VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOEVOLUČNÍ ALGORITMUS V FPGA

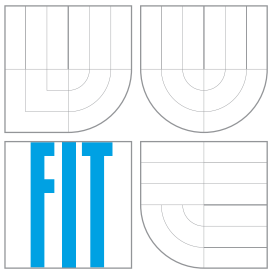
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK HRBÁČEK

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

KOEVOLUČNÍ ALGORITMUS V FPGA

COEVOLUTIONARY ALGORITHM IN FPGA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. RADEK HRBÁČEK

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. MICHAELA ŠIKULOVÁ

BRNO 2013

Abstrakt

Tato práce se zabývá návrhem hardwarové jednotky urychlující návrh obrazových filtrů pomocí koevolučních algoritmů. V práci je nejprve představena technologie rekonfigurovatelných logických obvodů, na kterých je akcelerační jednotka založena. Teoretická část dále stručně popisuje evoluční a koevoluční algoritmy, jejich principy a aplikace. Tradiční metody návrhu obrazových filtrů jsou porovnány s metodami inspirovanými procesy pozorovanými v přírodě. Navržená hardwarová jednotka využívá dvojici procesorů MicroBlaze doplněných o vlastní periferie pro akceleraci kartézského genetického programování. Koevoluční návrh obrazových filtrů je tak urychlen až 58 krát oproti optimalizované softwarové implementaci. Funkčnost jednotky je ověřena na úlohách návrhu filtru impulzního šumu a detektoru hran.

Abstract

This thesis deals with the design of a hardware acceleration unit for digital image filter design using coevolutionary algorithms. The first part introduces reconfigurable logic device technology that the acceleration unit is based on. The theoretical part also briefly characterizes evolutionary and coevolutionary algorithms, their principles and applications. Traditional image filter designs are compared with the biologically inspired design methods. The hardware unit presented in this thesis exploits dual MicroBlaze system extended by custom peripherals to accelerate cartesian genetic programming. The coevolutionary image filter design is accelerated up to 58 times. The hardware platform functionality in the task of impulse noise filter design and edge detector design has been empirically analyzed.

Klíčová slova

evoluční algoritmus, koevoluce, kartézské genetické programování, digitální zpracování obrazu, FPGA, MicroBlaze

Keywords

evolutionary algorithm, coevolution, cartesian genetic programming, digital image processing, FPGA, MicroBlaze

Citace

Radek Hrbáček: Koevoluční algoritmus v FPGA, diplomová práce, Brno, FIT VUT v Brně, 2013

Koevoluční algoritmus v FPGA

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením Ing. Michaely Šikulové.

.....
Radek Hrbáček
20. května 2013

Poděkování

Na tomto místě bych rád poděkoval vedoucí diplomové práce Ing. Michaele Šikulové za odborné vedení, konzultace a podnětné návrhy k práci.

Obsah

Seznam obrázků

Seznam tabulek

Seznam výpisů kódu

Kapitola 1

Úvod

Od nástupu výpočetní techniky zaznamenalo lidstvo obrovský pokrok, dnes se stále častěji setkáváme s různými formami umělé inteligence, která je schopna řešit stále složitější problémy dříve výhradně určené pro lidský mozek. S vývojem technologií roste i složitost návrhu nových zařízení, konvenční inženýrské metody návrhu již často nevedou k uspokojivým výsledkům s ohledem na dobu a cenu vývoje.

Moderní výpočetní a komunikační systémy čím dál více obsahují konfigurovatelné komponenty umožňující přizbůsobení architektury systému aktuálním požadavkům či podmínkám. Konfigurovat lze logické obvody (asi nejrozšířenější oblast zahrnující především obvody FPGA), analogové obvody, ale i mechanické systémy – např. antény.

Současně s rostoucím výkonem výpočetních systémů se rozvíjel také obor umělé inteligence. Dlouhou dobu byla snaha řešit složité problémy metodou *shora dolů*, tedy dekompozicí na jednodušší problémy – tyto algoritmy řešily lidské úlohy „inženýrským“ způsobem. Posupem času se však v umělé inteligenci začaly aplikovat mechanismy pozorovatelné v přírodě. Počítání podle přírody (natural computing) zahrnuje např. neuronové sítě, evoluční algoritmy, celulární automaty, výpočty založené na simulaci roje (částic nebo i hmyzu), dále pak výpočetní systémy založené na fyzikálních či chemických principech – kvantové, molekulární či DNA počítání.

Spojením konfigurovatelných hardwarových komponent a počítání podle přírody vzniká nová oblast výzkumu nazývaná evoluční hardware, která zahrnuje jak návrh obvodů, tak i proces adaptace v průběhu nasazení těchto obvodů.

Jednou z aplikací vyvíjejícího se hardware je návrh obrazových filtrů, nejčastěji nelineárních. S využitím evolučního návrhu je hledán vhodný kombinační obvod, jehož vstupem je okolí pixelu v obraze a výstupem vyfiltrovaný pixel. Proces evoluce je spuštěn na zadané množině trénovacích dat, nejčastěji je použita sada dvou obrázků – obrázek určený k filtraci a požadovaný výsledek filtrace. Pro urychlení evoluce a dosažení stabilnější konvergence se pak zavádí princip *koevoluce* inspirovaný interakcemi souběžně se vyvíjejících jedinců či celých populací v přírodě.

Tato práce pojednává o návrhu číslicových obrazových filtrů na FPGA za využití koevoluce. Kapitola 2 se zabývá rekonfigurovatelnými logickými obvody, zejména obvody FPGA, kapitola 3 stručně uvádí evoluční resp. koevoluční principy. Kapitola 4 je již věnována návrhu obrazových filtrů a to jak konvenčními způsoby, tak pomocí (ko)evoluce. Návrhu a implementaci akcelerační jednotky v FPGA urychlující evoluci obrazových filtrů je věnována kapitola 5. Kapitola 6 obsahuje vyhodnocení dosažených výsledků.

Kapitola 2

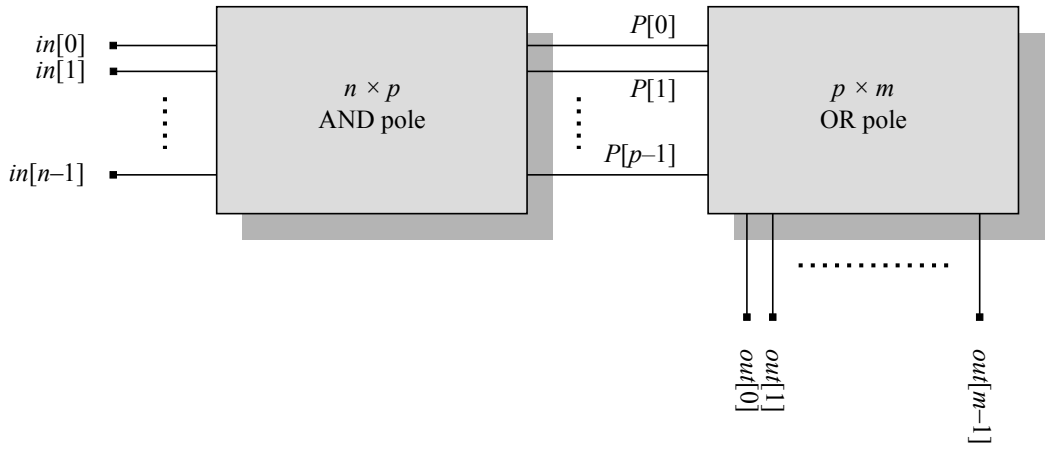
Rekonfigurovatelné logické obvody

V minulosti existovaly dvě varianty implementace číslicových obvodů – zákaznické integrované obvody (Application Specific Integrated Circuit, ASIC) a kombinace základních řad číslicových obvodů (např. řada 4000 nebo 7400). Cena vývoje zákaznického integrovaného obvodu byla velmi vysoká a vyplatila se pouze při rozsáhlých výrobních sériích. Oproti tomu číslicové obvody základních řad nabízely velmi omezené možnosti a výsledné obvody zabíraly velkou plochu.

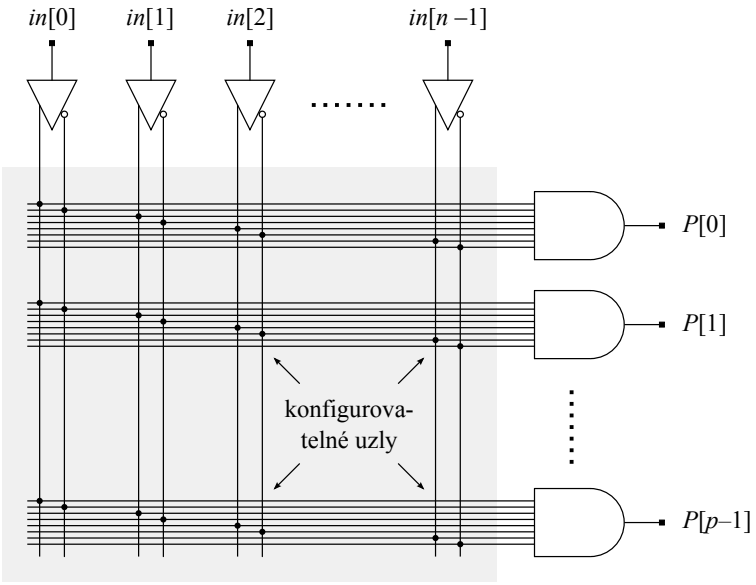
Z tohoto důvodu se začaly objevovat nové typy obvodů nabízející zpočátku omezené možnosti konfigurace. Místo několika základních obvodů tak stačilo použít jediný konfigurovatelný obvod.

2.1 ROM, PLA a PAL

Pro implementaci jednoduchých logických funkcí se nejprve začaly využívat paměti ROM (Read Only Memory). K realizaci n-vstupé logické funkce s m výstupy je potřeba paměť o kapacitě $2^n \cdot m$ bitů, vstupní signály slouží jako adresa místa v paměti, kde jsou uloženy výstupní hodnoty funkce. Teprve paměti PROM (Programmable ROM) však zákazníkovi umožnily nastavit vlastní obsah paměti. Následovala technologie EPROM (Erasable PROM) a EEPROM (Electricaly Erasable PROM) umožňující obsah paměti také smazat pomocí ultrafialového světla resp. pomocí elektrického napětí [2].



Obrázek 2.1: Struktura jednoduchých programovatelných obvodů – pole AND a OR.



Obrázek 2.2: Detail propojovací matice v poli AND.

Na obr. 2.1 je zobrazena struktura jednoduchého programovatelného obvodu složeného ze dvou konfigurovatelných polí AND a OR, detail pole AND je pak možné vidět na obr. 2.2. Paměti ROM také odpovídají tomuto modelu – pole AND představuje adresový dekodér s n vstupy a $p = 2^n$ výstupy, pole OR pak tvoří samotnou paměť [2].

Pro realizaci složitějších logických obvodů začaly být paměťové obvody nedostatečné. Doplněním pole OR o konfigurovatelné pole AND vznikly obvody PLA (Programmable Logic Array). Tyto obvody byly zpočátku drahé a poměrně pomalé, proto se objevily obvody PAL (Programmable Array Logic) s fixní maticí OR. Přestože některé pozdější typy obvodů PLA či PAL umožnily i vytvoření jednoduchých sekvenčních obvodů přivedením zpětné vazby z výstupu obvodu zpět na vstup pole AND, hlavní oblast použití byly kombinační obvody.

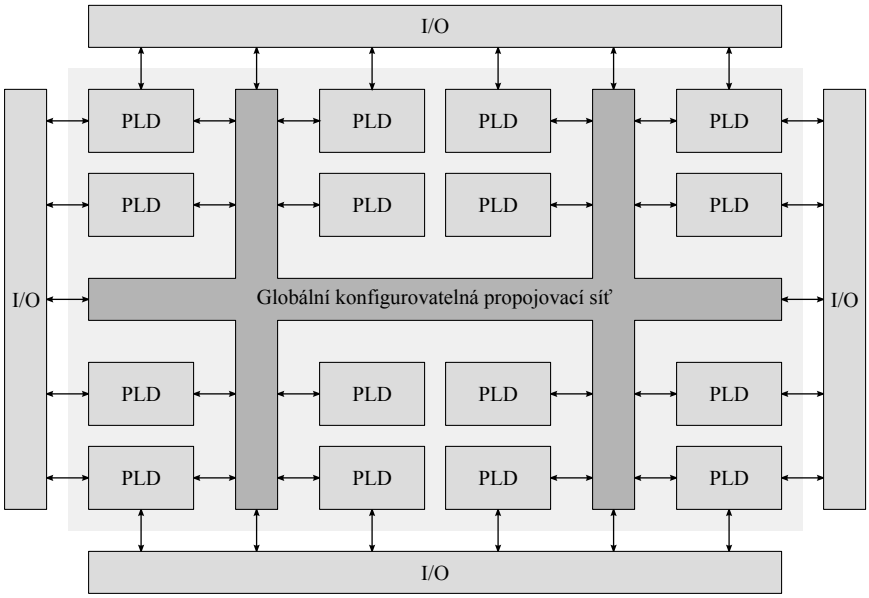
Posledním vývojovým stupněm těchto jednoduchých obvodů byla technologie GAL (Generic Array Logic) s totožnou logickou architekturou jako PAL, avšak s možností rekonfigurace. S těmito obvody se můžeme setkat i dnes, častěji se však používají složitější programovatelné obvody CPLD a FPGA [19].

2.2 CPLD

Rostoucí požadavky na složitost programovatelných obvodů nebylo možné donekonečna řešit zvětšováním polí AND resp. OR. Logickým krokem proto bylo vytvoření hierarchické struktury jednodušších obvodů (tzv. *makrobuněk*) spojených propojovací maticí. Jako makro-buňky byly využívány především jednodušší obvody PAL, rostoucí integrace obvodů později umožnila využít i složitější makrobuňky (PLA).

Typická architektura obvodu CPLD (Complex Programmable Logic Device) je vidět na obr. 2.3, programovatelné makrobuňky jsou propojeny globální propojovací sítí, vstupně-výstupní buňky jsou dostupné jak přes propojovací síť, tak i přímo z makrobuněk. Propojovací síť je také konfigurovatelná a dokáže mezi sebou propojit libovolné makrobuňky.

Oproti jednoduchým PLD obvodům nabízí CPLD výrazně více vstupů a výstupů při



Obrázek 2.3: Základní struktura obvodů CPLD.

rozumném nárůstu plochy čipu. Vstupně-výstupní buňky navíc často podporují různé napěťové úrovně a dovolují tak přímo propojovat obvody s různým napájecím napětím. Tyto buňky také často obsahují programovatelné zdvihací rezistory (*pull-up* resp. *pull-down*), obvody pro úpravu hran signálu nebo vstupní obvody s hysterezí [11].

Vzhledem ke své jednoduchosti se obvody CPLD hodí především v aplikacích, kde je potřeba dosáhnout velmi malého vstupně-výstupního zpoždění. Jejich konfigurace se nachází v non-volatilní paměti a tedy po odpojení napětí není ztracena. Obvody CPLD proto při startu okamžitě plní svou funkci.

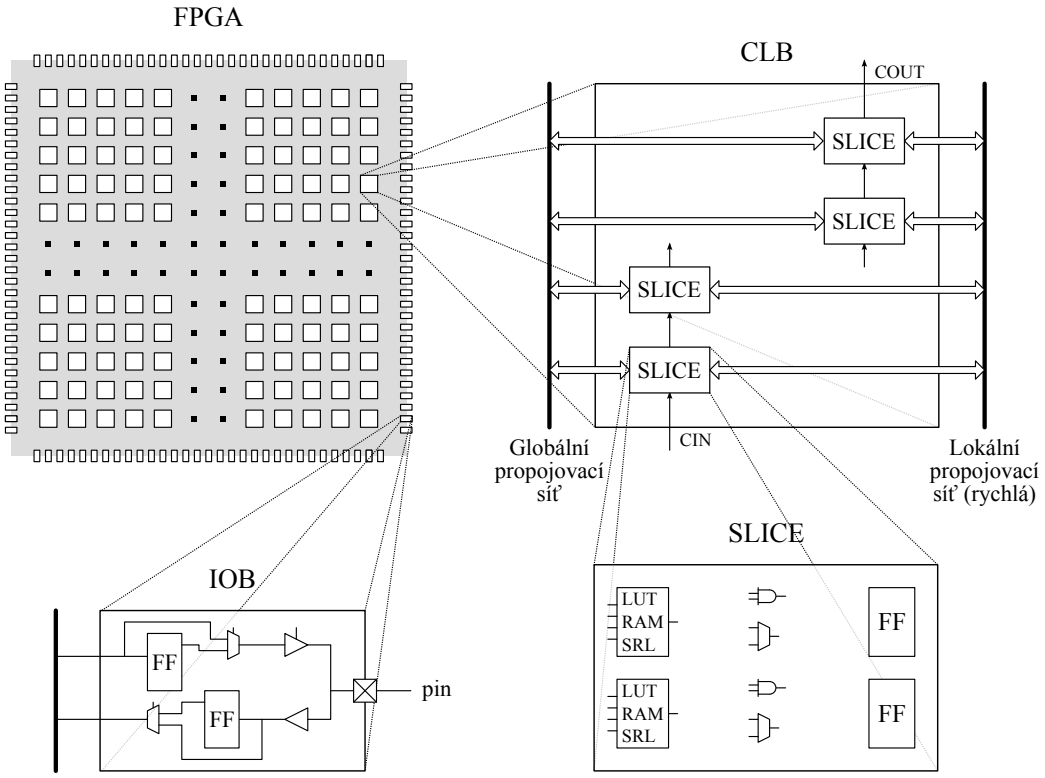
2.3 FPGA

Podobně jako obvody CPLD, jsou i FPGA (Field-Programmable Gate Array) tvořeny maticí konfigurovatelných bloků (Configurable Logic Block, CLB) propojených konfigurovatelnou sítí, složitost těchto obvodů je však mnohem vyšší. Bloky CLB jsou ještě dále děleny na menší bloky nazývané *slice* (řez) obsahující funkční generátory, multiplexory, hradla, registry, případně další konfigurovatelné komponenty (viz obr. 2.4).

Oproti obvodům CPLD jsou funkční generátory implementovány jako n-vstupé[1] vyhledávací tabulky (Look-Up Table, LUT). Tyto vyhledávací tabulky jsou fyzicky tvořené pamětí SRAM, většina FPGA umožňuje tuto paměť využít i jiným způsobem – jako tzv. distribuovanou paměť nebo jako posuvný registr. Registry je možné konfigurovat jako *flip-flop* (FF), tj. reagující na vzestupnou hranu hodinového signálu, nebo jako záchytné (*latch*) registry [17].

Kromě globální propojovací sítě obsahují FPGA i rychlé lokální linky k sousedním buňkám, v kombinaci s logikou přenosu uvnitř CLB tak dovolují vytváření např. rychlých sčítaček. Vzhledem k vysokému počtu CLB již není možné zaručit libovolné propojení těchto bloků a proto při vyšším zaplnění FPGA může přes dostatečné množství logiky

[1] Obvody FPGA obsahují nejčastěji 4-vstupé LUT, nejnovější typy pak 6-vstupé (Xilinx) či 8-vstupé (Altera) s více režimy použití (např. jako dva samostatné funkční generátory).



Obrázek 2.4: Základní struktura obvodů FPGA.

selhat propojení nebo mohou dlouhé propojovací cesty výrazně snížit pracovní frekvenci obvodu.

Současné rodiny obvodů FPGA nabízejí kromě základních logických bloků také blokové paměti s vysokými kapacitami, násobičky, DSP bloky, komunikační rozhraní (Ethernet, PCI Express) nebo i celé procesory [17]. DSP (Digital Signal Processing) bloky nabízejí vysoký výkon pro zpracování signálů v reálném čase. Typickou operací ve zpracování číslicových signálů je tzv. MAC (Multiply and Accumulate), DSP bloky jsou pro tento typ výpočtů patřičně vybaveny – kromě násobičky obsahují často také předsčítačku, za násobičkou může být další sčítačka, vše je odděleno registry a zřetězeno pro dosažení vysoké pracovní frekvence.

Důležitou částí obvodů FPGA je rozvod hodinových signálů tvořený speciální vrstvou rychlých linek uspořádaných tak, aby byl co nejvíce potlačen fázový rozdíl hodinového signálu v různých částech obvodu. Hodinových rozvodů bývá obvykle více a jsou napojeny na speciální bloky DCM (Digital Clock Manager), které hodinový signál upravují – mění jeho frekvenci, fázový posun, střídu apod.

