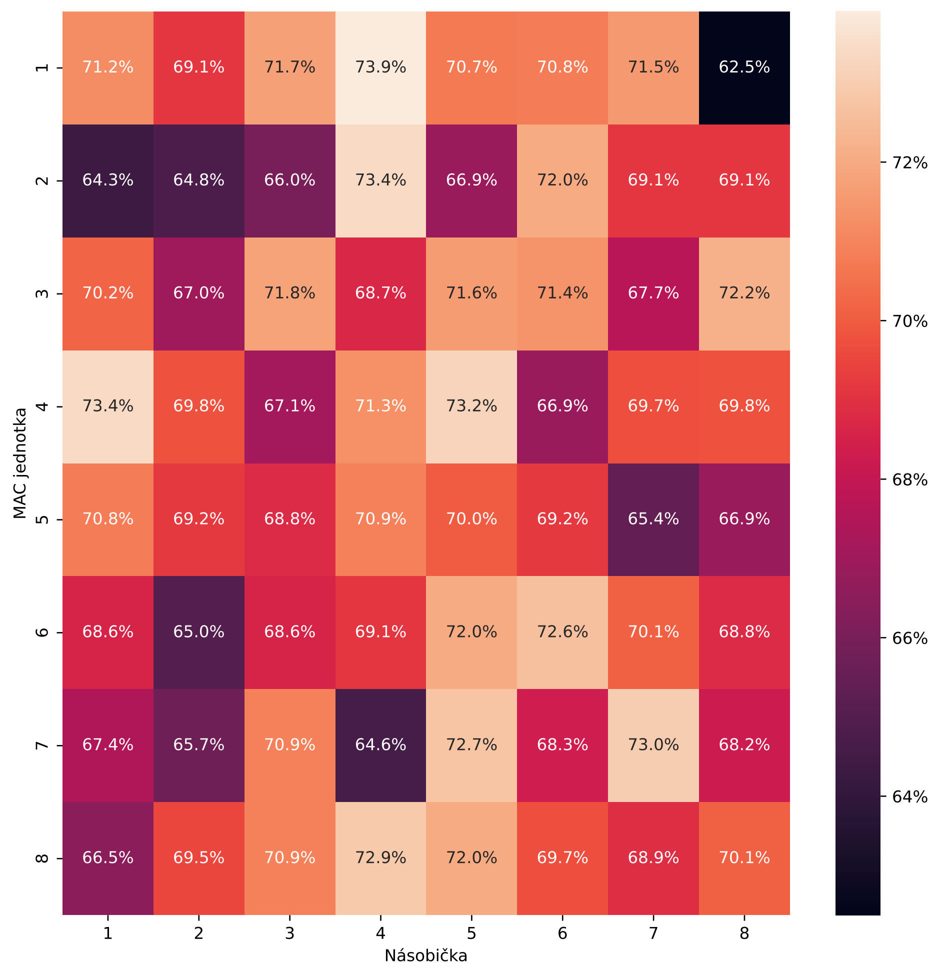
7.2.2 Senzitivita jednotlivých násobiček

Dalším předpokladem k ověření bylo, jestli není nějaká pozice násobiček nebo celá MAC jednotka více senzitivní na poruchu. K ověření tohoto předpokladu byla vždy ovlivněna jedna násobička v době vyhodnocování přesnosti. Ovlivněna byla celá výstupní hodnota z násobičky. Pro každou násobičku se ještě testovalo zvlášť vyhodnocení pro tři hodnoty, a to 0, 1 a -1.

Na teplotní mapě s výsledky (obrázek 7.5) není vidět žádný vzor, ze kterého by šlo vydedukovat nějaké závěry. Je pouze vidět, že některé násobičky skoro vůbec neovlivnili přesnost, a některé násobiček mají výraznější dopad na přesnost. Nenachází se ovšem na stejných pozicích v MAC jednotkách, ani ve stejné MAC jednotce.

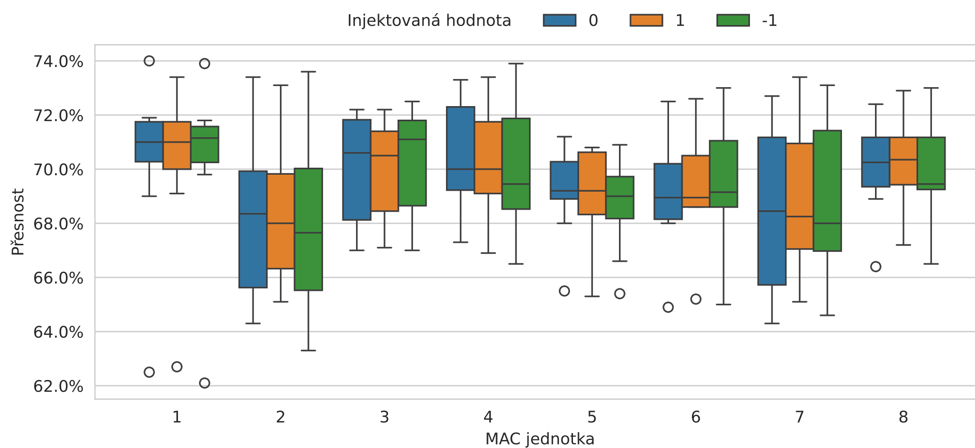
Medián celkové přesnosti činí 69,6 %. Nejhorší přesnost byla při ovlivnění poslední násobičky v MAC jednotce, konkrétně 62,5 %. Nejlépe naopak dopadla přesnost při ovlivnění čtvrté násobičky, také v první MAC jednotce, a to 73,9 %.

Při porovnání přesnosti mezi MAC jednotkami (obrázek 7.6) je vidět, že nejhůře dopadla 7. MAC jednotka s mediánem přesnosti 68,25 %, a hned poté 2. MAC jednotka s mediánem



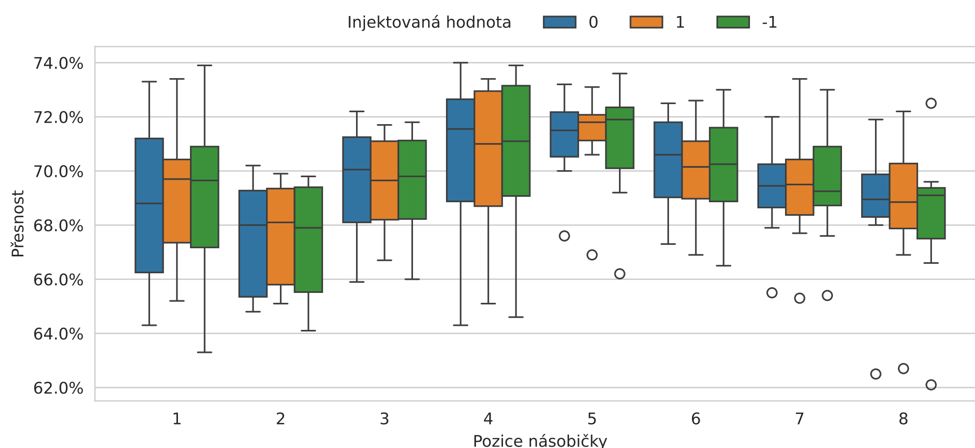
Obrázek 7.5: Teplotní mapa přesnosti při ovlivnění jednotlivých násobiček.

68,35 %. Pokles přesnosti ovšem není nijak významný, aby se dalo prohlásit, že jsou tyto jednotky senzitivnější než ostatní.



Obrázek 7.6: Graf přesnosti při injekci poruchy do jedné z násobiček dané MAC jednotky.

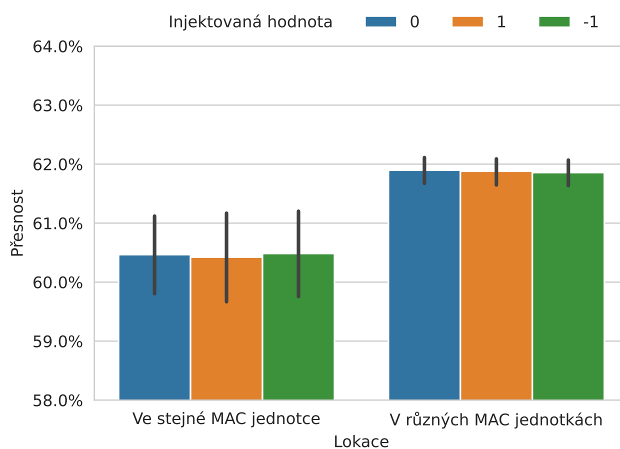
Výsledky pro pozici násobiček přes všechny MAC jednotky vypadají podobně (obrázek 7.7). Mediány pro všechny pozice jsou velmi podobné a nedá se prohlásit, že by byla přesnost při ovlivnění nějaké pozice násobiček výrazně ovlivněna.



Obrázek 7.7: Graf přesnosti při injekci poruchy do násobiček skrz MAC jednotky.

7.2.3 Rozdíl injekce dvou násobiček ve stejné a různé MAC jednotce

Dále byl zkoumán vliv injekce poruch do dvou násobiček naráz. Cílem bylo zjistit, jestli má na přesnost vliv vzájemná vzdálenost ovlivněných násobiček. Jako například, jestli se sníží přesnost víc, když budou ovlivněné dvě násobičky vedle sebe. Dalším cílem bylo zjistit, jestli je rozdíl v tom, když jsou obě ovlivněné násobičky ve stejné MAC jednotce a když je každá násobička v jiné MAC jednotce. Výsledky byly zjištěny pomocí ovlivnění dvou násobiček naráz pro všechny možné kombinace a byly ovlivněny vždy celé výstupy z násobiček na určenou hodnotu. Vyhodnocení proběhlo pro tři různé hodnoty, a to 0, 1 a -1.

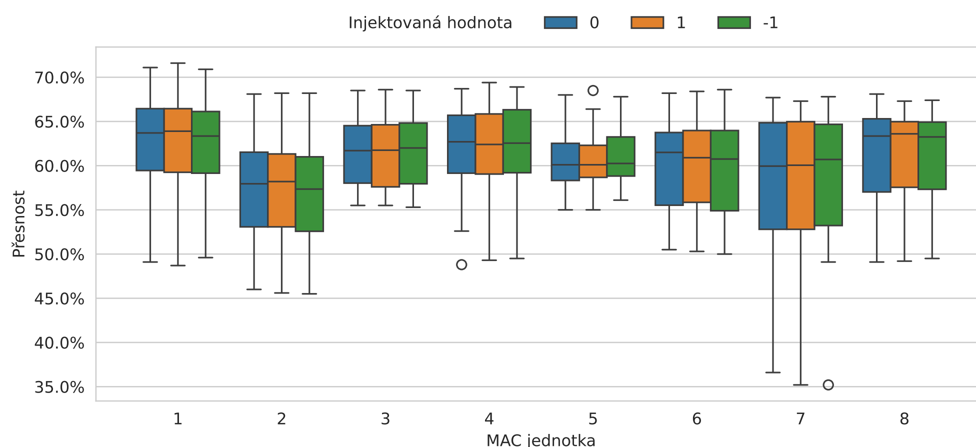


Obrázek 7.8: Porovnání přesnosti mezi ovlivněním dvou násobiček v jedné MAC jednotce a v různých MAC jednotkách.