Vstupně-výstupní bloky ještě více rozšiřují možnosti bloků v obvodech CPLD, každý pin je možné konfigurovat jako vstupní, výstupní nebo obousměrný, obvykle jsou podporovány různé napěťové úrovně, některé typy podporují dokonce diferenciální sběrnice. Existují obvody obsahující A/D nebo D/A převodníky.

Významným rozdílem oproti obvodům CPLD je způsob konfigurace FPGA obvodů – bitový řetězec (bitstream) je uložen v paměti SRAM uvnitř čipu a po odpojení napájení je nutné jej znovu do obvodu nahrát. Tento proces může v závislosti na velikosti obvodu tvořit významné zpoždění, se kterým je potřeba při návrhu systému počítat. Konfigurační

řetězec je obvykle načítán přes sériové rozhraní (SPI) z paměti FLASH nebo přes rozhraní JTAG přímo z počítače (při ladění obvodu). Novější rodiny obvodů již podporují i paralelní mód konfigurace, což do jisté míry problém zpoždění při startu řeší. Formát konfiguračního řetězce není až na výjimky znám.[2]

2.3.1 Vybrané rodiny obvodů FPGA

Za vznikem komplexních programovatelných logických obvodů stály především firmy Altera a Xilinx. V osmdesátých letech uvedla firma Altera svůj první obvod CPLD a firma Xilinx první FPGA. Od té doby se vývojem CPLD a FPGA obvodů zabývaly desítky firem, avšak pouze málo z nich dokázalo uspět, kromě uvedených firem Altera a Xilinx jmenujme firmy Lattice Semiconductor, Actel, Quick Logic a Atmel.

Produkty firem Xilinx a Altera pokrývají přes 80 % procent trhu, nejrozšířenějšími rodinami FPGA jsou řady Spartan (v nejnovější řadě nahrazena řadami Artix a Kintex) a Virtex od firmy Xilinx resp. Stratix, Cyclone a Arria od firmy Altera. Tabulka 2.1 nabízí srovnání těchto rodin, je zde uveden počet logických buněk, kapacita blokové paměti, počet DSP bloků (násobiček) či dostupné komunikační rozhraní PCI Express.

Tabulka 2.1: Srovnání vybraných rodin obvodů FPGA.

Xilinx	Virtex 5	Virtex 6	Virtex 7	Spartan 3	Spartan 6	Artix 7	Kintex 7
Počet LC[3]	25 920	760 000	2 000 000	5 968	150 000	215 000	480 000
Velikost BRAM	18,6 Mb	38 Mb	68 Mb	2,27 Mb	4,8 Mb	13 Mb	34 Mb
Počet DSP bloků	1 056	2 016	3 600	126	180	740	1 920
PCI Express	x8	x8	x8	x1	x1	x4	x8
Altera	Stratix III	Stratix IV	Stratix V	Cyclone IV	Cyclone V	Arria II	Arria V
Počet LE[3]	338 200	531 200	952 000	149 760	301 000	348 500	504 000
Velikost RAM	16,3 Mb	20,7 Mb	52,8 Mb	6,48 Mb	12,2 Mb	16,4 Mb	24,14 Mb
Počet násobiček	896	1 288	3 926	360	684	736	2 312
PCI Express	–	x4	x4	x1	x2	x1	x2

Vzhledem k velkým rozdílům ve struktuře logických bloků jsou tyto informace pouze přibližným ukazatelem složitosti a výkonnosti obvodů. Jednotlivé rodiny se zásadně liší architekturou logických bloků i propojovací sítě. Starší rodiny firmy Xilinx např. používaly 4-vstupé LUT, novější mají vstupů 6. S rostoucí velikostí LUT roste i zpoždění ze vstupu na výstup a je proto nutné volit tento parametr uvážlivě (novější technologie mají obvykle nižší zpoždění, proto počet vstupů pomalu roste).

Obvody FPGA jsou velmi často přímo výrobci nabízeny jako součást vývojové desky, existuje celá řada desek s obvody Spartan 3 a Spartan 6, u vyšších řad bývá menší výběr. Na Fakultě informačních technologií jsou dostupné např. desky Virtex-6 FPGA ML605 Evaluation Kit s obvodem XC6VLX240T[4] nebo Xilinx Kintex-7 FPGA KC705 Evaluation Kit s novějším XC7K325T[5].

Moderním trendem v oblasti obvodů FPGA je integrace rekonfigurovatelného logického pole a jednoho či více procesorů. Takový systém je pak nazýván SoC (System on Chip). Jako příklad uveďme obvody Zynq od firmy Xilinx, které integrují dvoujádrový procesor ARM Cortex-A9 a vybrané logické pole řady Artix-7 nebo Kintex-7. Na naší fakultě je

[2] Obvody Xilinx XC6200 mají známý formát konfiguračního řetězce, jsou však velmi jednoduché a dnes se již nevyrábí [17].

[3] Počet logických buněk (Logic Cells) v obvodech firmy Xilinx je možné přibližně porovnávat s počtem logických elementů (Logic Elements) v obvodech firmy Altera.

[4] 241 152 LC, 768 DSP bloků, 14,976 Mb blokové paměti [28]

[5] 326 080 LC, 840 DSP bloků, 16,02 Mb blokové paměti [24]

dostupná vývojová deska Xilinx Zynq-7000 All Programmable SoC ZC702 Evaluation Kit s obvodem XC7Z020[6].

2.4 Návrh číslicových obvodů v FPGA

V minulosti byly číslicové obvody navrhovány výhradně pomocí schématu. S rostoucí složitostí systémů se však tento způsob stával čím dál méně efektivní. Začaly proto vznikat specializované jazyky pro popis hardwaru (Hardware Description Language, HDL), obsahující speciální konstrukce zachycující paralelní činnost procesů v HW. V dnešní době opět dosáhla složitost navrhovaných systémů jistých mezí a HDL jazyky již často neposkytují dostatečnou abstrakci pro efektivní vývoj a verifikaci systémů. Objevují se jazyky vyšší úrovně, často specializované na nějaký typ systémů (např. jazyky pro popis architektury – Lisa, Codal apod.), ale i pro obecné použití (např. Catapult C).

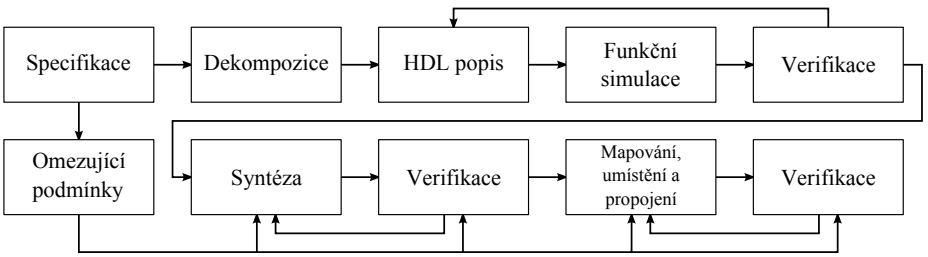
Nejpoužívanějšími jazyky HDL jsou VHDL[7] a Verilog. Tyto jazyky podporují celý proces návrhu číslicových obvodů, od samotného návrhu po verifikaci (viz obr. 2.5) [2]. Na začátku tohoto procesu je vždy *specifikace* – je nutné přesně vědět, co má navrhovaný obvod dělat a jaké má mít rozhraní. Specifikovaný problém je pak metodou *shora dolů* dekomponován na jednodušší celky a tyto jsou popsány nějakým HDL jazykem. Na základě HDL popisu je možné spustit *funkční simulaci* obvodu, k tomu slouží tzv. *test-benche* (zapouzdřují testovaný obvod, generují vstupní signály a případně verifikují výstupní signály).

Jestliže výsledky funkční simulace odpovídají specifikaci, je možné provést *syntézu* obvodu, tedy převod HDL popisu na úroveň hradel (tzv. netlist). Vstupem syntézy jsou také *omezující podmínky* (constraints) vytvořené uživatelem podle specifikace. Tyto omezující podmínky kladou požadavky na periodu hodinového signálu, zpoždění propojovací sítě, umístění obvodu na FPGA apod. Po skončení syntézy je nutné ověřit, že byly tyto podmínky splněny.

Po syntéze je nutné jednotlivé prvky obvodu namapovat na dostupné elementy konkrétní architektury, rozmístit je a propojit. Následuje opět verifikace omezujících podmínek a pokud nedojde k jejich splnění, je potřeba buď upravit navržený obvod nebo zvolit jinou cílovou architekturu. Výsledkem celého procesu je pak konfigurační řetězec, kterým je možné obvod FPGA nakonfigurovat, a simulační model na nejnižší úrovni abstrakce, pomocí něhož je možné systém věrně simulovat a verifikovat [17].

[6] 85 000 LC, 220 DSP bloků, 4,480 Mb blokové paměti [29]

[7] Zkratka VHDL pochází ze spojení Very High Speed Integrated Circuit HDL, VHSIC byl v 80. letech program amerického ministerstva obrany mající za cíl mimo jiné specifikovat jazyk pro popis číslicových (i analogových) obvodů, zejména kvůli verifikaci.



Obrázek 2.5: Obvyklý postup při návrhu číslicových obvodů pomocí jazýků HDL [2].

2.4.1 IP jádra

Jak bylo uvedeno v předchozí kapitole, rostoucí složitost číslicových obvodů vyžaduje zjednodušení návrhového procesu. Jednou z možností je využít jazyků vyšší úrovně abstrakce, druhou možností je dbát na znovupoužitelnost již odladěného kódu. Obdobou softwarových knihoven jsou v návrhu hardware tzv. IP jádra (Intellectual Property). Existují 3 typy IP jader:

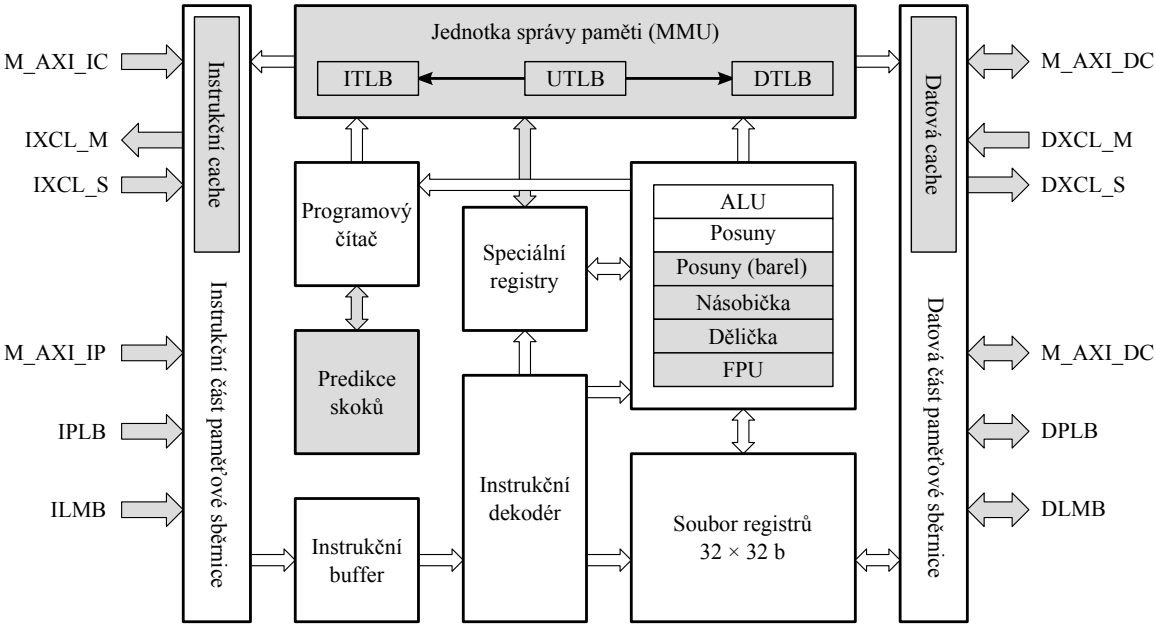
- Soft jádra – komponenta dostupná ve formě HDL popisu nezávislém na cílové architektuře.
- Firm jádra – syntetizovaná komponenta na úrovni hradel (netlist).
- Hard jádra – plně syntetizovaný obvod umístěný a propojený na konkrétním typu FPGA.

Tato jádra dodávají nejen samotní výrobci FPGA, ale i jiné technologické firmy, existují i jádra s otevřeným zdrojovým kódem.[8] Typickými příklady IP jader jsou periférie implementující komunikační protokoly, akcelerační jádra nebo i celé procesory. Společnost Xilinx dodává dva nejznámější soft procesory – 8-bitový PicoBlaze a 32-bitový MicroBlaze. PicoBlaze zabírá méně než 100 slice (v závislosti na konkrétní architektuře) a pracuje na frekvenci až 300 MHz, program je však nutné psát v jazyku symbolických instrukcí, neboť neexistuje předkladač z jazyka C. Mnohem větší možnosti nabízí soft procesor MicroBlaze.

Soft procesor MicroBlaze

Procesor MicroBlaze vyvíjený společností Xilinx je 32-bitový procesor s redukovanou instrukční sadou (RISC) s instrukční sadou velmi podobnou známé architektuře DLX. Instrukční linka procesoru je zřetězená, na výběr jsou dvě varianty architektury – 3-stupňová

[8] Např. komunitní server `<http://www.opencores.org>` nabízí tisíce soft jader.



Obrázek 2.6: Blokové schéma procesoru MicroBlaze (šedé komponenty jsou volitelné) [27].

linka (pomalejší, ale zabírá menší plochu čipu) a 5-stupňová (rychlejší, zabírá větší plochu). Blokové schéma procesoru MicroBlaze je možné vidět na obr. 2.6, kromě funkčních jednotek procesoru je zde patrné i napojení na různé datové a instrukční paměťové sběrnice. Dříve jako primární paměťová sběrnice sloužil standard PLB (Processor Local Bus) navržený firmou IBM. Novější rodiny FPGA (6. a 7. řada) již podporují modernější sběrnici AXI (Advanced eXtensible Interface), což je součást standardu AMBA (Advanced Microcontroller Bus Architecture) zavedený společností ARM pro stejnojmenné procesory. Přístup k lokální paměti (blokové) pak probíhá po sběrnici LMB (Local Memory Bus), aby byla redukována zátěž na ostatních sběrnicích.

MicroBlaze volitelně obsahuje jednotku správy paměti (Memory Management Unit, MMU) a umožňuje tak běh operačních systémů. Bez této jednotky je ale také možné využít např. real-time operačního systému FreeRTOS. Mezi další volitelné funkční jednotky patří Barrel Shifter (válcový posouvač), hardwarová násobička a dělička, jednotka výpočtů v plovoucí řádové čárce (FPU) nebo jednotka predikce skoků v programu (Branch Target Buffer).

Pro vývoj systémů využívajících procesoru MicroBlaze dodává firma Xilinx vývojové prostředí Embedded Development Kit (EDK). Konfigurace procesoru je prováděna pomocí programu Xilinx Platform Studio (XPS), samotný program je pak možné vyvíjet v integrovaném vývojovém prostředí SDK (Software Development Kit) založeném na známém prostředí Eclipse. Prostředí EDK také dovoluje vývojáři navrhnout vlastní periférie a připojit je k procesoru MicroBlaze pomocí dostupných sběrnic.

Kapitola 3

Evoluční algoritmy

Pojem evoluční algoritmus (EA) vznikl až počátkem 90. let, ačkoli techniky pod EA spadající sahají historicky až k polovině 20. století. Evoluční algoritmy zastřešují různé algoritmy inspirované evoluční teorií (zejména Darwinovou evoluční teorií, ale i neodarwinismem a dalšími teoriemi [6]), které vznikaly často nezávisle na sobě. Mezi nejvýznamnější patří genetické algoritmy, genetické programování, evoluční strategie, evoluční programování, diferenciální evoluce, gramatická evoluce a další [30].

Z pohledu teorie algoritmů se jedná o stochastické heuristické prohledávací algoritmy. Zpočátku byly využívány především pro optimalizaci, časem byly s úspěchem použity pro návrh systémů. Kromě optimalizace parametrů systému je tedy pomocí EA hledána samotná architektura systému [17].

Principem evolučních algoritmů je využití *populace* kandidátních řešení, která se postupně vyvíjí za využití operátorů inspirovaných biologickými procesy. Výpočet začíná vytvořením počáteční populace o stanoveném počtu *jedinců* (zakódovaných kandidátních řešení) a to buď náhodně nebo s využitím již známých řešení daného problému. Samotný výpočet je pak iterativní proces, kde v každé iteraci (podle biologické předlohy nazývané *generace*), je každému jedinci přiřazena hodnota *fitness* v závislosti na tom, jak dobré řešení problému představuje. Pomocí fitness je nad aktuální generací vytvořena relace uspořádání, „zdatnější“ jedinci mají vyšší fitness. Na základě tohoto uspořádání jsou vybráni jedinci, kteří budou rodiči jedinců v následující generaci (tzv. *selekce*). Potomci vzniknou aplikací genetických operátorů (*křížení*, *mutace*, apod.) na vybrané rodiče. Jedinci v nové generaci jsou vybráni z rodičů, jejich potomků a případně z předchozích generací.

Jednotlivé evoluční algoritmy se liší zejména ve způsobu výběru rodičů, použitých genetických operátorech a výběru jedinců do nové generace. Zpravidla však využívají hodnotu fitness, neboť bez této informace by prohledávání stavového prostoru řešení bylo náhodné. Vytvořením relace uspořádání nad populací jedinců vzniká selekční tlak a průměrná fitness populace s generacemi roste.

Úspěšnost evoluce kromě charakteru fitness funkce, způsobu selekce a zvolených genetických operátorů ovlivňuje také způsob zakódování problému. Při dřívějším využití EA jako optimalizačních algoritmů tvořily optimalizované parametry přímo kandidátní řešení. Pro návrh architektury systému s využitím EA je nutné požadovaný systém vhodně reprezentovat. Přitom platí, že příliš obecný popis zbytečně zvětšuje stavový prostor, zatímco příliš mnoho omezujících podmínek nedává evoluci dostatečný prostor pro nalezení inovativního řešení.

EA pracují nad datovými strukturami (nazývanými *chromozomy*), které reprezentují zakódované kandidátní řešení problému. Každý chromozom se skládá z jednoho a více *genů*.

Někdy se využívá také pojmů *genotyp* a *fenotyp*. Genotyp v EA je ekvivalentní chromozomu[1] a fenotyp je souhrn všech znaků, vlastností a projevů jedince. Obecně platí (tak jako v přírodě), že zobrazení genotypů na fenotypy není injektivní. Jak bude ukázáno později, existence jedinců s různým genotypem, ale stejným fenotypem (a tedy i hodnotou fitness) hraje velkou roli při fixování důležitých znaků populace [17].

3.1 Genetické algoritmy

Historie genetických algoritmů (GA) sahá až do sedmdesátých let, nicméně větší oblibě se začaly těšit až na konci let osmdesátých [17]. Používají se převážně pro optimalizační úlohy. GA vychází z obecné struktury evolučního algoritmu, je však charakteristický reprezentací kandidátních řešení ve formě řetězců konstantní délky. Chromozom G je tedy ve tvaru $G = (g_1, g_2, \ldots, g_L)$, kde $g_i \in A_i$, $A_i = \{a_i^1, a_i^2, \ldots, a_i^{N_i}\}$ jsou jednotlivé geny chromozomu, L je jeho délka a a_i^j jsou varianty genu g_i (*alely*). Zpravidla $N_i = N$ a $A_i = A$ pro všechna i a tedy všechny geny mají stejné alely. Nejčastěji bývá množina alel A dána intervalem celočíselných hodnot nebo nabývají geny pouze jedniček a nul (binární chromozom) [14]. Některé typy úloh vyžadují složitější způsob reprezentace, např. *permutační zakódování*, které zaručuje jedinečnost alel v rámci chromozomu.

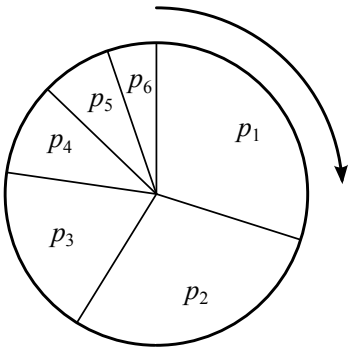
Počáteční populace je vždy vytvořena náhodnou inicializací chromozomů, v každé generaci je pak provedena evaluace populace (vyhodnocení fitness každého jedince populace) a pomocí selekčních operátorů vybrána množina jedinců, kteří se stanou rodiči nové generace. Selekce je vždy prováděna na základě fitness jedinců, existuje několik způsobů používaných v GA. Nejjednodušším způsobem je *deterministická selekce*, kdy jsou jedinci seřazeni podle fitness a prvních K jedinců je vybráno k reprodukci.

Prvek náhody zavádí *proporciální selekce*, která každému jedinci přiřazuje pravděpodobnost p_i, že bude vybrán do následující generace, podle jeho fitness:

$$p_i = \frac{f_i}{\sum_{j=1}^{P} f_j}, \tag{3.1}$$

kde f_i je fitness jedince i a P je počet jedinců v populaci. Tento způsob výběru je v praxi implementován pomocí algoritmu *rulety*, čím větší má jedinec fitness, tím větší kruhovou výseč rulety zabírá. Roztočením rulety dojde k náhodnému výběru jedince (viz obr. 3.1).

[1] Většinou je použit pouze jeden chromozom pro reprezentaci jedince, za určitých podmínek však může být výhodné využít více chromozomů. Např. pro řešení dynamických úloh je vhodná polyploidní, nejčastější diploidní, reprezentace.



Obrázek 3.1: Selekce pomocí algoritmu rulety.

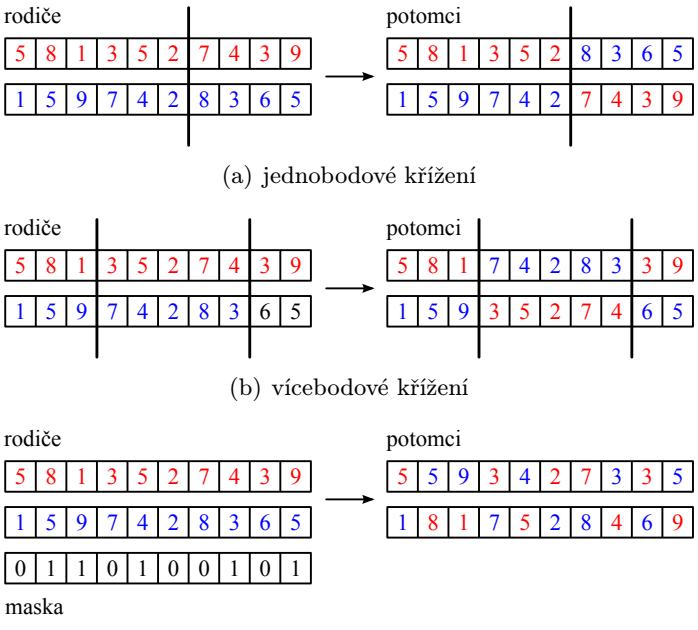
Nevýhoda proporciální selekce se projeví, pokud se v populaci vyskytne jedinec s velmi vysokou hodnotou fitness. Pak je velmi nepravděpodobné, že bude vybrán jiný jedinec a populace zdegeneruje. Řešením může být *výběr podle pořadí*, kdy se jedinci seřadí podle fitness, ale pravděpodobnost jejich výběru je úměrná pouze pořadí, nikoliv fitness.

Dalším používaným způsobem selekce je tzv. *turnajová selekce*, kdy je z aktuální populace vybráno vždy náhodně několik jedinců a ten s nejvyšší hodnotou fitness vyhrává turnaj a stává se z něj rodič. Tento mechanismus je opakován tolikrát, kolik je potřeba rodičů.

U vybraných jedinců je následně procesem křížení kombinována genetická informace a vznikají potomci. Základní možnosti provedení křížení jedinců v GA jsou ilustrovány na obr. 3.2. Při *jednobodovém křížení* je náhodně vygenerován bod křížení, první část chromozomu je zachována, v části za bodem křížení dojde k výměně alel mezi rodiči. Vzniknou tak dva noví jedinci (potomci). Princip *vícebodového křížení* je velmi podobný, genetický materiál je střídavě ponechán resp. vyměněn mezi jednotlivými body křížení. *Uniformní křížení* pak spočívá v náhodném vygenerování binární masky o stejné délce, jakou má chromozom, a prohozením těch genů, pro které odpovídající pozice v masce nabývá jedničky.

Mutace jedinců v GA jsou velmi jednoduché, náhodný počet genů je buď nahrazen náhodně vygenerovanými alelami, nebo je modifikován např. přičtením náhodného čísla. U binárních chromozomů je pak náhodný počet genů invertován.

Nově vytvoření jedinci tvoří tzv. *mezigeneraci*, v případě úplného nahrazení původní populace potomky hovoříme o *generační variantě* GA, pokud je nová populace tvořena zčásti i jedinci předchozí populace, dochází k *překrývání populací*. Často z důvodu zvýšení selekčního tlaku zavádíme *elitismus*, kdy nejlepší jedinec společně s několika jeho mutanty vždy postoupí do další generace [17, 14]. Evoluce probíhá buď stanovený počet generací nebo je ukončena při dosažení dostatečně kvalitního řešení.



(a) jednobodové křížení

(b) vícebodové křížení

(c) uniformní křížení

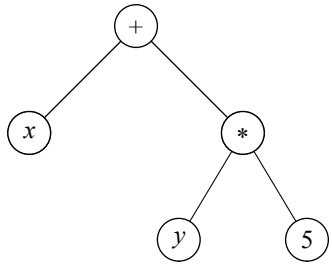
Obrázek 3.2: Způsoby křížení používané v genetických algoritmech.

3.2 Genetické programování

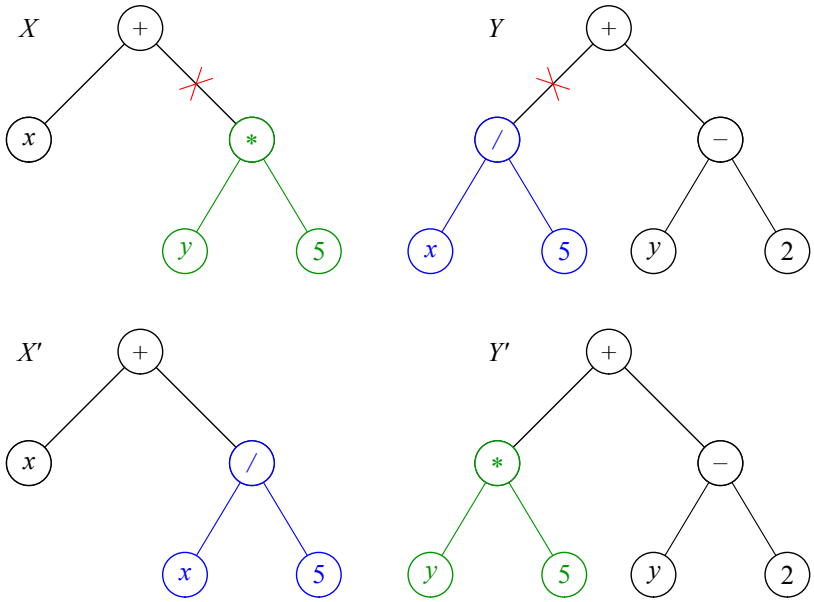
Základním specifikem genetického programování (GP) je způsob reprezentace problému. Roku 1985 navrhl Nichael L. Cramer, pracovník společnosti Texas Instruments, řešení problému generování krátkých programů pomocí funkcionální reprezentace a využitím principů genetických algoritmů [3]. Kandidátní program byl tvořen stromovou datovou strukturou, jejíž uzly obsahovaly jednoduché funkce (sčítání, násobení konstantou apod.) a uzly byly buď konstanty nebo vstupy programu (viz obr. 3.3). Jeho práci později rozšířil John R. Koza, který využil pro reprezentaci programu funkcionálního jazyka LISP. Řešil tak úlohu symbolické regrese.

Princip metody je velmi podobný jako u genetických algoritmů. Nejprve je náhodně vygenerována počáteční populace. Ohodnocení jedince v případě GP spočívá ve spuštění programu, jímž je reprezentován. Takto je získána hodnota, která je porovnána s referenční (požadovanou) hodnotou. Opakovaným spuštěním programu pro různé vstupní hodnoty ověřujeme funkci programu, zvolením vhodné fitness funkce pak získáme výslednou hodnotu fitness.

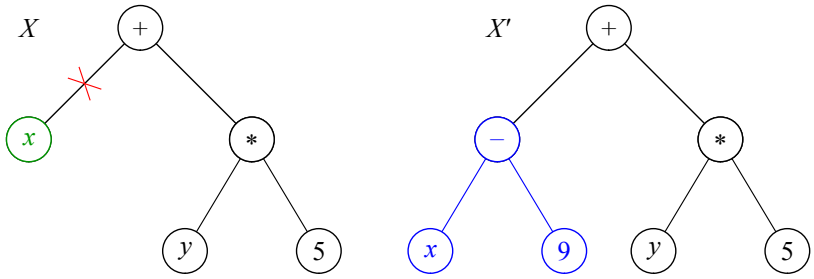
Proces selekce probíhá na stejných principech jako v případě GA. Vzhledem ke specifické reprezentaci jedinců se v GP používají speciální operátory křížení a mutace. Proces křížení



Obrázek 3.3: Program v GP je reprezentován formou syntaktického stromu.



Obrázek 3.4: Křížením v GP dojde k prohození náhodně zvolených podstromů rodičů.



Obrázek 3.5: Mutace v GP – náhodně vybraný podstrom je nahrazen novým, náhodně vygenerovaným podstromem.

spočívá v prohození náhodně vybraných podstromů obou rodičů (viz obr. 3.4), vzniknou tak dva noví jedinci.

Mutace jedinců v GP nastávají pouze velmi zřídka. S určitou velmi malou pravěpodobností je vybrán jedinec, jehož náhodně zvolený podstrom je nahrazen novým, tentokrát náhodně vygenerovaným, podstromem (viz obr. 3.5). Pravděpodobnost mutace je nutné uvážlivě volit, aby nedocházelo k degeneracím jedinců s vysokou fitness. Tak jako v přírodě je však mutace velmi důležitým mechanismem stojícím za „invencí“, kterou evoluce vytváří nové (lepší) jedince.

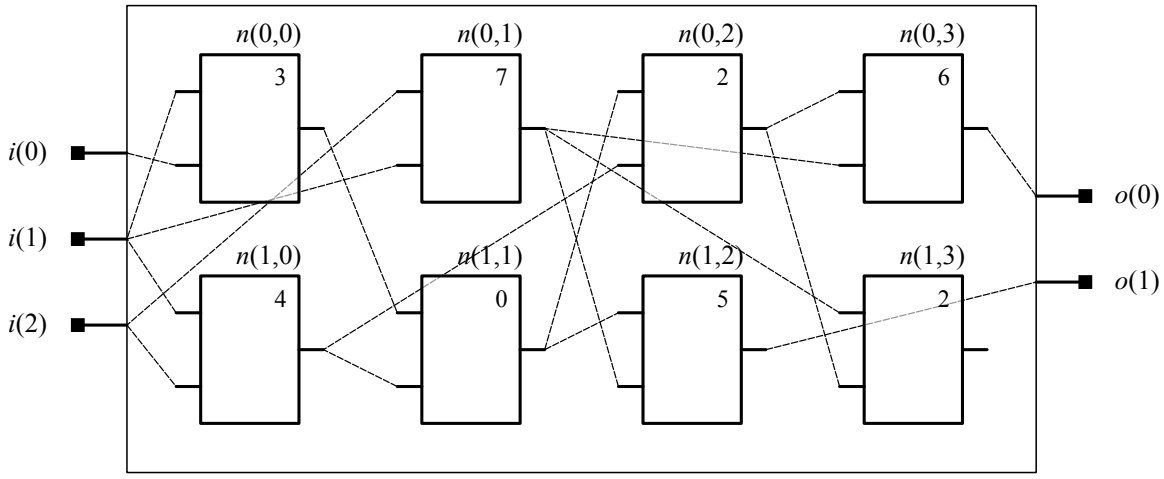
Genetické programování je s úspěchem používáno pro řešení úlohy symbolické regrese, kdy se snažíme pro data získaná experimentálním měřením najít matematický vztah, který je s požadovanou přesností aproximuje. Dalšími aplikacemi jsou např. syntéza struktury PID regulátorů, zesilovačů, hradla NAND apod [12].

Hodnota fitness je v GP často počítána pro nějakou množinu *případů fitness* (fitness cases). Případ fitness odpovídá vybrané situaci, kterou má kandidátní program za úkol vyřešit. Skládá se ze vstupů programu a očekávaného výstupu, který by ideální program měl vracet při daných vstupních hodnotách. V některých aplikacích však množina případů fitness obsahuje vzájemně protikladné případy, kdy pro stejné vstupní hodnoty očekáváme v různých případech fitness různé výstupní hodnoty. Při řešení takových problémů často požadujeme, aby kandidátní program vhodně *generalizoval* případy fitness a aby byla funkce programu zachována i pro případy fitness, které nebyly použity při evoluci.

3.3 Kartézské genetické programování

Speciální variantou genetického programování je *kartézské genetické programování* (CGP) [18]. Oproti GP pracuje s orientovaným grafem, kde uzly jsou uspořádány v pravoúhlé mřížce $n_c \times n_r$ (viz obr. 3.6). Obvykle bývá používán acyklický graf, je ale možné využít i obecných orientovaných grafů, např. při návrhu sekvenčních obvodů. V této práci bude uvažován vždy graf acyklický.

Každý uzel $n(r, c)$, $r \in \{0, \ldots, n_r - 1\}$, $c \in \{0, \ldots, n_c - 1\}$, má n_a vstupů a jeden výstup a jeho funkce je konfigurovatelná. Parametrizovatelné je i propojení uzlů, vstup každého uzlu může být připojen na libovolný vstup $i(k)$, $k \in \{0, \ldots, n_i - 1\}$, a na výstupy uzlů v předchozích sloupcích. Pro omezení stavového prostoru úlohy je možné omezit připojení vzdálenějších sloupců. Počet předchozích sloupců uzlů, ke kterým je možné vstup uzlu v aktuálním sloupci připojit, se nazývá parametr l-back. Pokud $l = 1$, je dovoleno propojovat pouze sousední sloupce a stavový prostor je poměrně malý. Naopak při $l = n_c - 1$ je propojení neomezené, evoluce však může trvat poměrně dlouho.



Obrázek 3.6: Schéma reprezentace programu v CGP.

Chromozom reprezentující jedince v CGP je tvořen A_{CGP} celočíselnými hodnotami:

$$A_{\mathrm{CGP}} = n_{\mathrm{r}} \cdot n_{\mathrm{c}} \cdot (n_{\mathrm{a}} + 1) + n_{\mathrm{o}}, \tag{3.2}$$

kde n_{r} je počet řádků, n_{c} počet sloupců, n_{a} počet vstupů uzlu a n_{o} počet výstupů grafu. Primární vstupy jsou označeny čísly $0 \ldots n_{\mathrm{i}} - 1$, výstupy uzlů pak čísly $n_{\mathrm{i}} + c \cdot n_{\mathrm{r}} + r$. Každý uzel je reprezentován n_{a} indexy uzlů, ke kterým jsou jeho vstupy připojeny, a jedním číslem určujícím funkci uzlu. Součástí chromozomu je pak ještě n_{o} indexů výstupních uzlů.

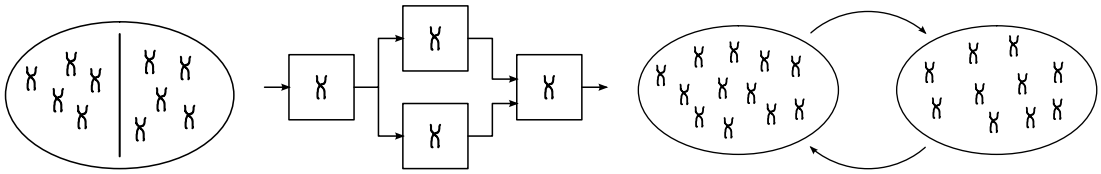
Je zřejmé, že díky uvedenému způsobu reprezentace se nemusí všechny uzly podílet na výstupních hodnotách a zároveň některé primární vstupy nemusí být využity. Oproti GP se v kartézském genetickém programování používá obvykle jediný genetický operátor – mutace. S určitou pravděpodobností je vybrán gen a ten je náhodně pozměněn. Mutací může být změněno propojení uzlů i funkce uzlu.

V každé generaci je vybrán jedinec s nejvyšší hodnotou fitness a nová populace je vytvořena mutací tohoto jediného rodiče. Tato varianta generování populací se označuje jako evoluční strategie (ES) $(1 + \lambda)$. Pokud je v populaci více jedinců s nejvyšší fitness, je vždy vybrán ten, který nebyl rodičem poslední generace (pokud takový existuje). Stejně kvalitní jedinci vznikají *neutrální mutací* rodiče a výměna rodiče pro následující generaci zajišťuje genetickou diverzitu a částečně zabraňuje uváznutí evoluce v lokálním extrému [17].

Operátor křížení se v CGP většinou nepoužívá, bylo ukázáno, že jeho použití např. při návrhu číslicových obvodů není vhodné [17]. Aplikace kartézského genetického programování zahrnují kromě návrhu kombinačních logických obvodů také návrh obrazových filtrů, strojové učení, klasifikaci apod.

3.4 Koevoluční algoritmy

Dosud uvedené evoluční algoritmy prováděly evoluci jediné populace jedinců stejného druhu. V přírodě však probíhá evoluce mnoha organismů souběžně a jedinci různých populací se navzájem ovlivňují. Nejběžnějším případem může být dravec, který se snaží ulovit kořist a posupně zlepšuje své schopnosti (obratnost, rychlost, výdrž apod.). Působí tak selekčním tlakem na populaci kořisti, jejíž schopnosti se také zlepšují. Tato uzavřená smyčka urychluje vývoj jednotlivých druhů a podporuje vznik účelných vlastností. Algoritmy, které jsou



(a) (b) (c)

Obrázek 3.7: Schéma (a) populace rozdělené bariérou, (b) kompoziční úlohy a (c) úlohy založené na testu.

principem souběžné evoluce v přírodě inspirovány, se nazývají *koevoluční*.

3.4.1 Klasifikace koevolučních algoritmů

Koevoluční algoritmy mohou využívat buď populaci jednoho druhu nebo více populací různých druhů. V případě jediné populace jsou jedinci odděleni *bariérou*, evoluce probíhá většinu času nezávisle na obou stranách bariéry, místy však dojde k interakci jedinců z obou stran bariéry (např. pokud evoluce uvázne v lokálním extrému).

Mnohdy může být užitečné řešit problém koevolucí více populací jedinců různých druhů. Řadu úloh je možné dekompozicí rozložit na jednodušší podúlohy, každou podúlohu je pak možné reprezentovat samostatným chromozomem. Základní varianty koevolučních algoritmů je možné vidět na obr. 3.7. Takové úlohy se nazývají *kompoziční* a koevoluce má *kooperativní* charakter, protože jednotliví jedinci společně tvoří řešení problému a jeho kvalita je tak dána kvalitou všech jednotlivých jedinců.

Jinou variantou koevolučních algoritmů využívajících více populací jsou *úlohy založené na testu*. Principem těchto úloh je testování jedinců jedné populace jedinci druhé populace. Standardní evoluční algoritmus je tedy rozšířen o evoluci původně statického výpočtu fitness. Tyto dvě nezávislé evoluce probíhají paralelně a to buď synchronně (po určitém počtu generací dojde k interakci mezi populacemi) nebo asynchronně [20, 21, 10, 16].

3.4.2 Vlastnosti koevolučních algoritmů

Vzhledem k interakcím jedinců různých populací a jejich souběžné evoluci u koevolučních algoritmů většinou není jednoznačně určena hodnota fitness. Např. u úloh založených na testu se spolu s řešením problému vyvíjí i ohodnocovače a vývoj fitness v čase nemusí být monotonní.

Zavedením koevoluce se obvykle snažíme urychlit konvergenci fitness funkce a zajistit stabilitu evoluce. Nedochází pak k uváznutím v lokálních optimech, jako je tomu často u evolučních algoritmů. Při návrhu kombinačních obvodů je evoluce úspěšná pouze pokud najde takové řešení, které pro všechny vstupní vektory dává správnou výstupní hodnotu. Pokud evoluce uvázne v lokálním optimu, nepodaří se ve stanoveném počtu generací nalézt řešení problému. Při použití koevoluce je v některých případech pravděpodobnost úspěšného běhu vyšší, rozptyl hodnot fitness v nezávislých bězích je nižší [21].