Na obrázku 7.8 je vidět porovnání přesnosti při ovlivnění dvou násobiček ve stejné MAC jednotce a ve dvou různých MAC jednotkách. Medián přesnosti při ovlivňování dvou

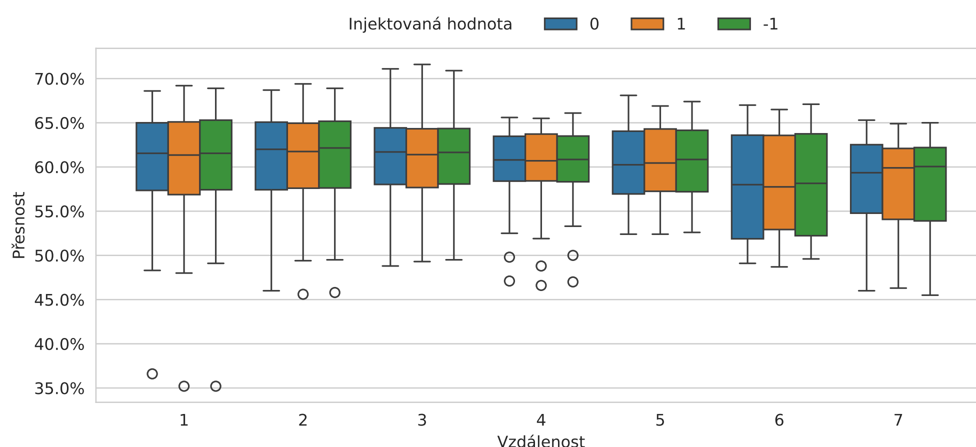
násobiček ve stejné MAC jednotce je 61,3 %, a ve dvou různých MAC jednotkách je 62,4 %. Rozdíl v přesnosti činní 1,1 %. Lze tedy říct, že při výskytu dvou vadných násobiček v jedné MAC jednotce je snížení přesnosti skoro stejné, jako při výskytu ve dvou různých MAC jednotkách.

Medián přesnosti při ovlivnění násobiček v rámci jedné MAC jednotky je 61,3 %. Z výsledků (obrázek 7.9) nijak nevyplývá, že by byla nějaká MAC jednotka výrazně senzitivnější.



Obrázek 7.9: Graf přesnosti při ovlivnění dvou násobiček v rámci dané MAC jednotky.

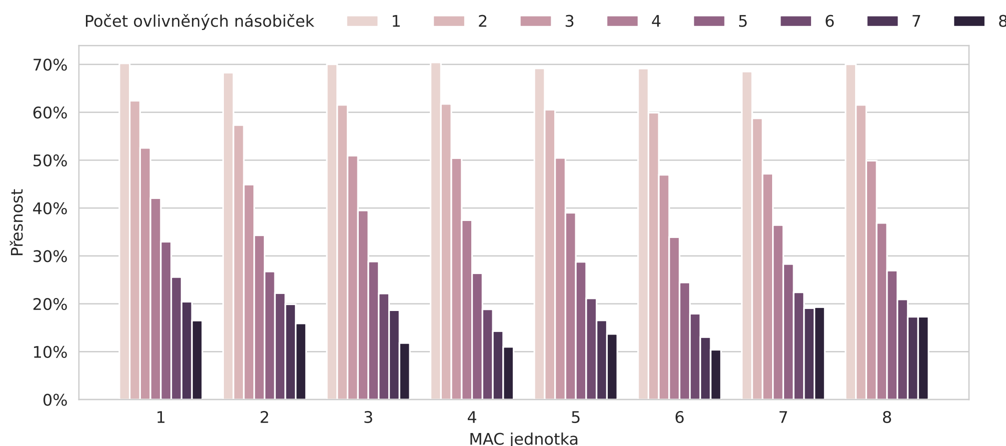
Na obrázku 7.10 je znázorněný graf, ukazující závislost přesnosti na vzájemné vzdálenosti ovlivněných násobiček v rámce stejné MAC jednotky. Vzdálenost je určena jako počet kroků, které se musí provést, aby se přešlo z 1. ovlivněné násobičky na 2. ovlivněnou násobičku. Nejmenší možná vzdálenost je tedy 1 a největší 7. Nejmenší dopad na přesnost měla vzdálenost 2, kde je medián přesnosti 62,0 %. Nejhorší dopad byl u vzdálenosti 6 s mediánem 57,9 %, tzn. horší o 3,4 % od celkového mediánu. Nelze tedy prohlásit, že by měla vzájemná vzdálenost ovlivněných násobiček v rámci stejné MAC jednotky nějaký zásadní vliv na přesnost.



Obrázek 7.10: Graf přesnosti v závislosti na vzdálenosti ovlivněných násobiček v rámci stejné MAC jednotky.