Kapitola 4

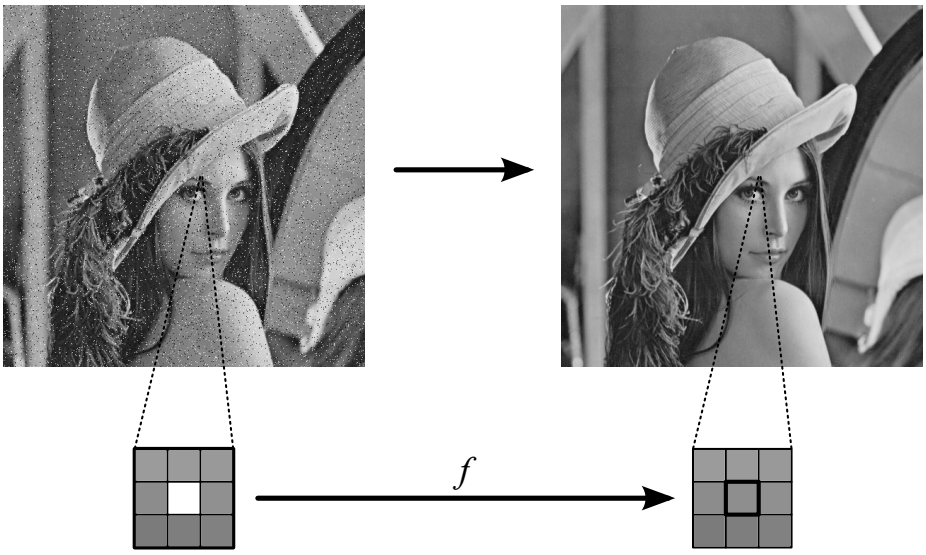
Obrazové filtry

Obrazové filtry slouží ke zpracování digitálních obrazových dat, nejčastěji k odstranění šumu, detekci hran, zaostřování apod. Jsou součástí mnoha vestavěných zařízení a tvoří obvykle první článek složitějšího řetězce zpracování obrazu. Schéma na obr. 4.1 ilustruje obvyklý proces filtrace, každý pixel filtrovaného obrazu je určen funkcí $f : \mathbb{R}^N \to \mathbb{R}$, kde N je počet pixelů tvořících okolí (obykle 9-okolí popř. 25-okolí).

Podle vlastností funkce f lze obrazové filtry rozdělit na *lineární* a *nelineární*. Pro lineární filtry platí pricip superpozice, výsledný obraz je dán konvolucí původního obrazu a konvolučního jádra (impulzní odezvy). Lineárními filtry však nelze odstranit některé druhy šumu, obecně je nelze použít, pokud má poškození obrazu jiný než aditivní charakter.

4.1 Konvenční filtry

Konvenční filtry určené k odstranění aditivního šumu (nejčastěji bílého) mají charakter dolní propusti. Příkladem může být *Gaussovský filtr*, jehož impulzní odezva je dána dvou-



Obrázek 4.1: Schéma digitální filtrace obrazu.

(a) zašuměný obraz (b) $\sigma = 0.5$ (c) $\sigma = 5$

Obrázek 4.2: Filtrace obrazu poškozeného bílým šumem: (a) zašuměný obraz, (b) a (c) výsledky filtrace pro různě nastavené Gaussovské filtry.

rozměrnou Gaussovou funkcí:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \mathrm{e}^{-\frac{x^2+y^2}{2\sigma^2}}, \tag{4.1}$$

kde σ je směrodatná odchylka a pro správnou funkci filtru by měla odpovídat intenzitě šumu. Výsledkem filtrace je nejen odstranění šumu, ale také rozmazání obrazu – vyšší prostorové frekvence tvořící detaily obrazu jsou potlačeny. Příklad filtrace pomocí Gaussovského filtru je vidět na obr. 4.2, je zde patrné, že nastavením nízkého σ nedojde k dostatečnému potlačení šumu, zatímco při příliš vysokém σ ztrácí obraz ostré hrany [7].

Pro odstranění šumu s jiným než aditivním charakterem slouží filtry nelineární. Příkladem takového filtru je *mediánový filtr*, který z daného okolí pixelu určí medián. Tento filtr se hodí pro odstranění impuzního šumu (např. výstřelový šum nebo šum typu sůl a pepř). Nevýhodou mediánového filtru je absence detailů ve filtrovaném obraze (viz obr. 4.3).

Další aplikací lineárních filtrů je detekce hran, cílem je nalezení pixelů v obraze, kde se výrazně mění jas. Taková místa v obraze mají vysokou první derivaci, nejvyšší hodnota derivace je pak ve směru kolmém na hranu, v praxi se hrany detekují pouze v některých směrech (horizontálním a vertikálním). Pro aproximaci směrových derivací se obvykle používají malá konvoluční jádra, nejjednodušší taková jádra mají pouze dva prvky: $\begin{pmatrix} -1 & 1 \end{pmatrix}$

(a) zašuměný obraz (b) medián 3×3 (c) medián 5×5

Obrázek 4.3: Filtrace šumu typu sůl a pepř mediánovým filtrem.

(a) původní obraz (b) po aplikaci Sobelova operátoru (c) po prahování

Obrázek 4.4: Detekce hran pomocí Sobelova operátoru.

resp. $\begin{pmatrix}-1 & 1\end{pmatrix}^{\mathrm{T}}$. Nejznámějším operátorem pro detekci hran je *Sobelův operátor*:

$$G_{\mathrm{x}} = \begin{pmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{pmatrix}, \quad G_{\mathrm{y}} = \begin{pmatrix}-1 & -2 & -1\\ 0 & 0 & 0\\ 1 & 2 & 1\end{pmatrix}, \tag{4.2}$$

kde G_{x} detekuje hrany v horizontálním směru a G_{y} ve vertikálním směru. Kombinací těchto dvou operátorů je možné vytvořit nelineární filtr:

$$G = \sqrt{(G_{\mathrm{x}} * I)^2 + (G_{\mathrm{y}} * I)^2}, \tag{4.3}$$

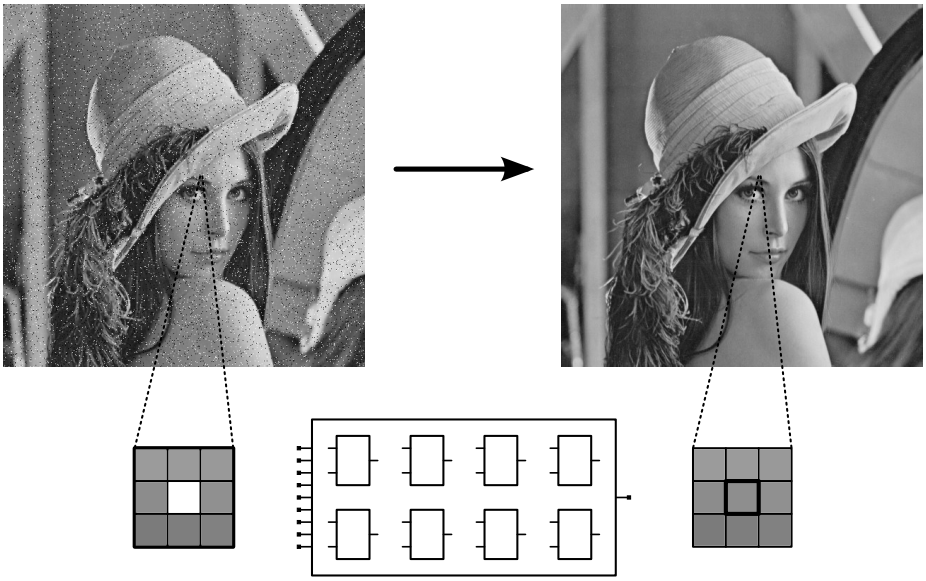
kde I je původní obraz a operace $*$ značí konvoluci [7]. Výsledek aplikace Sobelova operátoru je vidět na obr. 4.4. Obvykle bývá Sobelův operátor doplněn prahováním získaného obrazu, jehož výsledkem je binární obraz s jedničkami v místech detekovaných hran.

V případě, že obraz obsahuje velké množství šumu, může být detekce hran pomocí Sobelova operátoru nepřesná. Lepších výsledků pak lze dosáhnout např. pomocí *Cannyho hranového detektoru*, který obraz nejprve filtruje Gaussovským filtrem a následně aplikuje např. Sobelův operátor pro detekci hran. Aby byly detekované hrany co nejtenčí, jsou nalezena lokální maxima. Nakonec je provedeno prahování s hysterezí, jsou stanoveny dva prahy $T_1 < T_2$, pixely s hodnotou vyšší než T_2 jsou přímo označeny za hrany, pixely s hodnotou mezi T_1 a T_2 jsou označeny za hrany, pokud sousedí s pixelem, který již byl za hranu označen dříve [1]. Vyšší úspěšnost detekce hran pomocí Cannyho detektoru je vykoupena vysokou výpočetní náročností.

4.2 Návrh pomocí evolučních algoritmů

Filtry navržené tradičními metodami fungují na základě principů, které lze matematicky popsat a vysvětlit. Za určitých podmínek, např. při velmi vysoké intenzitě šumu, však filtrace nedosahuje uspokojivých kvalit. Pomocí evolučních algoritmů je možné navrhnout filtry, které za těchto podmínek pracují výrazně spolehlivěji. Jejich architektura není omezena tak jako u konvenčních filtrů systematičností inženýrského návrhu a princip jejich fungování nemusí být zřejmý.

K návrhu obrazových filtrů lze využít různé evoluční algoritmy, jako vhodné pro tuto úlohu se ale jeví použití kartézského genetického programování (viz obr. 4.5) [18]. Zvolené okolí každého pixelu je přivedeno na vstup kandidátního filtru, jediný výstup filtru



Obrázek 4.5: Návrh obrazových filtrů pomocí CGP.

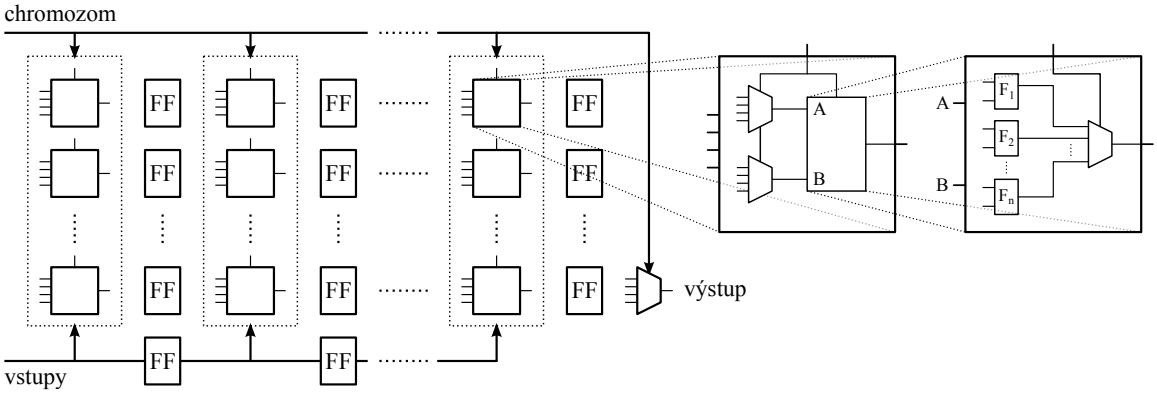
je pak vyfiltrovaný pixel. Jednotlivé bloky uvnitř filtru plní jednoduché funkce, příklady takovýchto funkcí uvádí tabulka 4.1.

Tabulka 4.1: Příklad použitých funkcí bloků při návrhu obrazových filtrů pomocí CGP [20].

#	funkce	#	funkce
0	255	8	$i_1 \gg 1$
1	i_1	9	$i_1 \gg 2$
2	$\neg i_2$	10	$(i_1 \ll 4) \vee (i_2 \gg 4)$
3	$i_1 \vee i_2$	11	$i_1 + i_2$
4	$\neg i_1 \vee i_2$	12	$i_1 +^s i_2$
5	$i_1 \wedge i_2$	13	$(i_1 + i_2) \gg 1$
6	$\neg(i_1 \wedge i_2)$	14	$\max(i_1, i_2)$
7	$i_1 \oplus i_2$	15	$\min(i_1, i_2)$

Samotná evoluce může probíhat na různých výpočetních prostředcích. Obrazové filtry jsou často implementovány pomocí číslicových obvodů, proto se nabízí využití rekonfigurovatelných obvodů (FPGA). Existuje několik možností, jak rekonfigurovatelný obvod při evoluci využít, řízení evoluce a výpočet hodnoty fitness můžou být umístěny buď přímo na FPGA (např. s využitím soft procesoru) nebo i samostatně např. na DSP. Některé typy FPGA umožňují tzv. *dynamickou rekonfiguraci*, pomocí speciálního rozhraní (ICAP u zařízení od firmy Xilinx) je možné za běhu rekonfigurovat část FPGA. Formát bitového řetězce však až na výjimky není znám, a proto je využití dynamické rekonfigurace v evoluci nesnadné (nikoli však nemožné [4]). Jiným řešením může být *virtuální rekonfigurovatelný obvod*.

Virtuální rekonfigurovatelný obvod (VRC) je číslicový obvod kopírující architekturu kartézského genetického programování [17]. Tvoří jej pole $n_c \times n_r$ rekonfigurovatelných elementů, každý z elementů obsahuje multiplexory pro výběr vstupních signálů a funkční



Obrázek 4.6: Schéma virtuálního rekonfigurovatelného obvodu.

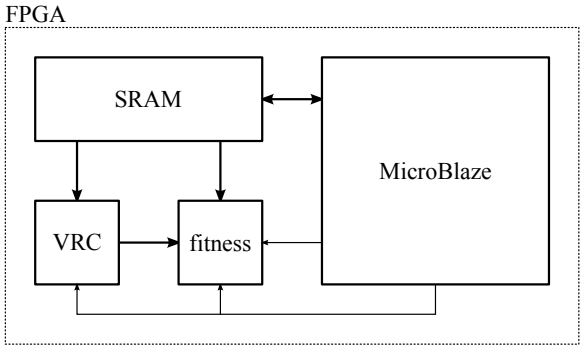
jednotku realizující zvolené funkce. Propojení bloků a výběr funkcí bloků je dán chromozomem, výstupní hodnota celého obvodu je vybrána multiplexorem z výstupů všech elementů.

Blokové schéma virtuálního rekonfigurovatelného obvodu a detail rekonfigurovatelného elementu je na obr. 4.6. Výstupy elementů v každém sloupci a vstupní signály jsou přivedeny na registry, celý obvod tedy může pracovat *zřetězeně*. Propojením VRC s jednotkou počítající hodnotu fitness tak lze velmi efektivně ohodnotit populaci kandidátních filtrů. Vstupní pixely jsou čteny z paměti SRAM společně s referenční hodnotou, která je přivedena přímo do fitness jednotky. Celý obvod je řízen např. soft procesorem MicroBlaze, který má na starosti především generování jedinců (viz obr. 4.7).

Důležitou roli hraje také výběr fitness funkce, mezi nejvíce používané patří *střední kvadratická chyba* (Mean Squared Error, MSE) a *střední odchylka pixelů* (Mean Difference Per Pixel, MDPP):

$$\begin{aligned} \mathrm{MSE}(x,y) &= \frac{1}{R \cdot C} \sum_{r=1}^{R} \sum_{c=1}^{C} \left(x(r,c) - y(r,c)\right)^2, \\ \mathrm{MDPP}(x,y) &= \frac{1}{R \cdot C} \sum_{r=1}^{R} \sum_{c=1}^{C} \left|x(r,c) - y(r,c)\right|, \end{aligned} \tag{4.4}$$

kde R je počet řádků a C počet sloupců trénovacího obrázku. Často se také můžeme setkat s hodnotou PSNR (Peak Signal to Noise Ratio) udávající odstup signálu od šumu v decibelech. Existují i metriky založené na subjektivním vjemu obrazu lidským mozkem,



Obrázek 4.7: Zapojení VRC, jednotky fitness, paměti a řídicího procesoru.

např. SSIM (Structural Similarity), jejich výpočet je však mnohem složitější a obvykle jejich použití nepřináší žádné výhody.

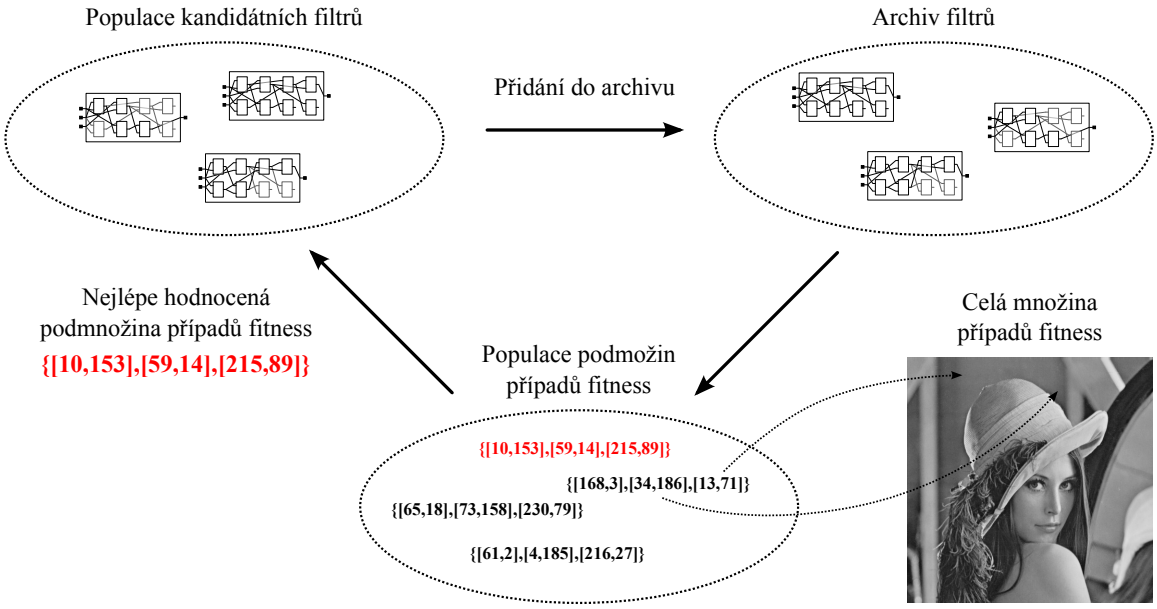
S využitím evoluce je možné navrhnout obrazové filtry, které pracují za určitých podmínek lépe nebo zabírají oproti konvenčním filtrům výrazně nižší plochu na čipu. Na základě požadované vlastnosti navrhovaného filtru je zvolena trénovací množina dat (případy fitness) a vhodná fitness funkce. Vlastní proces evoluce může být stejný pro libovolný typ filtru.

4.3 Návrh pomocí koevolučních algoritmů

Přestože lze návrh obrazových filtrů velmi urychlit pomocí virtuálního rekonfigurovatelného obvodu, počet generací potřebný pro dosažení kvalitního výsledku je v řádu desetitisíců. V článku [20] bylo ukázáno, že s využitím koevolučních algoritmů lze celý proces ještě více urychlit, a sice zmenšením počtu vyhodnocení případů fitness. Schéma na obr. 4.8 ilustruje princip koevoluce v CGP, populace kandidátních filtrů je ohodnocena pouze pomocí podmožiny případů fitness (fitness cases subset, FCS), tedy pouze vybranými pixely trénovacího obrazu. Filtry z různých generací jsou vkládány do *archivu* filtrů, který slouží k ohodnocení populace FCS. Oproti standardnímu CGP tedy není nutné počítat fitness kandidátního filtru z celé množiny případů fitness, ale pouze podmožiny. Hodnota částečné fitness je tedy dána:

$$\begin{aligned} \mathrm{MSE_p}(x,y,s) &= \frac{1}{S}\sum_{i=1}^{S}\left(x(s(i)) - y(s(i))\right)^2, \\ \mathrm{MDPP_p}(x,y,s) &= \frac{1}{S}\sum_{i=1}^{S}\left|x(s(i)) - y(s(i))\right|, \end{aligned} \tag{4.5}$$

kde $s \subset \{1,\ldots,R\}\times\{1,\ldots,C\}$ je podmnožina případů fitness (indexy v kompletní množině případů fitness) a S je počet prvků této podmožiny. Tato hodnota je závislá na aktuální



Obrázek 4.8: Schéma koevoluce v CGP [21].

FCS a její vývoj s generacemi nemusí být monotónní.

Ohodnocení populace FCS je možné provést dvěma způsoby. Prvním způsobem je určení fitness každého jedince na základě rozdílu hodnot částečné a skutečné fitness (s použitím všech případů fitness) filtrů v archivu. Tímto způsobem je zajištěno, že vybraná podmožina věrně modeluje celou množinu případů fitness, přístup je označován jako *koevoluce prediktorů fitness*. Nevýhodou však je, že je nutné znát skutečnou fitness každého kandidátního filtru v archivu pro celou množinu případů fitness. Tuto nevýhodu lze eliminovat použitím přístupu na principu *soutěživé koevoluce*, kdy jsou upřednostňovány takové FCS, pro které filtry v archivu dosahují co nejhorší fitness (tedy průměrná částečná fitness $\mathrm{MSE_p}$ nebo $\mathrm{MDPP_p}$ filtrů z archivu je co nejvyšší).

Evoluce FCS je založena na jednoduchém genetickém algoritmu, FCS jsou reprezentovány polem čísel s pevnou délkou, oproti CGP je využívána mutace i křížení (jednobodová výměna částí chromozomů rodičů). V každé generaci je navíc kvůli zajištění genetické diverzity určitý počet jedinců náhodně vygenerován [21].

Kapitola 5

Návrh a implementace akcelerační jednotky

Pro dosažení kvalitních výsledků při (ko)evolučním návrhu obrazových filtrů je obvykle nutné vyhodnotit desetitisíce generací, což je výpočetně velmi náročné. Tato kapitola se zabývá návrhem a implementací akcelerační jednotky pro urychlení tohoto procesu.

Úloha koevoluce je tvořena dvěma nezávislými procesy, které spolu musí průběžně komunikovat. Kritickou částí evoučního procesu z hlediska výpočetní náročnosti je vyhodnocení fitness každého kandidátního programu v každé generaci. Ostatní výpočty zahrnující zejména generování nové populace oproti tomu tvoří pouze malou režii a nejsou tedy předmětem akcelerace.

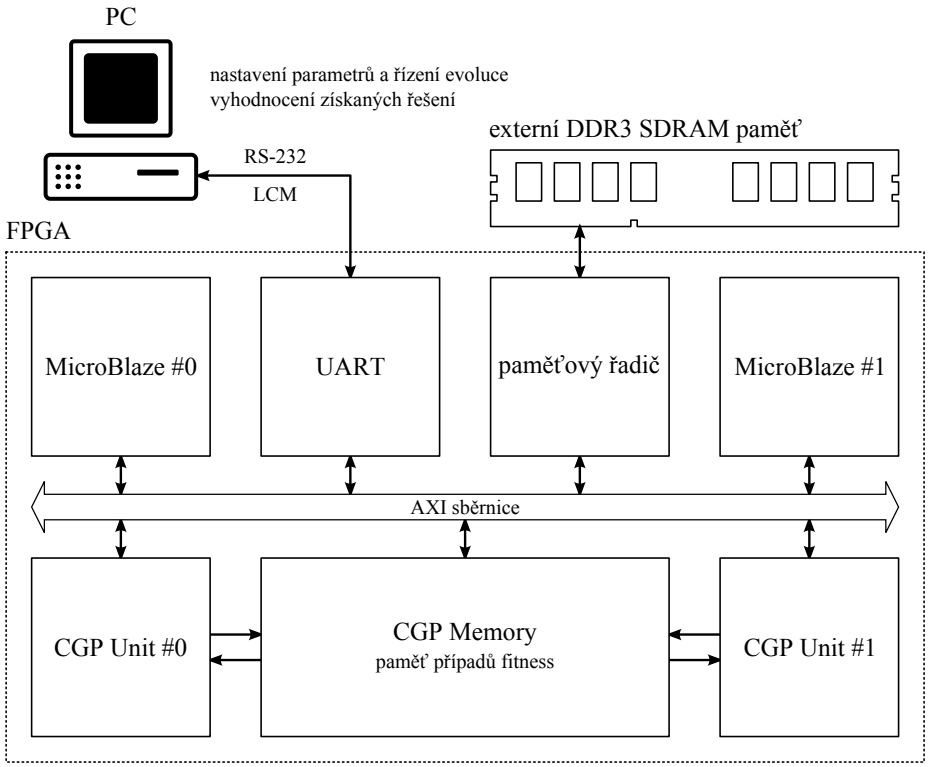
5.1 Volba cílové platformy

Při volbě cílové platformy byly uvažovány dvě základní varianty, které jsou dostupné na Ústavu počítačových systémů na naší fakultě. Obvody Xilinx Zynq díky zabudovanému dvoujádrovému procesoru ARM Cortex-A9 nabízí vysoký výpočetní výkon a pro urychlení kritických částí výpočtu je možné využít konfigurovatelného logického pole připojeného k procesorům sběrnicí AXI. Druhou variantou je řešení založené na standardním FPGA s využitím dvou soft procesorů MicroBlaze. V obou případech by byl proces vyhodnocení fitness přesunut do speciálně navržené periferie připojené na sběrnici AXI.

Vzhledem k nízké výpočetní náročnosti samotného procesu evoluce plně postačuje použití soft procesorů MicroBlaze. Oproti obvodům Zynq je navíc možné s výhodou využít větší plochy rekonfigurovatelného pole dostupného v FPGA řady Virtex-6 či Kintex-7 a dosáhnout tak většího zrychlení výpočtu fitness. Z těchto důvodů byla pro implementaci akcelerační jednotky zvolena platforma Virtex 6.

Akcelerační jednotka na FPGA je založena na dvou procesorech MicroBlaze [22] propojených AXI sběrnicí doplněných speciálními periferiemi navrženými na míru pro urychlení kartézského genetického programování. Trénovací obrazy (původní a zašuměný) jsou nejprve přeneseny z počítače do externí paměti DDR připojené k FPGA, následně jsou vytvořeny případy fitness a tyto jsou přeneseny do speciální paměti CGP Memory tvořené blokovou pamětí na FPGA. Každý procesor MicroBlaze má k dispozici vlastní jednotku CGP Unit, která čte data z paměti vektorů a podle nastavených chromozomů počítá fitness kandidátních filtrů.

Blokové schéma navrženého systému je vidět na obr. 5.1, počítač zde má pouze obslužnou



Obrázek 5.1: Schéma navrženého systému pro akceleraci koevoluce v FPGA.

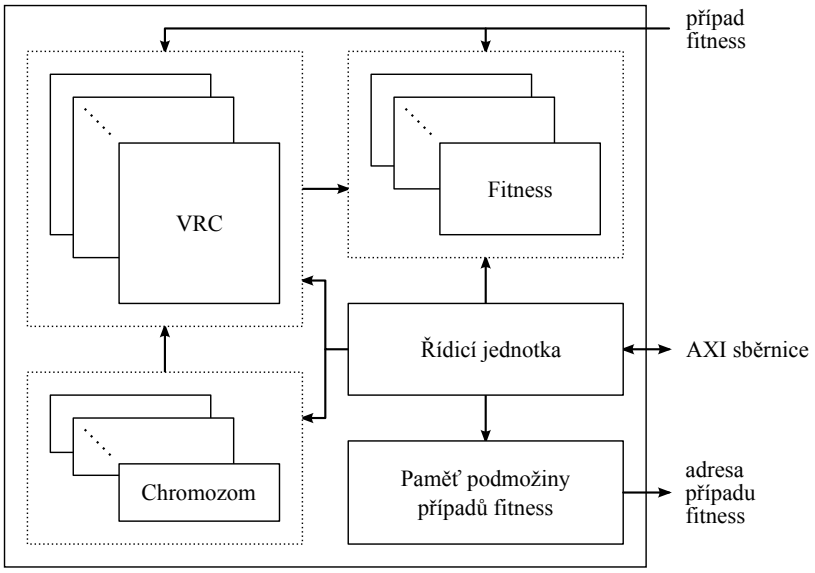
roli, celý proces koevoluce probíhá na FPGA připojeném k PC. Komunikace mezi počítačem a procesorem MicroBlaze #0 probíhá po sériové lince. K definici zpráv, zakódování a výměně dat je použita knihovna LCM (viz sekci 5.5.1). Výměna zpráv mezi procesory MicroBlaze je řešena komponentou Mailbox dodávanou firmou Xilinx jako standardní periferii procesoru MicroBlaze.

5.2 Akcelerace CGP

Jádrem akcelerační jednotky CGP Unit je sada virtuálních rekonfigurovatelných obvodů (VRC), které podle nastavených chromozomů implementují konkrétní kandidátní filtry. Jejich výstupy jsou zapojeny do Fitness jednotek. Případy fitness jsou dodávány na vstup kandidátních filtrů pamětí CGP Memory a adresa případu fitness je čtena z paměti podmožiny případů fitness (viz obr. 5.2).

Počet VRC je volitelný (parametr periferie), architektura VRC odpovídá obr. 4.6 v sekci 4.2. Počet sloupců a řádků je také volitelný a každý element uvnitř VRC obsahuje dva multiplexory pro výběr vstupního signálu a 16 funkcí těchto vstupů (prezentovaných v tabulce 4.1), jejichž výstup je vybrán výstupním multiplexorem.

Propojovací síť dovoluje vstupy elementů libovolně připojit na výstupy elementů v předchozích sloupcích a je plně zřetězená, s každým taktem hodinového signálu je tedy spočítána výstupní hodnota ve všech VRC. Tato hodnota je spolu s referenční hodnotou přivedena na vstup jednotky Fitness, která počítá ohodnocení kandidátního filtru (viz obr. 5.3). Po přečtení všech případů fitness a vyprázdnění zřetězené linky jsou na výstupu této jednotky dvě hodnoty fitness určené jako součet kvadratických (f_{sq}) resp. absolutních (f_{abs}) odchylek



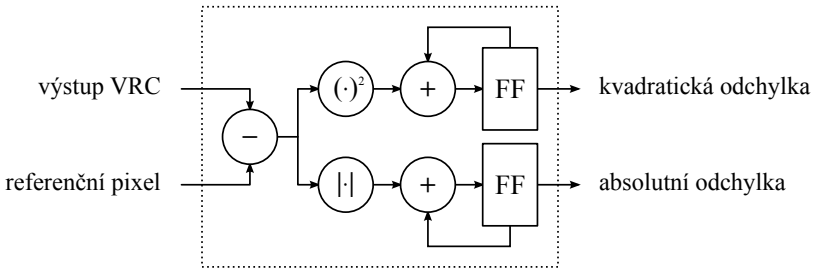
Obrázek 5.2: Blokové schéma komponenty CGP Unit.

pixelů filtrovaných kandidátním filtrem a pixelů původních (nepoškozených):

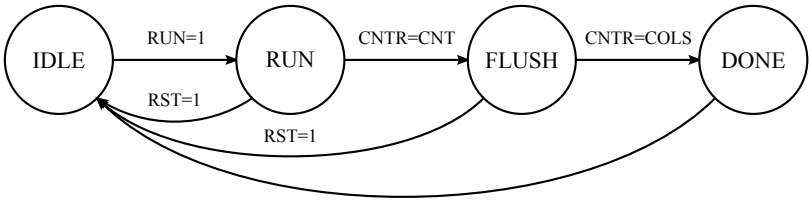
$$f_{\text{sq}} = \sum_{i=1}^{N} (u_i - v_i)^2 \,, \qquad f_{\text{abs}} = \sum_{i=1}^{N} |u_i - v_i| \,, \tag{5.1}$$

kde u_i je aktuální výstup VRC, v_i je zpožděná referenční hodnota a N je počet případů fitness. Normalizace počtem vektorů zde oproti MSE chybí, vzhledem ke konstantní velikosti podmnožiny případů fitness (FCS) však tato normalizace není podstatná a její vynechání šetří hardwarové prostředky (není potřeba dělička). Při provádění experimentů je pak možné zvolit, která hodnota fitness je použita pro ohodnocení populace.

Proces výpočtu fitness je řízen řídicí jednotkou, která obsahuje jednoduchý konečný stavový automat se čtyřmi stavy (viz obr. 5.4). Jednotka čeká ve stavu IDLE, nastavením nultého bitu v registru CTRL_REG je výpočet spuštěn, automat přechází do stavu RUN a je nastaven bit BUSY v registru STAT_REG. V tomto stavu je sekvenčně adresována paměť FCS, která pak nepřímo adresuje CGP Memory. Po dosažení stanoveného počtu případů fitness přechází automat do stavu FLUSH, ve kterém čeká na vyprázdnění zřetězené linky



Obrázek 5.3: Jednotka Fitness počítá kvadratickou a absolutní odchylku.



Obrázek 5.4: Konečný stavový automat uvnitř řídicí jednotky.

uvnitř VRC. Následuje zápis hodnot fitness do uživatelsky přístupných registrů a přechod do stavu IDLE, což je signalizováno vynulováním bitu BUSY v registru STAT_REG.

5.2.1 Paměťové mapování

Komponenta CGP Unit je připojena na společnou AXI sběrnici a k paměti případů fitness pomocí adresových a datových signálů. Paměťové mapování řídicích a stavových registrů, registrů s chromozomy a paměti FCS zajišťuje opět řídicí jednotka (viz tabulku 5.1).

Tabulka 5.1: Paměťové mapování na sběrnici AXI.

adresa	velikost (B)	přístup	název	popis
0x00000	MEM_SIZE · 2 B	R/W	FCS_MEM	paměť podmnožiny případů fitness
0x20000	4	R/W	CTRL_REG	řídicí registr
0x20004	4	R/W	STAT_REG	stavový registr
0x20008	4	R/W	CNT_REG	počet případů fitness
0x20024	4	R	INPUT_CNT	počet vstupů VRC
0x20028	4	R	COL_CNT	počet sloupců VRC
0x2002C	4	R	ROW_CNT	počet řádků VRC
0x20030	4	R	VRC_CNT	počet VRC
0x20034	4	R	MEM_SIZE	kapacita paměti FCS_MEM
0x20038	4	R	FIT_CNT	počet vstupů VRC
0x20100	VRC_CNT · 2 · 4	R	FITNESS	hodnoty fitness
0x30000	VRC_CNT · CHR_B	W	CHR_REG	chromozomy

Na úplném začátku paměťového prostoru je mapována paměť podmnožiny případů fitness (FCS_MEM), jejíž kapacita je určena generickým parametrem periferie CGP Unit. Každá položka v této paměti má velikost 2 B, celková kapacita je tedy MEM_SIZE · 2 B. Od adresy 0x20000 jsou mapovány registry – nejprve řídicí registr, stavový registr a registr CNT_REG určující počet případů fitness, dále pak konstantní registry s hodnotami parametrů jednotky CGP Unit (díky těmto hodnotám je SW běžící na procesorech MicroBlaze nezávislý na konfiguraci HW). Od adresy 0x20100 dále se nachází registry se spočtenými hodnotami fitness pro jednotlivé VRC. Chromozomy můžeme zapsat od adresy 0x30000 dále, registry s chromozomy jsou stínovány, tudíž je možné kopírovat nové chromozomy ještě během výpočtu fitness. Formát chromozomu je kompromisem mezi úsporou paměti a složitostí manipulace s chromozomem. Každý gen chromozomu je kódován samostatným bytem, celková délka chromozomu je tedy $(3 \cdot n_r \cdot n_c + 1)$ B. Pokud by jednotlivé geny nebyly zarovnány na celé byty, bylo by nutné při softwarové manipulaci využít mnoha bitových posunů a maskování, což by neúměrně zpomalovalo celý proces. Chromozomy jednotlivých VRC jsou v paměti zarovnány na celá slova (4 B).

5.2.2 Softwarové rozhraní komponenty

Každá periferie procesoru MicroBlaze má vlastní ovladač, který tvoří rozhraní mezi hardwarem a softwarem. Při vytváření vlastní periferie nabízí nástroj Xilinx Platform Studio možnost vygenerovat šablonu jak hardwarové implementace ve VHDL, tak i ovladače. Tato šablona obsahuje zpočátku pouze pomocná makra pro snadný přístup k paměťově mapovaným registrům a pamětem, rozhraní ovladače je pak vhodné rozšířit o složitější operace. Ovladač komponenty CGP Unit usnadňuje všechny dílčí operace prováděné při evoluci – nahrávání chromozomů, podmnožiny případů fitness, spouštění výpočtu fitness, čtení hodnot fitness apod. Pro manipulaci s konkrétní instancí komponenty CGP Unit je zaveden datový typ `cgp_unit_t` (viz výpis kódu A.1 v příloze A.1).

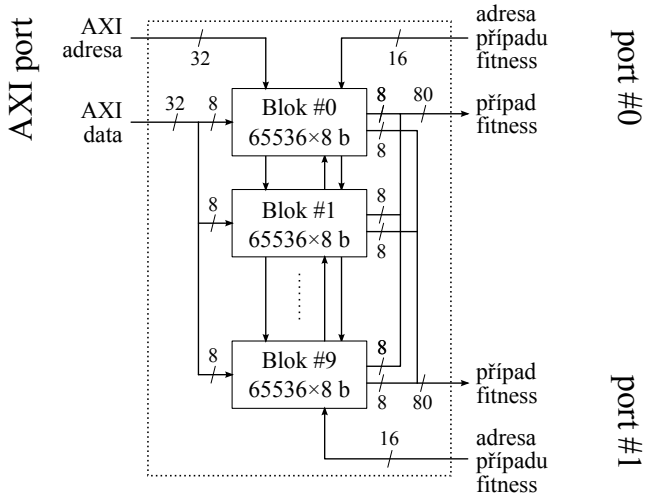
Deskriptor každé instance CGP Unit je ihned po startu programu inicializován metodou `CGP_UNIT_init()`, parametry (počet vstupů, řádků, sloupců apod.) jsou vyčteny z jednotky a na jejich základě jsou spočteny vlastnosti chromozomu (délka v bytech, slovech aj.). Ovladač dále poskytuje následující metody:

- `CPG_UNIT_printParams()` – vytiskne parametry dané jednotky na standardní výstup,
- `CGP_UNIT_reset()` – resetuje jednotku do výchozího stavu,
- `CGP_UNIT_run()` – spustí výpočet fitness,
- `CGP_UNIT_busy()` – zjistí, zda je jednotka zaneprázdněna,
- `CGP_UNIT_readFitness()` – přečte hodnotu fitness,
- `CGP_UNIT_copyChr()` – zkopíruje chromozom do jednotky,
- `CGP_UNIT_reloadChr()` – provede přepis chromozomu ze stínového registru,
- `CGP_UNIT_copyMem()` – zkopíruje obsah paměti podmnožiny případů fitness.

5.3 Paměť případů fitness

Paměť případů fitness je zavedena z důvodu dosažení potřebné propustnosti mezi pamětí a zřetězenými linkami. Všechny VRC totiž potřebují v každém hodinovém taktu na svém vstupu celý případ fitness (okolí pixelu) a adresování a načítání takového množství dat z obrazové paměti by bylo značně neefektivní. Vysoké kapacity blokových pamětí na dnešních FPGA dovolují načíst celou množinu případů fitness[1] a současně zásobovat dvě jednotky CGP Unit daty (blokové paměti jsou dvouportové). Paměť je složena z bloků s datovou šířkou 8 bitů, které jsou adresovány ze strany sběrnice AXI po (až) čtyřech bankách, ze strany akceleračních jednotek jsou všechny banky adresovány společně a data jsou spojena do jedné široké datové cesty (viz obr. 5.5). CGP Memory má celkem tři porty, ovšem blokové paměti na FPGA jsou pouze dvouportové. Řešením by bylo zdvojení těchto pamětí, pak by bylo možné realizovat skutečnou tříportovou paměť, ovšem za cenu zdvojnásobení použité kapacity blokových pamětí. Vzhledem k tomu, že zápis do této paměti je prováděn pouze před samotnou evolucí, může být připojení na sběrnici AXI multiplexováno s jedním z výstupních portů. Díky tomu není zbytečně plýtváno prostředky FPGA a přitom není narušena činnost obvodu.

[1] V případě obrázku o velikosti 265×256 a 9-okolí pixelu se jedná o $256 \cdot 256 \cdot 10 \cdot 8 = 5242880$ bitů (cca 5 243 Mb), kapacita blokových pamětí na FPGA Virtex-6 XC6VLX240T je 14 976 Mb [24]. Pro srovnání kapacita blokových pamětí na SoC Zynq-7020 XC7Z020 je pouze 4 480 Mb [29].



Obrázek 5.5: Blokové schéma komponenty CGP Memory.

Toto řešení umožňuje vysokou propustnost a zároveň minimalizuje množství dat přenášených při změně podmnožiny případů fitness – do jednotek CGP Unit jsou přenášeny pouze 16-bitové adresy případů fitness.

5.3.1 Softwarové rozhraní komponenty

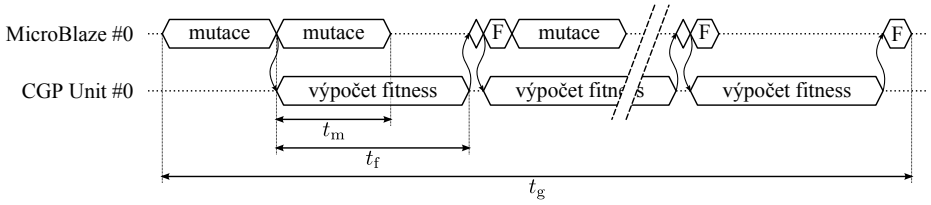
Ovladač komponenty CGP Memory poskytuje pouze metody umožňující zápis případů fitness do paměti uvnitř komponenty. Metoda `CGP_MEMORY_copyTrainVectors9()` vytvoří případy fitness z okolí 3×3 všech pixelů trénovacích obrazů a tyto případy nakopíruje do jednotky CGP Memory. V případě použití akcelerační jednotky s okolím o velikosti 5×5 je k dispozici obdobná metoda `CGP_MEMORY_copyTrainVectors25()`.