7.2.4 Rozdíl při injekce více násobiček naráz ve stejné MAC jednotce

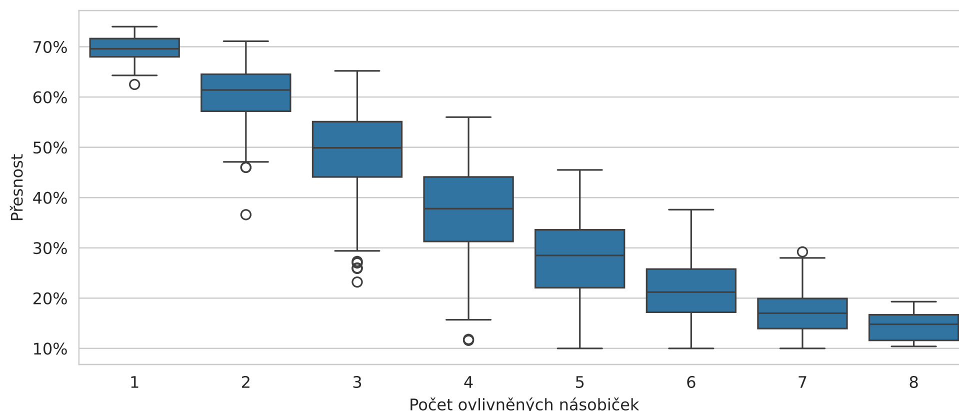
Tento experiment se mohl provést díky tomu, že bylo v předchozím experimentu 7.2.3 dokázáno, že ovlivnění dvou násobiček v jedné MAC jednotce není výrazně odlišné od ovlivnění ve dvou různých MAC jednotkách. Cílem tohoto experimentu bylo vyzkoušet větší počet ovlivněných násobiček pro všechny možné kombinace, a tím zpřesnit výsledky z experimentu 7.2.1, který byl proveden jako první. Při ovlivňování více než dvou násobiček je ovšem počet všech možných kombinací na rozsahu všech MAC jednotek příliš velký na to, aby bylo možné provést všechny kombinace (např. už při pouhém ovlivnění tří násobiček je 41 664 všech možných kombinací, což by zabralo zhruba 53 hodin pro jednu injektovanou hodnotu). Při vycházení z výsledků předchozího experimentu 7.2.3 tedy stačí vyzkoušení všech kombinací v rámci MAC jednotky, a výsledky budou podobné jako při zkoušení kombinací přes jednotlivé MAC jednotky. Tímto zjednodušením se počet všech kombinací stal testovatelným v poměrně krátkém čase (zhruba 156 minut pro jednu injektovanou hodnotu). Nevýhodou při tomto zjednodušení je fakt, že nelze otestovat víc naráz ovlivněných násobiček, než je jejich počet v jedné MAC jednotce, tzn. 8. Při tomto experimentu byla injektována pouze jedna hodnota poruchy, a to přesně 0.



Obrázek 7.11: Graf porovnání přesnosti mezi MAC jednotkami při ovlivnění daného počtu násobiček.

Na obrázku 7.11 je vidět porovnání počtu ovlivněních násobiček uvnitř jednotlivých MAC jednotek. Při injekci 1, 2 a 3 násobiček jsou výsledky velice podobné. Výraznější rozdíl je jenom při ovlivnění 3 násobiček ve 2. MAC jednotce. Při ovlivnění více než 3 násobiček se už výsledky liší. Zajímavé je, že MAC jednotky 3, 4, 5 a 6 jsou více senzitivní při ovlivnění všech násobiček v nich, tzn. 8. To samé platí při ovlivnění 7 násobiček, až na 3. MAC jednotku, ta je na tom podobně jako ostatní. Dále je zajímavé, že u 7. a 8. MAC jednotky není žádný rozdíl mezi ovlivněním 7 a 8 násobiček. Nedošlo k poklesu při zvýšení počtu ovlivněních násobiček tak, jak by se očekávalo z ostatních MAC jednotek. Teoreticky by se takto mohlo dít kvůli tomu, že v těchto MAC jednotkách není poslední násobička tolik využita, to ovšem nesedí s výsledky z experimentu 7.2.2. Ověření této teorie by bylo potřeba potvrdit dalším experimentem, třeba pomocí rozboru kódu pro NVDLA, ze kterého by se mělo dát zjistit využití jednotlivých násobiček. Analýza NVDLA kódu je už ovšem mimo rozsah této práce, takže nebyla provedena a důvod tohoto jevu tedy zůstává neznámý.

Přesnost v závislosti na počtu ovlivněných násobiček skrz všechny MAC jednotky je znázorněna na grafu 7.12. Při porovnání s grafem 7.3 z experimentu 7.2.1 jsou vidět jisté rozdíly. Přesnost při ovlivnění 1, 2 a 3 násobiček je velmi podobná, ale při větším počtu ovlivněných násobiček je medián přesnosti nižší, až o 10 %, přesné hodnoty jsou v tabulce 7.5. Porovnány jsou přesnosti s injektovanou hodnotou 0.



Obrázek 7.12: Graf přesnosti při ovlivnění daného počtu násobiček v rámci jedné MAC jednotky.

Počet ovlivněných násobiček	Náhodně	Všechny kombinace	Pokles přesnosti
1	71,70 %	69,60 %	2,10 %
2	61,20 %	61,40 %	-0,20 %
3	50,65 %	49,90 %	0,75 %
4	44,05 %	37,80 %	6,25 %
5	38,95 %	28,50 %	10,45 %
6	31,50 %	21,20 %	10,30 %
7	24,90 %	17,00 %	7,90 %

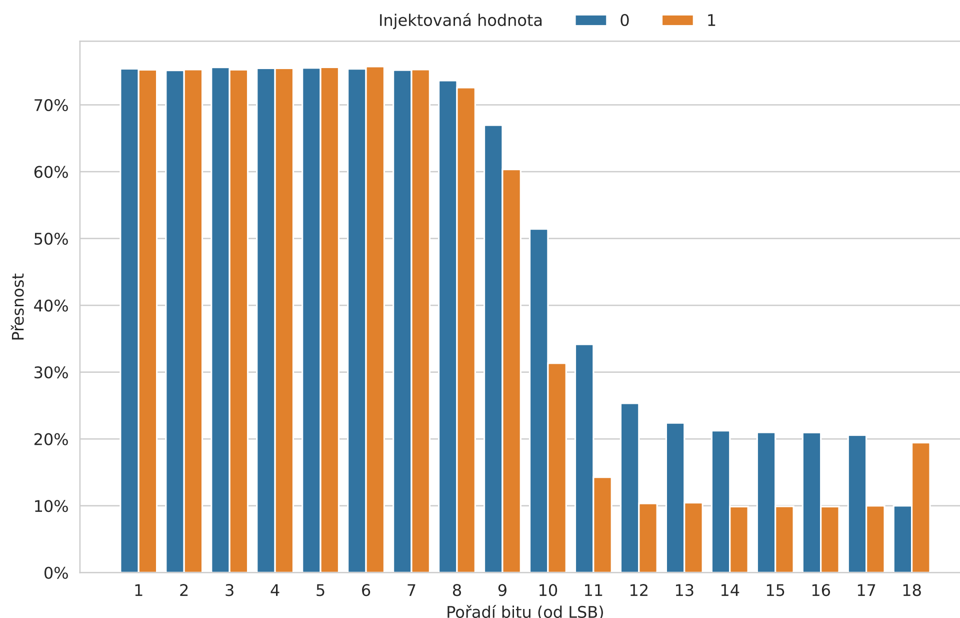
Tabulka 7.5: Rozdíl přesnosti mezi ovlivněním daného počtu násobiček přes všechny MAC jednotky náhodně (10 různých pokusů) a všechny kombinace v rámci MAC jednotky.

7.2.5 Odolnost jednotlivých bitů

Dalším experimentem bylo zkusit odolnost při injekci jednotlivých bitů kvůli předpokladu, že bude postupně klesat úspěšnost při ovlivňování bitů na vyšších pozicích. Experiment byl proveden pomocí injekce dané hodnoty postupně do všech bitů. Injekce byla kvůli snížení časové náročnosti vyhotovena pro každou 1. násobičku v každé MAC jednotce, tzn. celkem 8 injekcí pro každou hodnotu na daném bitu. Injekce proběhla pro obě možné hodnoty bitu, tzn. 0 a 1.

Z výsledků (obrázek 7.13) je vidět, že k poklesu opravdu dochází při zvyšování pozice injektovaného bitu, ale pokles je převážně mezi 8. a 12. pozicí, jinak je přesnost velmi podobná. Při ovlivnění bitů na 1. až 7. pozici nedošlo k žádnému poklesu přesnosti. Došlo ke

spíše velmi mírnému zlepšení přesnosti o zhruba 0,35 %. Tím, že byly ovlivněny jen některé násobičky, mohla chyba v některých případech i zlepšit přesnost, což se stává i například při injekci gaussovského a jiného šumu [17].



Obrázek 7.13: Graf přesnosti při injekci jednotlivých bitů v násobičce.

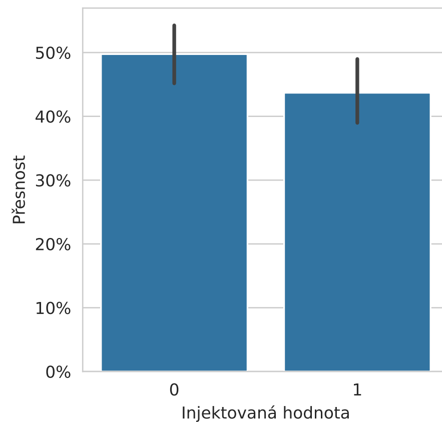
Mezi 8. a 12. pozicí bitů je zajímavé, že při injekci logické jedničky je pokles přesnosti větší než u logické nuly. Při injekci 12. pozice bitu logickou jedničkou se už přesnost dostala na pouhých 10 % a i pro vyšší bity zůstává na této hodnotě. Při rozdělování obrázků do 10 kategorií to znamená, že by se dosáhlo stejné úspěšnosti, kdyby se vždy určovala stejná kategorie pro všechny obrázky. Přesnost tedy aspoň neklesla pod tuto hranici.

Mezi 13. až 17. bitem je už i přesnost při injekci logické nuly téměř ustálena, a to na hodnotě kolem 19,7 %. Z hodnot lze tedy předpokládat, že se na těchto pozicích nachází převážně logické nuly, a proto došlo k menšímu poklesu. Změna je až u nejvíce významného bitu, kde se přesnost pro logické hodnoty obrátila, tzn. při injekci do logické jedničky je větší přesnost než při logické nule. Jedná se o celočíselný datový typ se znaménkem a nejvyšší bit tedy určuje právě znaménko. Z toho lze předpokládat, že se při výpočtech nachází více hodnot se záporným číslem a kvůli tomu je přesnost vyšší pro logickou jedničku.

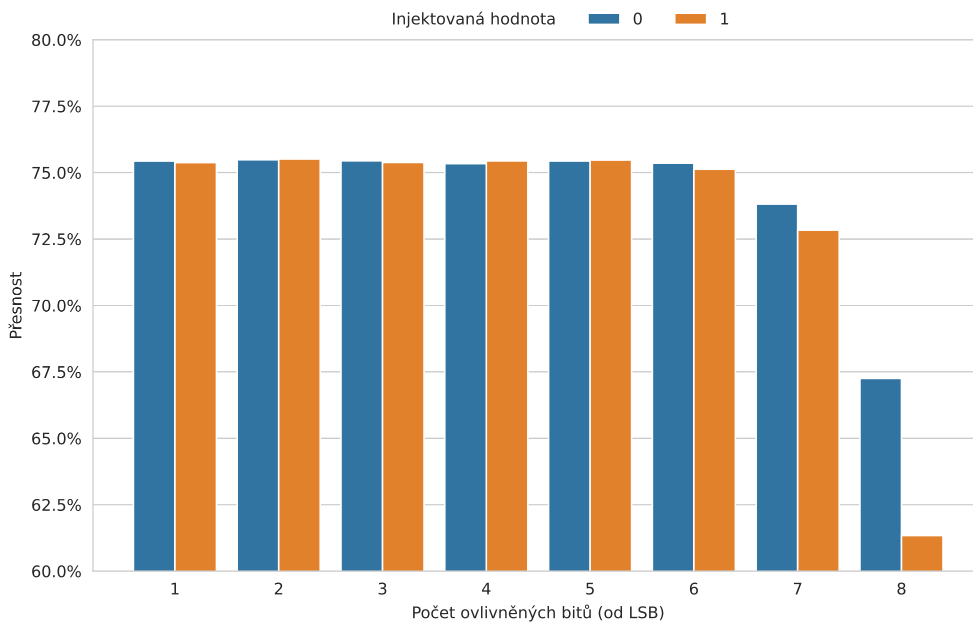
Při porovnání přesnosti podle pouhé injektované hodnoty bitu (obrázek 7.14) je tedy přesnost vyšší při injekci logické nuly o 9 %.