5.4 Průběh (ko)evoluce

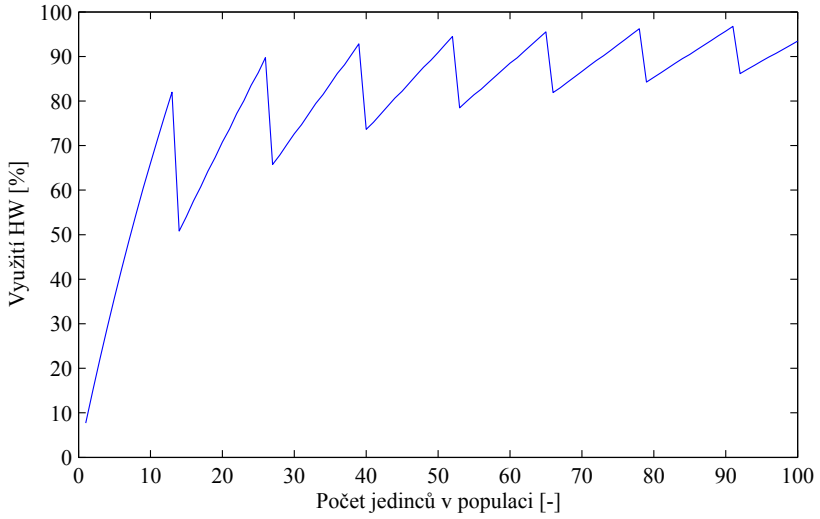
Časový diagram na obr. 5.6 ilustruje průběh jedné generace evolučního návrhu. Populace je rozdělena do N_{ch} skupin po P_{ch} jedincích v závislosti na počtu VRC N_{VRC} a velikosti populace P:

$$N_{\text{ch}} = \left\lceil \frac{P}{N_{\text{VRC}}} \right\rceil, \quad P_{\text{ch}} = \left\lceil \frac{P}{N_{\text{ch}}} \right\rceil. \tag{5.2}$$

Tímto způsobem je rovnoměrně rozložen výpočet fitness jedinců populace do N_{ch} kroků. Každá skupina jedinců musí být před započetím výpočtu fitness připravena, tzn. musí být provedeny mutace a chromozomy musí být přeneseny do CGP Unit. Až na první skupinu



Obrázek 5.6: Časový diagram průběhu evoluce.



Obrázek 5.7: Využití hardwaru v závislosti na velikosti populace P (zde pro počet VRC $N_{\mathrm{VRC}} = 13$, 65 536 případů fitness a až 5 mutací na chromozom).

je možné mutace překrýt s výpočtem fitness předchozí skupiny a lépe tak využít výpočetní výkon akcelerační jednotky. Musí však platit, že doba mutací t_{m} je kratší než doba výpočtu fitness t_{f}. Při zanedbání určité režie lze dobu jedné generace t_{g} vyjádřit jako:

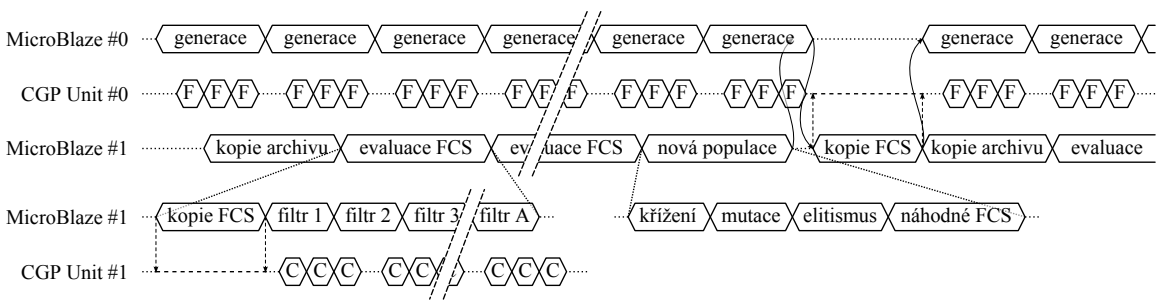
$$\begin{aligned} t_{\mathrm{m}} &= P_{\mathrm{ch}} \cdot T_{\mathrm{m}}, \\ t_{\mathrm{f}} &= (S + n_{\mathrm{c}} + 2) \cdot \frac{1}{f}, \\ t_{\mathrm{g}} &= t_{\mathrm{m}} + (N_{\mathrm{ch}} - 1) \cdot \max(t_{\mathrm{m}}, t_{\mathrm{f}}) + t_{\mathrm{f}}, \end{aligned} \tag{5.3}$$

kde T_{m} je doba provedení mutace jednoho jedince (závislá na počtu mutací na gen a velikosti chromozomu), S je velikost podmnožiny případů fitness a f pracovní frekvence systému. Využití hardwaru U_{HW} je pak:

$$U_{\mathrm{HW}} = \frac{t_{\mathrm{f}} \cdot P}{t_{\mathrm{g}} \cdot N_{\mathrm{VRC}}}. \tag{5.4}$$

Při použití klasické evoluce je možné využít obě dostupné jednotky CGP Unit a dosáhnout tak dalšího zrychlení. Počet jedinců v populaci je však výhodné volit jako celočíselný násobek počtu VRC, jak je patrné z obr. 5.7.

Zapojením koevoluce podmnožin případů fitness (FCS) se stává proces evolučního návrhu složitější, jak je možné vidět na časovém diagramu na obr. 5.8. Evoluce filtrů probíhá



Obrázek 5.8: Časový diagram průběhu koevoluce.

téměř jako při přístupu bez koevoluce, paralelně k ní běží evoluce FCS. Na konci každé generace je nejlepší podmnožina použita pro ohodnocení kandidátních filtrů. Vzhledem k navržené architektuře systému není nutné tyto podmnožiny sdílet mezi procesory MicroBlaze, procesor MicroBlaze #1 jednoduše zkopíruje nejlépe hodnocenou podmnožinu přímo do jednotky CGP Unit #0. Musí však počkat na dokončení generace v evoluci filtrů, aby byly všechny kandidátní filtry evaluovány na stejné podmnožině.

Evoluce FCS je časově náročnější než evoluce filtrů, neboť je nutné spočítat fitness všech filtrů z archivu na všech kandidátních FCS. Hodnota fitness i-tého jedince v populaci FCS je pak střední hodnota jednotlivých fitness na všech filtrech z archivu:

$$f_{\mathrm{FCS}}^{i} = \frac{1}{A} \sum_{j=1}^{A} f(i,j), \tag{5.5}$$

kde A je velikost archivu a $f(i,j)$ je hodnota fitness (buď kvadratická odchylka f_{sq} nebo absolutní odchylka f_{abs}, které jsou definovány vztahy (5.1) v kapitole 5.2) j-tého filtru z archivu na i-té FCS. Poté, co je ohodnocena celá populace FCS, je vytvořena nová populace. Zvolený počet jedinců vznikne křížením jedinců vybraných turnajovou selekcí, pro zajištění selekčního tlaku je použit elitismus. Několik jedinců je pak vytvořeno náhodně, tím je zajištěna genetická variabilita populace.

5.5 Software akcelerační jednotky

Soft procesory MicroBlaze nabízí několik softwarových variant, buď je možné software založit na plnohodnotném operačním systému (Xilinx dodává vlastní systém Xilkernel splňující standard POSIX, ale je možné na MicroBlaze provozovat i Linuxové jádro) nebo vytvořit tzv. *standalone* aplikaci, kdy má programátor k dispozici pouze základní ovladače. Tato jednodušší varianta byla zvolena i v případě navržené akcelerační jednotky.

Pro vývoj softwaru nabízí firma Xilinx vývojové prostředí Xilinx Software Development Kit založené na známém prostředí Eclipse. Každý systém využívající procesor MicroBlaze má odpovídající *Hardware Platform Specification*, což je samostatný projekt v Eclipse obsahující specifikaci daného hardwaru. Tato specifikace je vygenerována v Xilinx Platform Studiu a zahrnuje konfiguraci procesorů MicroBlaze a jejich periferií. Na základě této specifikace je pak pro každý přítomný procesor vytvořen tzv. *Board Support Package*, který umožňuje zvolit mezi operačním systémem a standalone řešením, zvolit dodatečné knihovny (podpora TCP/IP, FAT, Flash apod.) a nastavit parametry překladu. Přeložením tohoto projektu dojde k vytvoření knihoven dle parametrů procesoru MicroBlaze a dostupných periferií (včetně uživatelských).

Po přeložení výsledného projektu následuje fáze sestavení, při níž jsou kódy a data umístěny na konkrétní místa v paměti. Procesor MicroBlaze je vybaven více typy pamětí, přímo na FPGA využívá malé blokové paměti (8–64 kB), mimo FPGA pak na vývojové desce ML605 využívá externí DDR3 SDRAM paměť o kapacitě 512 MB. Paměťové mapování jednotlivých sekcí (program, konstantní data, zásobník, apod.) je dáno speciálním souborem *Linker Script*. Vzhledem k rozsáhlosti softwaru jsou všechny sekce mapovány do externí paměti. Každý ze dvou procesorů MicroBlaze je implementován samostatným projektem, sdílené části kódu jsou umístěny do zvláštních knihoven, jejichž implementačními detaily se zabývají následující sekce.

5.5.1 Komunikační knihovna LCM

Komunikace mezi počítačem a FPGA je usnadněna použitím knihovny LCM (Lightweight Communications and Marshalling), která byla původně navržena týmem DARPA Urban Challenge na Massachusetts Institute of Technology[2]. Tato knihovna umožňuje definovat vlastní typy zpráv, vygenerovat jejich implementaci v různých programovacích jazycích (C, C++, Java, Python, MATLAB a C#[3]) a řeší také výměnu těchto zpráv různými komunikačními rozhraními. Zprávy je možné posílat UDP i TCP pakety, přes sériové rozhraní a případně není problém doplnit knihovnu o vlastní komunikační prostředek.

Pro použití ve vestavěných systémech existuje odlehčená varianta LCM-lite, která postrádá některé pokročilé funkce plné verze knihovny (výběr kanálů regulárními výrazy apod.), jinak je však plně kompatibilní s plnou verzí. Tuto zjednodušenou verzi bylo nutné upravit tak, aby ji bylo možné použít na procesoru MicroBlaze, zejména bylo potřeba vhodně implementovat přijímání a odesílání dat přes periferii AXI UART 16550 [26] připojenou k procesoru MicroBlaze.

Zprávy jsou v LCM definovány ve formátu, který vychází z jazyka C (viz výpis A.4 v příloze A.3), pomocí nástroje lcm-gen jsou pak vygenerovány implementace těchto zpráv ve zvolených jazycích (viz výpisy A.6 a A.5). Každá taková implementace zahrnuje definici patřičného datového typu a pomocných metod, které zajišťují zpracování zprávy z proudu bytů resp. vytvoření proudu bytů z dané zprávy. Formát tohoto proudu bytů je shodný pro všechny implementace a zajišťuje také transparentnost vůči endianitě jednotlivých systémů.

Tímto způsobem je definováno následujících 9 typů zpráv:

- `cgp_params_msg_t` – parametry evoluce filtrů (počet generací, velikost populace atd.),
- `ga_params_msg_t` – parametry evoluce FCS,
- `control_msg_t` – řídicí zpráva (spuštění/zastavení evoluce),
- `status_msg_t` – stavová zpráva,
- `fitness_msg_t` – zpráva o aktuální hodnotě fitness,
- `image_msg_t` – zpráva pro přenos trénovacích obrazů,
- `chromosome_msg_t` – zpráva s chromozomem nejlepšího filtru,
- `log_msg_t` – výpis textu do konzole obslužné aplikace,
- `seed_msg_t` – inicializace generátorů náhodných čísel.

Komunikaci s počítačem má na starosti procesor MicroBlaze #0, po inicializaci periferií a knihoven probíhá v nekonečné smyčce periodické čtení a zpracování LCM zpráv. Při obdržení řídicí zprávy `control_msg_t` s pokynem ke spuštění evoluce je smyčka přerušena podprogramem `evolutionRun()`, po návratu pokračuje program v nekonečné smyčce. Procesor MicroBlaze #0 také periodicky indikuje svůj stav prostřednictvím stavové zprávy `status_msg_t`, pomocí komponenty AXI Timer [25] je každých 500 ms vyvolána metoda `sendStatus()`, která odešle aktuální stav (IDLE v případě nečinnosti nebo RUN, pokud běží evoluce).

[2] Knihovna je volně dostupná na adrese `<https://code.google.com/p/lcm/>`.

[3] Podpora jazyka C# byla přidána firmou Bender Robotics s.r.o. pro účely řízení autonomního reklamního robotu Advee [9].

5.5.2 Komunikace mezi procesory MicroBlaze

Duální systém procesorů MicroBlaze nabízí speciální komponentu AXI Mailbox [23] určenou k meziprocesorové komunikaci. Tato komponenta obsahuje dvouportové paměti umožňující výměnu krátkých zpráv, softwarové rozhraní tvoří dodaný ovladač, vyšší úroveň pak tvoří vlastní knihovna *mbox* poskytující funkce pro inicializaci, odeslání a příjem zprávy (viz výpis kódu 5.1).

Výpis kódu 5.1: Knihovna mbox pro meziprocesorovou komunikaci.

```
static XMbox mbox;
static XMbox_Config * mboxCfg;
u8 mboxBuf[MBOX_MAX_MSG_SIZE+4] __attribute__ ((aligned(4)));

void mboxInit(u16 deviceId)
{
    mboxCfg = XMbox_LookupConfig(deviceId);
    XMbox_CfgInitialize(&mbox, mboxCfg, mboxCfg->BaseAddress);
}

void mboxSend(u16 msgType, u16 msgSize, u8 *msgData)
{
    memcpy(mboxBuf, &msgType, 2);
    memcpy(mboxBuf+2, &msgSize, 2);
    memcpy(mboxBuf+4, msgData, msgSize);
    u32 size = (msgSize + 4 + 3) & 0xFFFFFFFC;
    XMbox_WriteBlocking(&mbox, (u32 *)&mboxBuf, size);
}

void mboxRecv(u16 *msgType, u16 *msgSize, u8 *msgData)
{
    u32 received;
    if (XMbox_Read(&mbox, (u32 *)mboxBuf, 4, &received) == XST_SUCCESS && ↩
        received == 4)
    {
        *msgType = (u16)mboxBuf[0];
        *msgSize = (u16)mboxBuf[2];
        u32 size = (*msgSize + 3) & 0xFFFFFFFC;
        XMbox_ReadBlocking(&mbox, (u32 *)&mboxBuf[4], size);
        memcpy(msgData, &mboxBuf[4], *msgSize);
    }
    else
    {
        *msgType = 0;
    }
}
```

Mezi procesory MicroBlaze dochází k výměně následujících typů zpráv:

- `MBOX_MSG_LOG` – výpis textu do konzole,
- `MBOX_MSG_CHR` – přenos chromozomu kandidátního filtru do archivu filtrů,
- `MBOX_MSG_CTRL` – řídicí zpráva (spuštění/zastavení evoluce FCS, synchronizace),
- `MBOX_MSG_STAT` – stavová zpráva (synchronizace),
- `MBOX_MSG_PARAMS` – parametry evoluce FCS,
- `MBOX_MSG_FITNESS` – aktuální hodnota fitness nejlepší FCS,
- `MBOX_MSG_SEED` – inicializace generátoru pseudonáhodných čísel.

5.5.3 Přesměrování standardního výstupu

Základní knihovny dostupné na standalone systému poskytují mimo jiné funkce pro práci se standardním vstupem a výstupem, které jsou obvykle přesměrovány na sériové rozhraní procesoru. Vzhledem k tomu, že sériové rozhraní je již obsazeno komunikační knihovnou LCM, bylo nutné přesměrovat standardní výstup pomocí LCM zprávy. K tomuto účelu byla navržena zpráva `log_msg_t`, která je zaslána do počítače vždy, když je zapsán nějaký řetězec na standardní výstup (přesněji při obdržení znaku nového řádku nebo při zavolání metody `flush()`). V případě procesoru MicroBlaze #1 však musí být tento řetězec nejprve odeslán pomocí rozhraní MailBox na procesor MicroBlaze #0 a teprve poté odeslán LCM zprávou do počítače. Zpráva `log_msg_t` pak nese informaci, z kterého procesoru řetězec pochází. Detaily implementace ilustruje výpis kódu 5.2, je zde vidět metoda `outbyte`, jejíž prototyp je definován v knihovnách dodaných firmou Xilinx a jejíž implementace určuje způsob zpracování standardního výstupu. Metodou `stdIoAdapterInit` je nastaveno zpětné volání, které zajistí zabalení textu do zprávy `log_msg_t` a odeslání po sériovém rozhraní.

Výpis kódu 5.2: Přesměrování standardního výstupu.

```
static char stdoutBuf[BUF_SIZE];
static int stdoutBufWrite = 0;
typedef void (*stdout_handler_t)(char *);
static stdout_handler_t stdoutHandler;

void stdIoAdapterInit(stdout_handler_t stdoutHdl)
{
    stdoutHandler = stdoutHdl;
}

void outbyte(char ch)
{
    stdoutBuf[stdoutBufWrite] = ch;
    stdoutBuf[stdoutBufWrite+1] = '\0';
    stdoutBufWrite++;
    if (ch == '\n' || stdoutBufWrite == BUF_SIZE-1)
        flush();
}

void flush()
{
    stdoutHandler(stdoutBuf);
    stdoutBufWrite = 0;
}
```

5.5.4 Knihovna pro evoluční algoritmy

Funkce pro manipulaci s chromozomy kandidátních filtrů (CGP) a podmnožin případů fitness (GA) jsou vyčleněny do samostatné knihovny. Parametry evoluce jsou předávány ve strukturách typu `cgp_params_t` resp. `ga_params_t` (viz výpisy kódu A.2 a A.3 v příloze A.2). Pro manipulaci s chromozomy kandidátních filtrů jsou pak k dispozici tyto metody:

- `cgpParamsInit()` – inicializuje knihovnu, předpočítá některé neměnné hodnoty,
- `cgpChromInit()` – náhodně inicializuje chromozom,
- `cgpChromMutate()` – dle předaných parametrů provede mutace na chromozomu.

Nad podmnožinami případů fitness je pak možné provádět následující operace:

- `gaTovInit()` – náhodná inicializace chromozomu,
- `gaTovMutate()` – mutace chromozomu,
- `gaTovCrossover()` – křížení dvou chromozomů vybraných pomocí turnajové selekce (`gaTournament()`).

5.5.5 Optimalizace

Pro dosažení maximálního využití hardwaru při koevolučním návrhu je nutné zajistit, aby doba mutací t_m byla kratší než doba výpočtu fitness t_f, která se úměrně snižuje s klesající velikostí FCS. Z tohoto důvodu byly zvoleny parametry obou procesorů MicroBlaze tak, aby bylo dosaženo co nejvyššího výkonu. Procesory MicroBlaze byly optimalizovány na rychlost (5-ti stupňová zřetězená instrukční linka), byl povolen válcový posouvač (barrel shifter), hardwarová násobička i dělička, predikce skoků, instrukční i datové cache o kapacitě 64 kB.

Analýzou evolučního procesu bylo zjištěno, že výrazného urychlení celého procesu je možné dosáhnout optimalizací generátoru pseudonáhodných čísel. Generátor ze standardní knihovny jazyka C byl nahrazen vlastním generátorem založeném na algoritmu *Xorshift* [15], který používá pouze operace bitových posunů a bitového XOR (viz výpis kódu 5.3).

Výpis kódu 5.3: Generátor pseudonáhodných čísel na bázi algoritmu Xorshift.

```
u32 x = 123456789;
u32 y = 362436069;
u32 z = 521288629;
u32 w = 88675123;

inline u32 randGet()
{
    u32 t = x ^ (x << 11);
    x = y; y = z; z = w;
    w = (w ^ (w >> 19)) ^ (t ^ (t >> 8));
    return w;
}
```

Turnajová selekce vyžaduje náhodný výběr několika jedinců z populace. Pro zajištění unikátnosti vybraných jedinců je použita Durstenfeldova verze algoritmu *Fisher–Yates shuffle* [5], která umožňuje generování pseudonáhodných posloupností (viz výpis kódu 5.4).