7.2.6 Injekce více bitů naráz

Jelikož nebyla v předchozím experimentu nijak ovlivněna přesnost při injekci bitů na nižších pozicích, vznikl předpoklad, jestli se dojde ke stejnému závěru i při injekci více bitů na těchto pozicích. Byla tedy provedena injekce s postupným zvyšováním ovlivněných bitů od nejméně významného bitu. Počet ovlivněných bitů byl od jednoho do osmi bitů. Hodnoty byly změněny na samé logické jedničky a samé logické nuly. Injekce byla vyzkoušena na všech násobičkách.



Obrázek 7.14: Graf porovnání přesnosti v závislosti na injektované hodnotě bitu.



Obrázek 7.15: Graf přesnosti při ovlivnění více bitů naráz v rámci násobičky.

Na výsledcích (obrázek 7.15) jsou vidět velmi podobné výsledky jako v předchozím experimentu. Jediný rozdíl je, že dochází k poklesu přesnosti už při ovlivněných 7 bitech (tzn. že byla ovlivněna i 7. pozice bitu). Lze tedy říct, že prvních 7 bitů od nejméně významného bitu je možné při výpočtech vynechat bez jakéhokoliv dopadu na přesnosti. Toho by šlo například využít při ukládání výsledků do paměti, kde by mohlo dojít právě ke zmenšení paměti o těchto 7 bitů nebo použití méně přesné paměti, které jsou i jednodušší na vytvoření [26]. V MAC jednotce dochází ještě ke sčítání všech hodnot z násobiček a až pak se hodnota dostává ven z MAC jednotky, takže by bylo potřeba řešit dopad s omezením méně významných bitů spíše až u výsledné sumy, která má 19 bitů. Jedná se tedy spíše o nastínění možností.

7.3 Dopad postupného sčítání v MAC jednotkách

Jako bylo již zmíněno v kapitole 4.4.1, sčítání výsledné sumy z výstupu násobiček je v MAC jednotce realizováno pomocí sčítaček zapojených do série. Toto řešení není ideální z časového hlediska, jelikož dochází ke zpoždění o čase *7t*, kde t reprezentuje čas potřebný sčítačkou pro sečtení dvou čísel. Mnohem lepší řešení je použít stromovou strukturu sčítaček, kde dochází k nižšímu zpoždění, a to *3t*. Díky této úpravě bylo možné zvýšit frekvenci NVDLA z původně maximálních 187,5 MHz na 250 MHz se stále úspěšným časováním. Je nutné ovšem dodat, že došlo k úpravě původního NVDLA bez modulu s injekcí chyb. Touto změnou došlo ke zrychlení inference o 19,3 % při zvýšení frekvence o 33,3 %. Porovnání konkrétních časů je uvedeno v tabulce 7.6.

Zařízení	**Frekvence**	**Čas na dataset (s)**	**Čas na obrázek (ms)**
NVDLA „small"	187,5 MHz	45,8	4,585
NVDLA „small"	250,0 MHz	38,4	3,843

Tabulka 7.6: Čas evaluace neuronové sítě ResNet-18 na testovací části datasetu CIFAR-10 (10 000 obrázků).

Kapitola 8

Závěr

Cílem práce bylo přidat do akcelerátoru neuronových sítí podporu pro injekci poruch a následně provést testování jeho odolnosti na FPGA. Pro demonstraci funkčnosti řešení byla provedena analýza jeho odolnosti pomocí neuronové sítě ResNet-18 natrénované na datasetu CIFAR-10.

Na rozšíření o injekci poruch byl vybrán akcelerátor NVIDIA Deep Learning Accelerator (NVDLA). Z důvodů limitů FPGA byla vybrána „small“ konfigurace, která obsahuje 8 MAC jednotek po 8 násobičkách. Rozšíření injekce poruch bylo realizováno invazivní metodou pomocí tzv. sabotážní techniky (saboteurs technique). V hardwaru byl vytvořen modul, který přerušuje spojení mezi násobičkou a sčítačkou. Modul podporuje úpravu jednotlivých bitů na logickou jedničku a nulu. Je tedy možné provádět simulaci trvalých, přechodných a občasných poruch.

Akcelerátor dosahoval průměrné rychlosti 4,585 ms na jeden obrázek při inferenci neuronové sítě ResNet-18 (CIFAR-10). Vyhodnocení přesnosti na celé testovací části datasetu trvalo 45,8 sekund. Při velmi hrubých výpočtech je představené řešení zhruba 2 500krát rychlejší než řešení v práci [27]. Při experimentech byla testovací část datasetu zmenšena na 10 %, aby došlo k rychlejšímu získání dat. Pro získání dat bylo potřeba provést kolem 15 000 inferencí, což by na plném datasetu trvalo zhruba 190 hodin. Díky zmenšení datasetu bylo možné získat tyto data za zhruba 21 hodin.

Celá implementace akcelerátoru s injekcí poruch zabrala na FPGA 41,7 % LUT jednotek a 23,0 % klopných obvodů. Rozšíření o injekci poruch zvýšilo počet využitých LUT jednotek o 0,7 % a klopných obvodů o 0,3 %.

Experimenty neprokázaly, že by byly nějaké MAC jednotky či násobičky výrazně senzitivnější na poruchy. Úbytek přesnosti při injekci hodnot 0, 1 a -1 byl téměř stejný. Bylo zjištěno, že při ovlivnění sedmi nejméně významných bitů došlo k velmi mírnému nárůstu přesnosti. Jinak řečeno, nabízí se zde možnost vynechat tyto bity za účelem zjednodušení.

Práce má několik různých rozšíření. Je možné zkusit akcelerátor zjednodušit o méně významné bity, jak již bylo naznačeno. Dále je zde prostor ke zrychlení akcelerátoru, a to dvěma způsoby. Jedno možné řešením je umístit do FPGA více akcelerátorů. Vzhledem k tomu, že celé řešení zabírá aktuálně 41,7 % LUT jednotek, je jistě možné použít dva akcelerátory. Teoreticky by mohlo být možné použít i tři. Druhou možností je změnit konfiguraci NVDLA. Mohlo by dojít ke zvýšení počtu MAC jednotek a násobiček až na limit použitého FPGA. Dále je možné zkoumat vliv těchto dvou parametrů na rychlost inference různých neuronových sítí a najít optimální řešení.

Hlavním přínosem práce je ovšem připravení platformy pro rychlejší testování akcelerátoru na odolnost vůči poruchám. Architektura je navržená tak, aby bylo možné vytvořit si

vlastní injekce chyb. Aktuální řešení může být rozšířené například o poruchu, kdy dochází k převrácení hodnoty bitu (bit flipping) nebo se můžou injektovat hodnoty z náhodného generátoru čísel. Díky rychlejšímu vyhodnocení vůči simulacím lze také očekávat komplexnější analýzu odolnosti systému. Další možností je přidání injekce poruch do jiných částí akcelerátoru, například do vstupních dat násobiček.

Literatura

[1] Advanced Micro Devices, Inc. *DPUCZDX8G for Zynq UltraScale+ MPSoCs: Product Guide (PG338)* online. Průvodce produktem, V4.1. USA: Advanced Micro Devices, Inc., leden 2023. Dostupné z: https://docs.amd.com/viewer/book-attachment/qb1FjKi6LzGJ8PO23Kdbig/VxJj3TGK9Y4tnw3YHr2w9g. [cit. 2024-04-02].

[2] Ahmadilivani, M. H.; Barbareschi, M.; Barone, S.; Bosio, A.; Daneshtalab, M. et al. Special Session: Approximation and Fault Resiliency of DNN Accelerators. In: online. Duben 2023. Dostupné z: https://arxiv.org/abs/2306.04645. [cit. 2024-04-02].

[3] Benso, A. a Prinetto, P. *Fault injection techniques and tools for embedded systems reliability evaluation.* 1. vyd. Boston: Kluwer Academic Publishers, 2003. Frontiers in electronic testing. ISBN 1-4020-7589-8.

[4] Chen, Y.-H.; Krishna, T.; Emer, J. S. a Sze, V. *Eyeriss: An Energy-Efficient Reconfigurable Accelerator for Deep Convolutional Neural Networks* online. 2017. 127-138 s. Dostupné z: https://ieeexplore.ieee.org/document/7738524. [cit. 2024-04-02].

[5] Ge, Z.; Liu, S.; Wang, F.; Li, Z. a Sun, J. *YOLOX: Exceeding YOLO Series in 2021* online. Megvii Technology, červenec 2021. Dostupné z: https://arxiv.org/abs/2107.08430. [cit. 2024-04-02].

[6] Google Inc. *Edge TPU – Run Inference at the Edge* online. Dostupné z: https://cloud.google.com/edge-tpu. [cit. 2024-04-02].

[7] He, K.; Zhang, X.; Ren, S. a Sun, J. *Deep Residual Learning for Image Recognition* online. Microsoft Research, prosinec 2015. Dostupné z: https://arxiv.org/abs/1512.03385. [cit. 2024-04-02].

[8] He, Y.; Balaprakash, P. a Li, Y. FIdelity: Efficient Resilience Analysis Framework for Deep Learning Accelerators. In: *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* online. 2020, s. 270–281. Dostupné z: https://doi.org/10.1109/MICRO50266.2020.00033. [cit. 2024-04-02].

[9] Jia, Y.; Shelhamer, E.; Donahue, J.; Karayev, S.; Long, J. et al. *Caffe: Convolutional Architecture for Fast Feature Embedding* online. 2014. Dostupné z: https://arxiv.org/abs/1408.5093. [cit. 2024-04-02].

[10] Johnson, B. W. *Design and analysis of fault-tolerant digital systems.* 1. vyd. Addison-Wesley Pub. Co., 1989. Addison-Wesley series in electrical and computer engineering. ISBN 0-20-107570-9.

[11] Koren, I. a Krishna, C. M. *Fault Tolerant Systems*. 1. vyd. Elsevier, 2007. ISBN 978-0-12-088525-1.

[12] Krizhevsky, A. *Learning Multiple Layers of Features from Tiny Images* online. MaRS Centre, West Tower, 661 University Avenue, Suite 505: Canadian Institute for Advanced Research, duben 2009. Dostupné z: https://www.cs.toronto.edu/~kriz/learning-features-2009-TR.pdf. [cit. 2024-04-02].

[13] Kvasnička, J.; Kubalík, P. a Kubátová, H. *Experimental emulation of FPGA bitstream faults in combinatorial circuits* online. Karlovo nam. 13, 121 35 Praha 2: Department of Computer Science and Engineering, Czech Technical University in Prague, 2008. Dostupné z: https://ddd.fit.cvut.cz/ddd/publ/2008/Kvasnicka_CSE.pdf. [cit. 2024-04-02].

[14] LeiWang1999. *NVDLA Xilinx FPGA Mapping* online. Dostupné z: https://zhuanlan.zhihu.com/p/378202360. [cit. 2024-04-02].

[15] Lin, T.-Y.; Maire, M.; Belongie, S.; Bourdev, L.; Girshick, R. et al. *Microsoft COCO: Common Objects in Context* online. Microsoft Corporation, květen 2015. Dostupné z: https://arxiv.org/abs/1405.0312. [cit. 2024-04-02].

[16] Mrazek, V.; Hrbacek, R.; Vasicek, Z. a Sekanina, L. EvoApprox8b: Library of approximate adders and multipliers for circuit design and benchmarking of approximation methods. In: *Design, Automation Test in Europe Conference Exhibition (DATE), 2017* online. March 2017, s. 258–261. ISSN 1558-1101. Dostupné z: https://doi.org/10.23919/DATE.2017.7926993. [cit. 2024-04-02].