Výpis kódu 5.4: Generování pseudonáhodných posloupností.

```
void randSeq(u16 n, u16 k, u16 *seq)
{
    u16 i;
    static u16 scratch[RAND_MAX_N];
    for (i = 0; i < n; ++i)
        scratch[i] = i;
    for (i = 0; i < k; ++i)
    {
        u16 roll = randGet() % (n - i);
        seq[i] = scratch[roll];
        scratch[roll] = scratch[n-i];
        scratch[n-i] = seq[i];
    }
}
```

Tento algoritmus využívá pomocné pole `scratch` o velikosti n, které je nejprve inicializováno posloupností $0 \ldots n-1$. Pseudonáhodná posloupnost délky k je pak generována v k krocích, v i-tém kroku je náhodně vygenerován parametr `roll` v rozsahu $0 \ldots (n-i-1)$,

položka pole `scratch` na pozici `roll` je vyměněna s položkou na pozici $(n-i-1)$ a zapsána do výstupní posloupnosti. Tím je zajištěno, že v každém kroku je generováno číslo, které ještě ve výstupní posloupnosti není. Tabulka 5.2 ilustruje činnost algoritmu pro $n = k = 8$.

Tabulka 5.2: Ukázka činnosti algoritmu *Fisher–Yates shuffle*.

rozsah	`roll`	`scratch`	výsledná posloupnost
		0 1 2 3 4 5 6 7	
0...7	4	0 1 2 3 7 5 6 4	**4**
0...6	1	0 6 2 3 7 5 1 4	4 **1**
0...5	5	0 6 2 3 7 5 1 4	4 1 **5**
0...4	2	0 6 7 3 2 5 1 4	4 1 5 **2**
0...3	2	0 6 3 7 2 5 1 4	4 1 5 2 **7**
0...2	1	0 3 6 7 2 5 1 4	4 1 5 2 7 **6**
0...1	0	3 0 6 7 2 5 1 4	4 1 5 2 7 6 **0**
0	0	3 0 6 7 2 5 1 4	4 1 5 2 7 6 0 **3**

Mutace chromozomů jak kandidátních filtrů, tak i FCS vyžaduje generování pseudonáhodných čísel v rozsahu $0 \ldots (N-1)$. Tato operace obecně vyžaduje provedení operace modulo, která je výpočetně velmi náročná, neboť je implementována pomocí dělení (na procesoru MicroBlaze s hardwarovou děličkou a optimalizací na rychlost trvá dělení 32 taktů [27]). V některých případech se však můžeme omezit na generování čísel v rozsahu $0 \ldots (2^k - 1)$ a operaci modulo nahradit bitovou operací AND s maskou $m = 2^k - 1$. Omezením velikosti množiny případů fitness a FCS na mocninu dvojky tak dosáhneme výrazného urychlení. Další optimalizace evolučního procesu zahrnují rozbalení smyček (až 16×) především u mutací a křížení FCS, předpočítání některých neměnných hodnot před spuštěním evoluce apod.

5.5.6 Inicializace FPGA

Po zapnutí vývojového kitu je nutné nejprve inicializovat FPGA konfiguračním řetězcem a následně nahrát software obou procesorů MicroBlaze do paměti a spustit jejich činnost. Pro zjednodušení celého procesu byl navržen skript, který pomocí programu Xilinx Microprocessor Debugger (XMD) celý proces provede automaticky. Nástroj XMD je konzolová aplikace umožňující inicializaci FPGA a komunikaci nejen se soft procesory MicroBlaze. Kromě konzolového ovládání je možné tomuto nástroji předat cestu k TCL skriptu, jehož příkazy jsou následně vyvolány (`xmd.exe -tcl <cesta>`).

Výpis kódu 5.5: Inicializace FPGA pomocí Tcl skriptu a nástroje XMD.

```
fpga -f "C:/Data/VUT/FIT/MS4/DIP/bin/v6_100_3x3.bit"

connect mb mdm -cable type xilinx_platformusb port USB2 frequency 12000000 ↩
    -debugdevice deviceNr 2 cpunr 1
rst -processor
dow "C:/Data/VUT/FIT/MS4/DIP/bin/microblaze_0.elf"
run

connect mb mdm -cable type xilinx_platformusb port USB2 frequency 12000000 ↩
    -debugdevice deviceNr 2 cpunr 2
rst -processor
dow "C:/Data/VUT/FIT/MS4/DIP/bin/microblaze_1.elf"
run
```

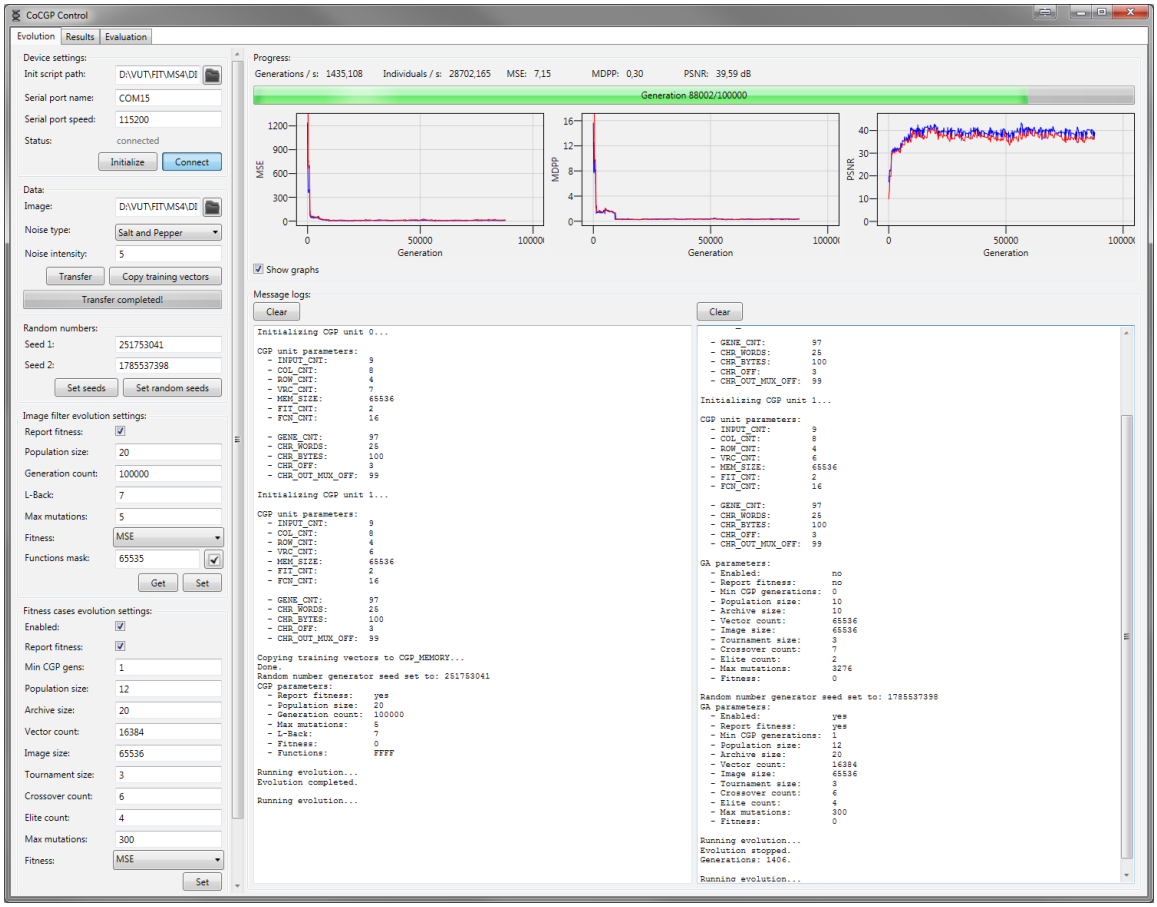
Navržený skript je možné vidět ve výpisu kódu 5.5, do FPGA je nejprve nahrán konfigurační řetězec, následně je XMD připojeno k jednotlivým procesorům a na každém z nich je spuštěn vlastní software.

5.5.7 Obslužná aplikace

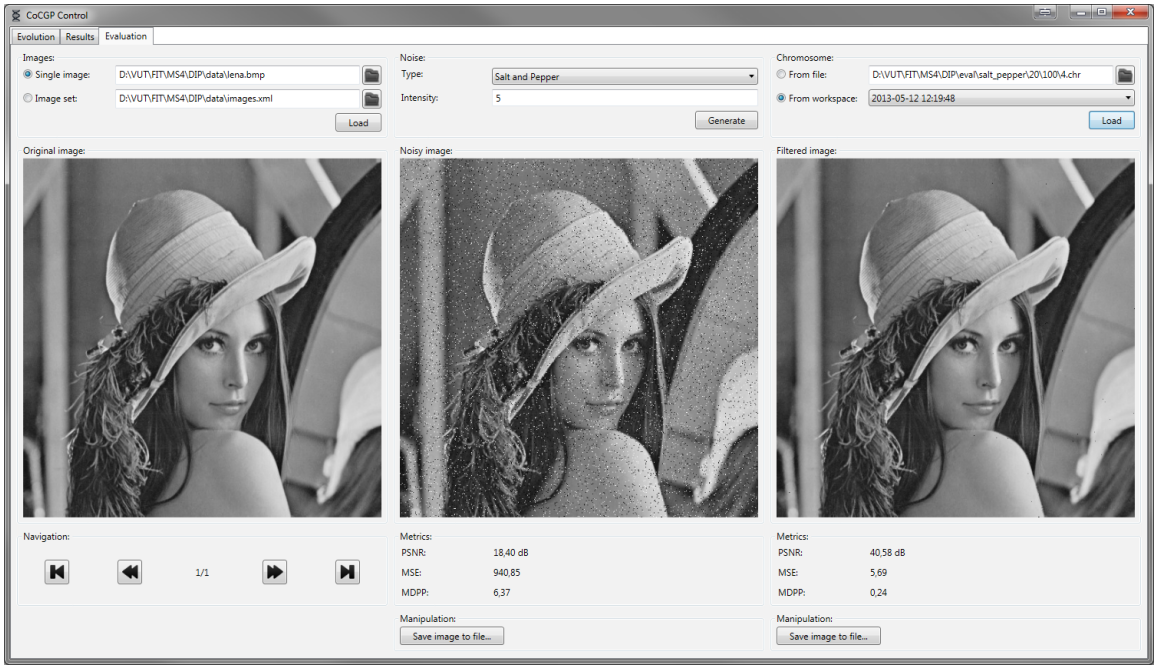
Poslední částí celého systému je obslužný program běžící na počítači, ke kterému je připojen vývojový kit s FPGA. Tato aplikace slouží k inicializaci úlohy, sledování stavu a vyhodnocení výsledků evoluce. Vývoj aplikace probíhal v prostředí Microsoft .NET Framework v jazyce C#, pro vytvoření uživatelského rozhraní byla použita technologie Windows Presentation Foundation (WPF) [13].

Aplikace obsahuje tři panely, první z nich slouží k nastavení parametrů evoluce, přenosu trénovacích obrazů a sledování průběhu evoluce (viz obr. 5.9). Po nastavení sériového portu a připojení k běžící akcelerační jednotce na FPGA je možné nastavit parametry evoluce filtrů a FCS a následně evoluční návrh spustit. Průběh evoluce je volitelně zobrazován pomocí grafů vývoje hodnoty fitness a ukazatelů rychlosti evoluce (počet generací za sekundu, počet jedinců za sekundu).

Po skončení evoluce je zachycen chromozom nejlepšího nalezeného kandidátního filtru, všechny získané chromozomy je možné vidět na druhém panelu. Kromě zobrazení grafů vývoje fitness je možné chromozom uložit do souboru nebo vyhodnotit kvalitu navrženého



Obrázek 5.9: Ovládání evoluce pomocí obslužné aplikace.



Obrázek 5.10: Vyhodnocení výsledků evoluce pomocí obslužné aplikace.

filtru pomocí posledního panelu.

Třetí panel, který je možné vidět na obr. 5.10, umožňuje vyhodnotit výsledky evoluce. Zobrazuje originální, zašuměný a vyfiltrovaný obraz, je možné načíst samostatný obrázek nebo i sadu obrázků definovanou XML souborem. Je možné zvolit typ šumu a jeho intenzitu a testovat tak úspěšnost filtrace i při jiném nastavení šumu než při evoluci. Pod zašuměným a vyfiltrovaným obrazem jsou zobrazeny hodnoty PSNR, MSE a MDPP.

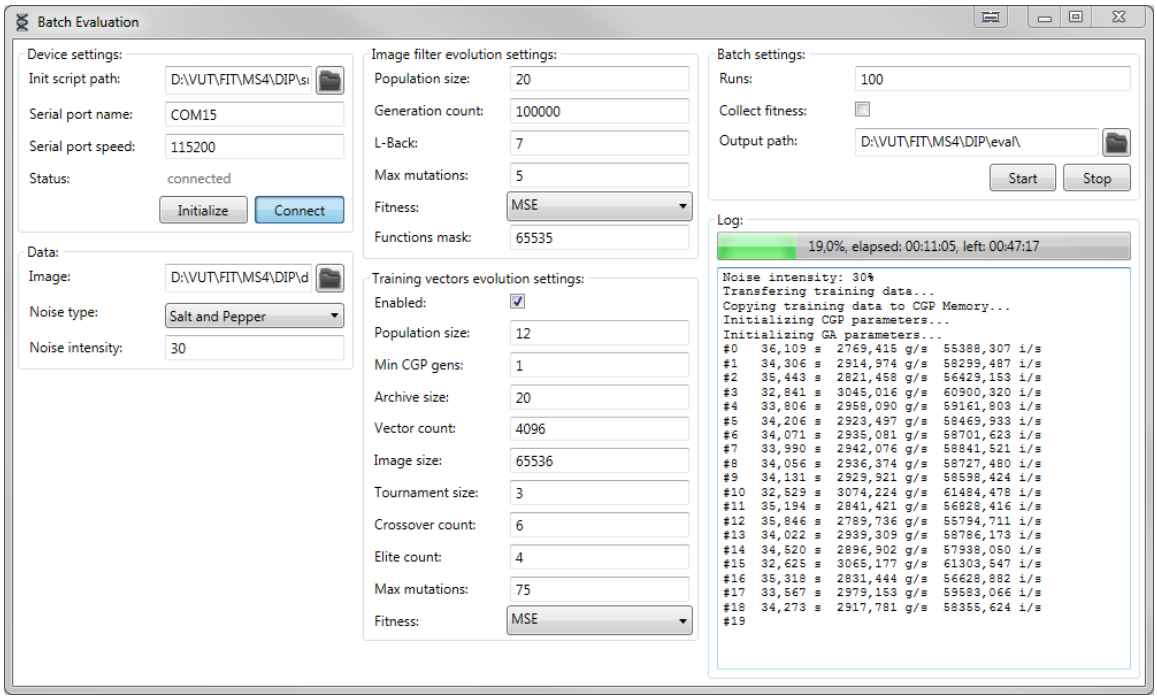
Kapitola 6

Experimentální výsledky

Tato kapitola se zabývá experimenty provedenými na navržené akcelerační jednotce. Je zde provedeno vyhodnocení akcelerace hardwarové platformy vůči velmi optimalizované softwarové verzi (ko)evolučního algoritmu. Na velkém množství experimentů je ukázán vliv koevoluce na kvalitu získaného obrazového filtru.

6.1 Dávkové spouštění úloh

Pro statistické vyhodnocení evolučního návrhu je nutné vždy spustit větší množství běhů pro stejné nastavení evoluce. Za tímto účelem byla navržena samostatná aplikace pro dávkové spouštění úloh (viz obr. 6.1), pomocí které je možné sbírat data ze zvoleného počtu běhů evolučního návrhu. Navíc je možné spustit dávky pro více úrovní šumu najednou. Program ukládá do zvoleného adresáře všechny získané chromozomy a volitelně i průběhy fitness



Obrázek 6.1: Aplikace pro dávkové spouštění úloh.

hodnot (do souboru MAT prostředí MATLAB). Po spuštění dávky informuje o aktuálním stavu, zobrazuje časy jednotlivých běhů a rychlost evoluce, z těchto údajů pak počítá odhad zbývající doby trvání dávky.

6.2 Softwarová implementace (ko)evoluce

Za účelem posouzení výkonnosti hardwarové platformy byla navržena také softwarová implementace (ko)evolučního návrhu obrazových filtrů. Sada konzolových aplikací napsaných v jazyce `C++` byla silně optimalizována použitím knihovny OpenMP pro běh ve více vláknech a použitím instrukcí SSE 4.1. Multimediální instrukce umožňují načíst 16 bytů (pixelů) do jednoho XMM registru (128 b) a provádět tak všechny operace nad všemi šestnácti pixely současně. Dalšího významného zrychlení je dosaženo analýzou chromozomů kandidátních filtrů před vyhodnocením fitness, neaktivní bloky CGP jsou zcela vynechány z výpočtu. V případě koevolučního návrhu je polovina počtu vláken vyhrazena pro evoluci kandidátních filtrů, druhá polovina pak pro evoluci podmnožin případů fitness (FCS).

Softwarová implementace zahrnuje mimo jiné nástroje pro návrh filtru pomocí evolučního a koevolučního přístupu, evaluační nástroje počítající statistiky fitness hodnot na zvolené sadě testovacích obrazů a další pomocné nástroje.

6.3 Konfigurace hardwarové platformy

Tabulka 6.1 uvádí parametry dvou hardwarových platforem pro okolí 3×3 a 5×5 pixelů s maximálními možnými počty VRC. V tabulce je také možné nalézt využití zdrojů FPGA – počty obsazených slice, LUT, registrů, blokových pamětí a DSP bloků.

Tabulka 6.1: Parametry hardwarové platformy a obsazenost FPGA (údaje o jednotlivých komponentách jsou přibližné, jedná se o odhady po syntéze).

Parametr	Okolí 3×3 px	Okolí 5×5 px
FPGA	XC6VLX240T	XC6VLX240T
Pracovní frekvence	100 MHz	100 MHz
Počty VRC	7 / 6	2 / 2
Rozměry VRC ($n_c \times n_r$)	8×4	8×6
Kapacita CGP Memory (příp. fitness)	65536	32768
Počet obsazených slice	32 065 / 37 680 (85,1 %)	24 399 / 37 680 (64,8 %)
Počet obsazených LUT	105 626 / 150 720 (70,1 %)	81 079 / 150 720 (53,8 %)
z toho MicroBlaze (#0/#1)	4 828 / 4 816	4 828 / 4 816
z toho CGP Unit (#0/#1)	49 093 / 42 182	31 962 / 31 962
z toho CGP Memory	248	254
Počet obsazených registrů	59 553 / 301 440 (19,8 %)	43 001 / 301 440 (14,2 %)
z toho MicroBlaze (#0/#1)	4 019 / 4 014	4 019 / 4 014
z toho CGP Unit (#0/#1)	19 996 / 17 184	16 551 / 16 551
z toho CGP Memory	131	131
Počet obsazených BRAM	300 / 416 (72,1 %)	316 / 416 (76,0 %)
z toho MicroBlaze (#0/#1)	36 / 36	36 / 36
z toho CGP Unit (#0/#1)	32 / 32	16 / 16
z toho CGP Memory	160	208
Počet obsazených DSP bloků	21 / 768 (2,7 %)	12 / 768 (1,6 %)

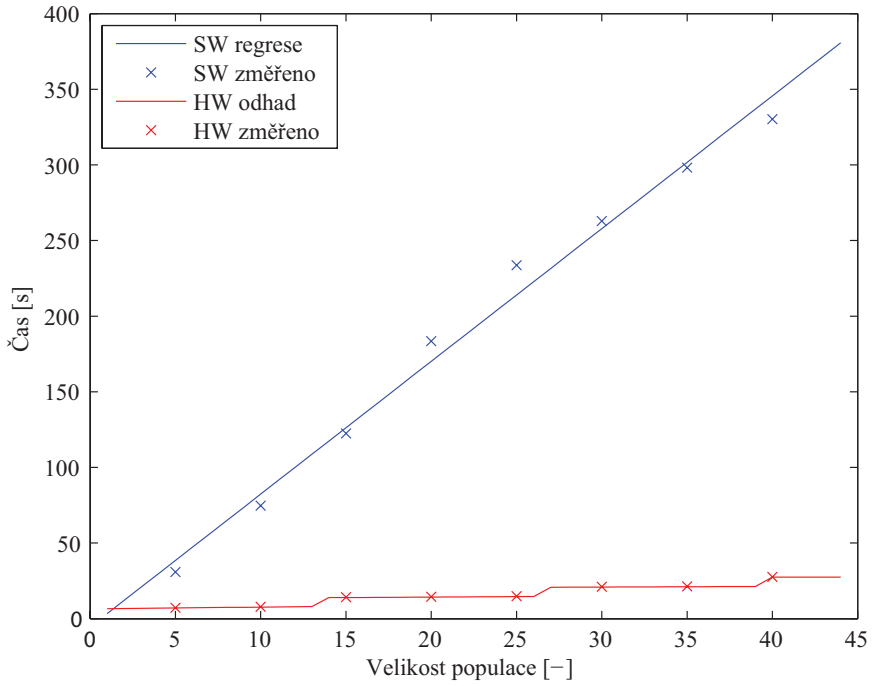
6.4 Vyhodnocení akcelerace (ko)evolučního návrhu

Výkonnost hardwarové platformy pro okolí 3×3 pixelů (viz tabulku 6.1) byla srovnána se softwarovou implementací, která byla spuštěna na procesoru Intel Core i7-860 (2,8 GHz) v osmi vláknech. Výsledky srovnání výkonnosti při evolučním návrhu jsou uvedeny v tabulce 6.2. Ačkoli byla softwarová verze značně optimalizována a běžela na výkonném procesoru, hardwarová platforma dosáhla mnohem lepších výsledků. Uvedené hodnoty jsou průměrné časy třiceti běhů evolučního algoritmu, výkonnost byla testována na úloze návrhu filtru šumu typu sůl a pepř, v každém běhu bylo vyhodnoceno 10 000 generací. Velikost populace byla volena od pěti do čtyřiceti jedinců, počet mutací na chromozom byl volen náhodně od jedné do pěti. Počet případů fitness byl 65 536 (trénovací obraz Lena 256×256 px).