[17] Mrazek, V.; Vasicek, Z.; Sekanina, L.; Hanif, M. A. a Shafique, M. ALWANN: Automatic Layer-Wise Approximation of Deep Neural Network Accelerators without Retraining. In: *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)* online. IEEE, Listopad 2019. Dostupné z: https://doi.org/10.1109/iccad45719.2019.8942068. [cit. 2024-04-02].

[18] NVIDIA Corporation. *NVDLA Open Source Hardware* online. Dostupné z: https://github.com/nvdla/hw. [cit. 2024-04-02].

[19] NVIDIA Corporation. *NVDLA Open Source Hardware* online. Dostupné z: https://github.com/nvdla/sw. [cit. 2024-04-02].

[20] NVIDIA Corporation. *NVIDIA Deep Learning Accelerator* online. Dostupné z: http://nvdla.org/. [cit. 2024-04-02].

[21] Nvidia Corporation. NVIDIA Jetson Xavier Series. *NVIDIA Jetson Xavier Series* online. 21. července 2023. Dostupné z: https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-xavier-series/. [cit. 2024-04-02].

[22] OPEN AI LAB. *Tengine* online. Dostupné z: https://github.com/OAID/Tengine. [cit. 2024-04-02].

[23] Redmon, J.; Divvala, S.; Girshick, R. a Farhadi, A. *You Only Look Once: Unified, Real-Time Object Detection* online. University of Washington, Allen Institute for AI, Facebook AI Research, červen 2015. Dostupné z: https://arxiv.org/abs/1506.02640. [cit. 2024-04-02].

[24] Ruospo, A.; Gavarini, G.; Sio, C. de; Guerrero, J.; Sterpone, L. et al. Assessing Convolutional Neural Networks Reliability through Statistical Fault Injections. In: *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)* online. 2023, s. 1–6. Dostupné z: https://doi.org/10.23919/DATE56975.2023.10136998. [cit. 2024-04-02].

[25] Sharma, H.; Park, J.; Mahajan, D.; Amaro, E.; Kim, J. K. et al. From high-level deep neural models to FPGAs. In: *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)* online. 2016, s. 1–12. Dostupné z: https://doi.org/10.1109/MICRO.2016.7783720. [cit. 2024-04-02].

[26] Srinivasan, G.; Wijesinghe, P.; Sarwar, S. S.; Jaiswal, A. a Roy, K. Significance driven hybrid 8T-6T SRAM for energy-efficient synaptic storage in artificial neural networks. In: *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)* online. 2016, s. 151–156. Dostupné z: Significancedrivenhybrid8T-6TSRAMforenergy-efficientsynapticstorageinartificialneuralnetworks. [cit. 2024-04-02].

[27] Taheri, M.; Daneshtalab, M.; Raik, J.; Jenihhin, M.; Pappalardo, S. et al. SAFFIRA: a Framework for Assessing the Reliability of Systolic-Array-Based DNN Accelerators. In: online. 2024. Dostupné z: https://arxiv.org/abs/2403.02946. [cit. 2024-04-02].

[28] Weller, D. D.; Bleier, N.; Hefenbrock, M.; Aghassi Hagmann, J.; Beigl, M. et al. Printed Stochastic Computing Neural Networks. In: *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)* online. 2021, s. 914–919. Dostupné z: https://doi.org/10.23919/DATE51398.2021.9474254. [cit. 2024-04-02].

[29] Xilinx, Inc. *PetaLinux Tools Documentation: Reference Guide UG1144 (v2019.1)* online. Referenční příručka, V2019.1. USA: Xilinx, Inc., květen 2019. Dostupné z: https://docs.amd.com/v/u/2019.1-English/ug1144-petalinux-tools-reference-guide. [cit. 2024-04-02].

[30] Xilinx, Inc. 66780 – AXI SmartConnect – Release Notes and Known Issues. *AMD Customer Community: Adaptive SoC & FPGA Support* online. 25. srpna 2022. Dostupné z: https://support.xilinx.com/s/article/66780?language=en_US. [cit. 2024-04-02].

[31] Xilinx, Inc. 72293 – PetaLinux 2019.1 – Product Update Release Notes and Known Issues. *AMD Customer Community: Adaptive SoC & FPGA Support* online. 5. ledna 2022. Dostupné z: https://support.xilinx.com/s/article/72293. [cit. 2024-04-02].

[32] Xilinx, Inc. *Vivado Design Suite User Guide: Release Notes, Installation, and Licensing (UG973)* online. Uživatelská příručka, V2022.1. USA: Xilinx, Inc., duben 2022. Dostupné z: https://docs.amd.com/r/2022.1-English/ug973-vivado-release-notes-install-license/Supported-Operating-Systems. [cit. 2024-04-02].

Příloha A

Obsah DVD

- `README.txt` – soubor s obsahem paměťového média
- `Data` – data použitá pro práci a výstupy
  - `Datasets` – použité datasety
  - `Fault_injection` – vstupní a výstupní data k injekci chyb a jejich analýza
  - `NN_models` – modely neuronových sítí
- `Hardware` – soubory pro hardware
  - `Prebuild` – sestavené soubory pro FPGA
  - `Vivado_projects` – projekty pro Vivado Design Suite
- `Software` – soubory pro software
  - `Operating_systems` – archivy s distribucemi Linuxu
  - `PetaLinux` – sestavený PetaLinux a modul s NVDLA KMD
  - `Prebuild` – sestavené binární soubory
  - `Source_codes` – zdrojové kódy pro software
    - `Apps` – zdrojové kódy k vytvořeným aplikacím pro injekci poruch
- `Text` – text práce
  - `Tex` – zdrojové kódy pro text práce