Tabulka 6.2: Výkonnost hardwarové platformy při evolučním návrhu (30 běhů po 10 000 generacích).

velikost populace	5	10	15	20	25	30	35	40
SW čas (s)	30,83	74,65	122,39	183,40	233,71	262,82	298,24	330,25
HW čas (s)	7,21	7,86	14,17	14,44	14,83	21,06	21,31	27,64
akcelerace	4,28	9,50	8,63	12,70	15,77	12,48	13,99	11,95

Graf na obr. 6.2 zobrazuje naměřené hodnoty spolu s odhadem času hardwarové platformy založeném na vztahu (5.3) z kapitoly 5.4. Časy softwarové implementace jsou proloženy přímkou. Je patrné, že časy u softwarové implementace vykazují značný rozptyl, který je dán především způsobem vyhodnocení kandidátních filtrů. Oproti hardwarové platformě zde totiž dochází k analýze chromozomu každého jedince a vyloučení všech neaktivních bloků z výpočtu. Při určitých destruktivních mutacích (např. nastavení výstupu filtru na



Obrázek 6.2: Srovnání výkonnosti HW platformy a SW implementace při evolučním návrhu.

jeden z primárních vstupů) tak vyhodnocení jedince trvá podstatně kratší dobu.

Výkonnost koevolučního návrhu byla porovnána pro konstantní velikost populace $P = 20$ jedinců, 10 000 generací, trénovací obraz měl opět rozměry 256×256 px. Velikost podmnožiny případů fitness byla volena s ohledem na omezení zavedená optimalizací hardwarové platformy, tedy jako mocniny dvojky. Výsledky experimentu je možné vidět v tabulce 6.3. Velikost FCS je zde vyjádřena relativně vůči velikosti celé množiny případů fitness. Akcelerace hardwarové platformy je v případě koevoluce významnější, neboť softwarová implementace již není tak efektivní kvůli špatné datové lokalitě. Případy fitness (pixely trénovacího obrazu) již nejsou čteny sekvenčně, ale v pořadí daném aktuální FCS. Výkonnost je také degradována synchronizací mezi vlákny a předáváním velkých objemů dat.

Tabulka 6.3: Výkonnost hardwarové platformy při koevolučním návrhu (velikost populace $P = 20$, 30 běhů po 10 000 generacích).

velikost FCS	50 %	25 %	12,5 %	6,25 %	3,125 %	1,5625 %
SW čas (s)	713,23	405,47	223,18	133,91	88,26	71,92
HW čas (s)	12,51	6,91	4,23	3,57	4,20	3,97
akcelerace	56,99	58,64	52,82	37,49	21,04	18,11

Poslední sloupce tabulky 6.3 ukazují, že snížením velikosti FCS pod určitý práh (zde cca 12,5 % velikosti množiny případů fitness) dojde k saturování výkonnosti hardwarové platformy. Od tohoto prahu již totiž neplatí nerovnost $t_\mathrm{m} < t_\mathrm{f}$ (viz vztahy (5.3) v kapitole 5.4), tedy doba mutací již převyšuje dobu vyhodnocení fitness. Se snižováním velikosti FCS se navíc urychluje evoluce FCS a dochází častěji k interakci obou evolučních procesů. Proto pokud neomezíme frekvenci výměny FCS, může překvapivě doba koevoluce se snižující se velikostí FCS od určitého prahu růst.

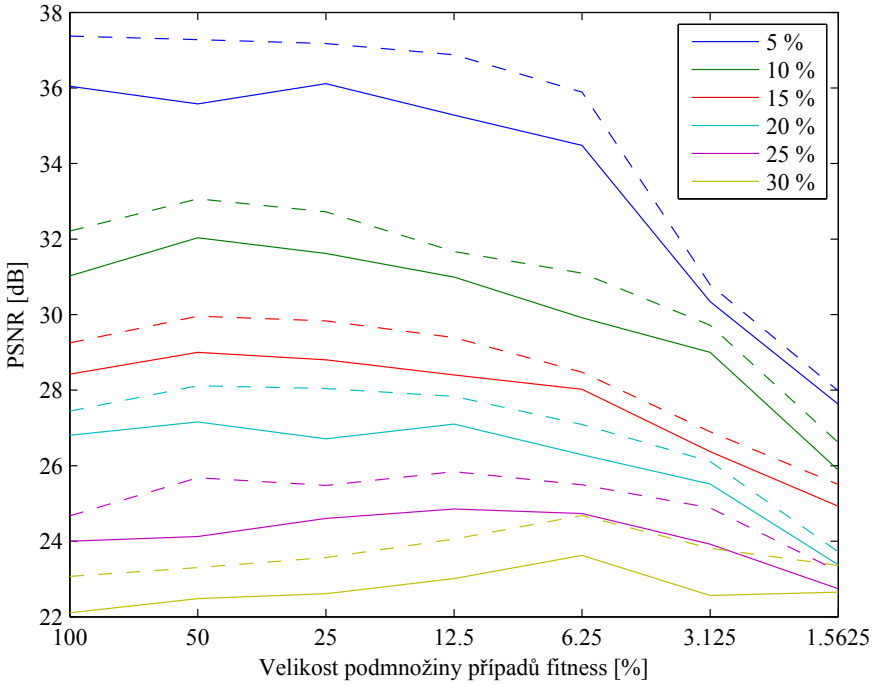
6.5 Kvalita nalezených řešení

Důležitým experimentem bylo srovnání kvality nalezených filtrů získaných evolučním přístupem a koevolučním přístupem při různých velikostech FCS. Kvalita filtrů byla posuzována na základě hodnoty PSNR (peak signal-to-noise ratio, špičkový poměr signálu k šumu), což je běžně používaná metoda ve zpracování obrazu. PSNR je logaritmická škála definovaná pomocí střední kvadratické chyby (MSE) a maximální možné hodnoty signálu (v našem případě 255):

$$\mathrm{PSNR}(x,y) = 10\log_{10}\frac{255^2}{\frac{1}{R\cdot C}\sum_{r,c}\left(x(r,c)-y(r,c)\right)^2} = 10\log_{10}\frac{255^2}{\mathrm{MSE}(x,y)}, \tag{6.1}$$

kde $x, y \in \{0,1,\ldots,255\}^{R\times C}$ jsou srovnávané obrazy o rozměrech $R \times C$ px.

Při experimentu byly použity následující parametry CGP: počet sloupců $n_\mathrm{c} = 8$, počet řádků $n_\mathrm{r} = 4$, parametr l-back $l = 7$, velikost populace $P = 20$. Evoluce FCS využívala jednoduchého GA s těmito parametry: velikost archivu $A = 20$, velikost populace $P_\mathrm{FCS} = 12$, šest jedinců vzniklo křížením, čtyři pomocí elitismu a zbylí dva byli vytvořeni náhodně v každé generaci. Jedinci určení ke křížení byli vybráni turnajovou selekcí tří náhodně vybraných jedinců, až 2 % chromozomu podléhalo mutaci. Tyto parametry byly určeny experimentálně s přihlédnutím k parametrům publikovaným v [21]. Jako trénovací obraz byl použit obraz Lena o velikosti 256×256 px. Pomocí (ko)evolučního přístupu byly navrhovány filtry šumu



Obrázek 6.3: Statistiky PSNR spočítané pro 100 evolvovaných filtrů (pro každou velikost FCS a úroveň šumu bylo spuštěno 100 běhů evoluce) na sadě 14 testovacích obrazů. Plnou čarou je vyznačena průměrná hodnota, čárkovanou pak medián.

typu sůl a pepř o intenzitách 5–30 %. Velikost FCS byla volena od 100 % (evoluční návrh) až po 1,5625 % velikosti množiny případů fitness. Pro každé nastavení bylo spuštěno 100 běhů po 100 000 generací, kvalita získaných řešení byla vyhodnocena na sadě 14 testovacích obrazů [8].

Vyhodnocení experimentu je možné vidět na obr. 6.3, podrobné statistiky pro jednotlivé úrovně šumu jsou uvedeny v příloze B.1. Výsledky experimentu ukazují, že použitím koevolučního návrhu je možné získat stejně kvalitní nebo i lepší obrazové filtry ve srovnání se standardním evolučním návrhem. V případě vyšších úrovní šumu se výhoda koevolučního návrhu projevuje výrazněji, kvalitních výsledků lze dosáhnout i s pouhými 6,25 % případů fitness.

6.6 Nejlepší nalezené filtry

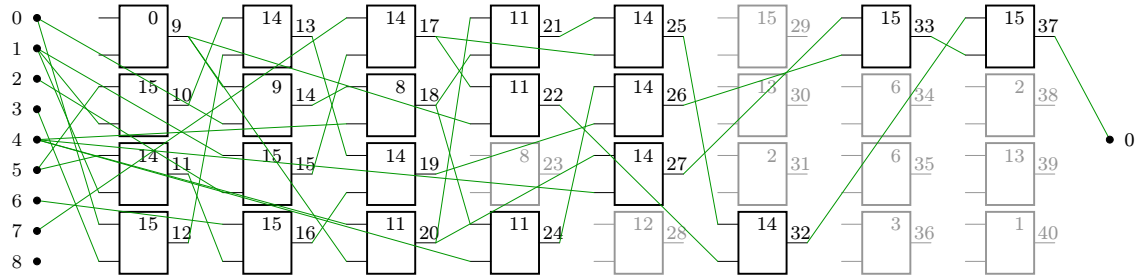
Obrázek 6.4(d) ukazuje zapojení nejlepšího nalezeného filtru pro šum typu sůl a pepř o intenzitě 5 %. Výsledek filtrace evolučně navrženým filtrem (obr. 6.4(c)) je srovnán s konvenčně používaným mediánovým filtrem (obr. 6.4(b)). Zatímco mediánový filtr odstraňuje drobné detaily v obraze, evolvovaný filtr tyto detaily zanechává. Rozdíl kvality je patrný také z hodnot PSNR, zašuměný obraz měl odstup od šumu 18 dB, po aplikaci mediánového filtru byl odstup od šumu 34,57 db a při použití evolučně navrženého filtru byla hodnota PSNR 44,49 dB. Při pohledu na zapojení filtru je patrné, že aktivní bloky využívají jen malou podmnožinu dostupných funkcí. Je zde jednou využita konstantní hodnota 255, dělení dvěma a čtyřmi, čtyřikrát součet, sedmkrát maximum a šestkrát minimum.

Při návrhu filtrů pro vyšší intenzity šumu byla použita hardwarová platforma umožňující

(a) šum 5 % (18,48 dB)

(b) medián 3×3 (34,57 dB)

(c) evolvovaný filtr (44,49 dB)

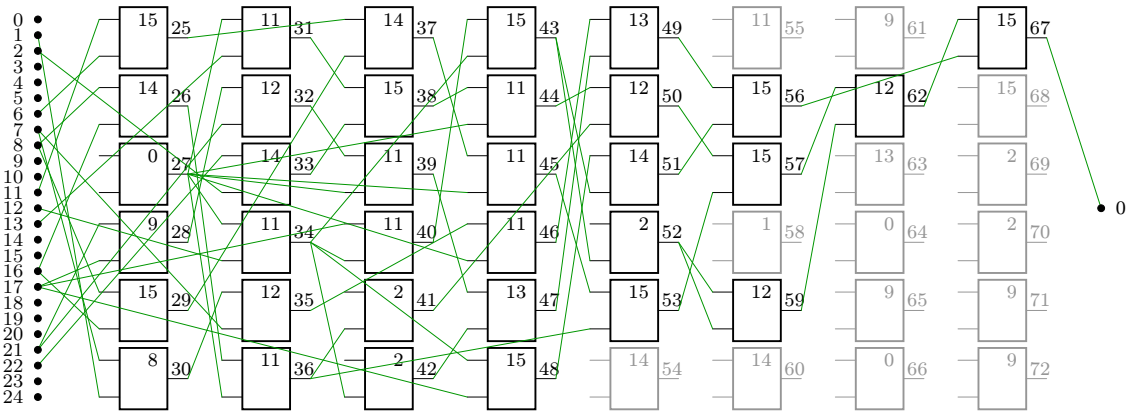


(d) zapojení filtru 3×3

(e) šum 50 % (8,40 dB)

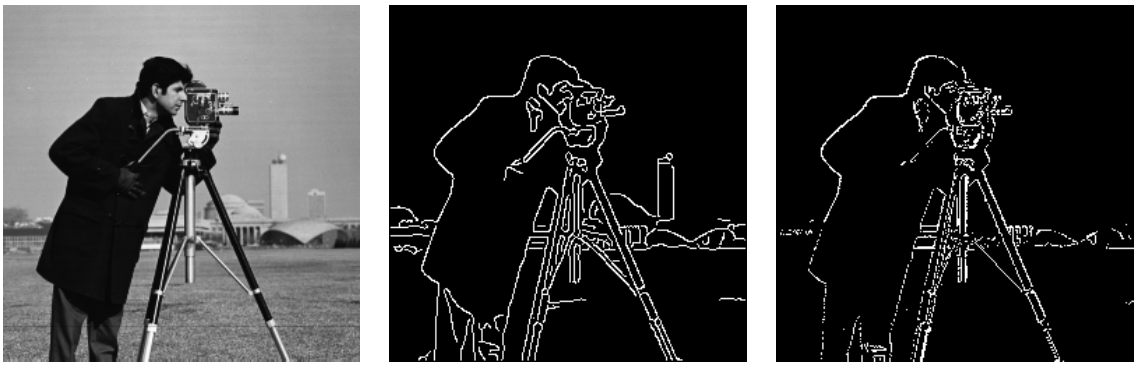
(f) medián 5×5 (22,60 dB)

(g) evolvovaný filtr (27,37 dB)

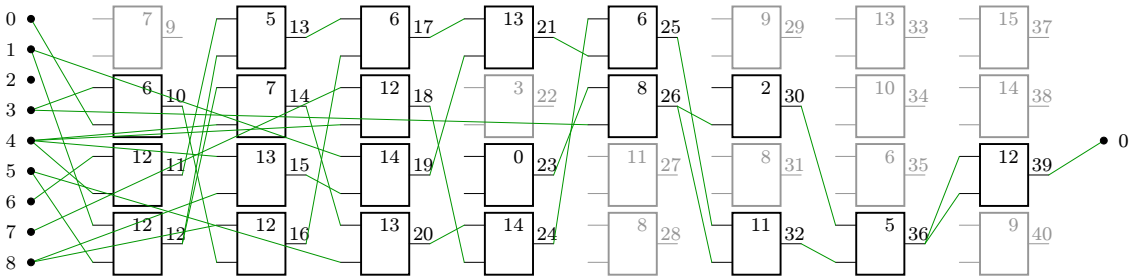


(h) zapojení filtru 5×5

Obrázek 6.4: Nejlepší nalezené filtry pro 5% resp. 50% šum typu sůl a pepř.



(a) původní obraz (b) Cannyho hranový detektor (c) evolvovaný detektor



(d) zapojení detektoru 3×3

Obrázek 6.5: Evolučně navržený detektor hran.

návrh filtrů pro okolí 5×5 px, neboť účinnost filtrace pomocí filtrů 3×3 již není dostatečně vysoká (viz statistiky na obr. 6.3). Nejlepší nalezený filtr šumu typu sůl a pepř o intenzitě 50 % byl opět porovnán s mediánovým filtrem, tentokrát však o velikosti 5×5 px. Jak je vidět na obr. 6.4(f), mediánový filtr této velikosti již velmi výrazně degraduje detaily v obraze a s vetšími shluky vadných pixelů si vůbec neporadí. Podstatně lepší kvality výsledného obrazu je dosaženo použitím evolvovaného filtru (obr. 6.4(g)), který velmi úspěšně rekonstruuje drobné detaily v obraze. Větší shluky vadných pixelů sice stejně jako mediánový filtr není schopen korektně filtrovat, nicméně je evidentní, že jsou tyto shluky podstatně méně rušivé v případě evolvovaného filtru. Při evolučním návrhu filtrů impulzního šumu se empiricky ukázalo jako vhodnější využití fitness funkce MSE, při použití MDPP docházelo častěji k uváznutí v lokálních optimech.

Evoluční návrh pomocí hardwarové platformy byl otestován i na jiných úlohách, než je filtrace impulzního šumu. Obrázek 6.5 ilustruje, že je možné pomocí evolučního návrhu vytvořit také detektor hran. Kvalita evolvovaného detektoru hran není tak vysoká, jako kvalita Cannyho hranového detektoru, nicméně je třeba vzít v potaz podstatně nižší výpočetní náročnost navrženého detektoru. Při evoluci byla v tomto případě na rozdíl od návrhu filtrů impulzního šumu využita fitness funkce MDPP. Lepších výsledků by mohlo být dosaženo změnou množiny funkcí jednotlivých bloků.

Kapitola 7

Závěr

Cílem práce bylo seznámit se s technologií rekonfigurovatelných logických obvodů, evolučními a koevolučními algoritmy a navrhnout hardwarovou jednotku pro akceleraci koevolučního návrhu obrazových filtrů. Použití evolučních algoritmů jako prostředku pro automatizovaný návrh systémů přináší při řešení některých problémů řadu výhod oproti tradičnímu inženýrskému přístupu. Koevoluční algoritmy využívají biologické předlohy interakce jedinců různých populací k dalšímu zefektivnění evolučního procesu a dosažení kvalitnějších výsledků evolučního návrhu.

Navržená akcelerační jednotka umožňuje urychlení procesu koevoluce obrazových filtrů založeném na kartézském genetickém programování. Jednotka je implementována na rekonfigurovatelném obvodu FPGA, využívá soft procesorů MicroBlaze, které jsou doplněny vlastními periferiemi, zejména virtuálními rekonfigurovatelnými obvody pro rychlé vyhodnocení fitness populace. Komunikace mezi počítačem, který má pouze obslužnou roli, a FPGA probíhá po sériové lince. Trénovací data jsou umístěna do blokové paměti na FPGA, čímž je umožněno čtení případů fitness s vysokou propustností. Je tak využit potenciál zřetězených linek ve virtuálních rekonfigurovatelných obvodech.

Pomocí navržené hardwarové platformy se podařilo významně (až 58×) urychlit koevoluční návrh obrazových filtrů oproti optimalizované softwarové implementaci. Bylo ukázáno, že při návrhu filtru impulzního šumu především vyšší intenzity je možné zavedením koevoluce případů fitness dosáhnout kvalitnějších řešení oproti standardnímu evolučnímu návrhu a zároveň snížit dobu potřebnou k nalezení takového řešení. Využitelnost hardwarové platformy pro návrh jiných typů obrazových filtrů byla prokázána při návrhu hranového detektoru.

Jednotku lze využít jednak pro jednorázový návrh filtrů, lze ji však také vestavět do existujícího systému zpracovávajícího obrazová data a vytvořit tak adaptivní systém reagující např. na změny okolního prostředí nebo poruchy hardwaru. S mírnými úpravami je možné navrženou akcelerační jednotku využít i pro návrh jiných typů obvodů, např. filtrů jiných typů šumu nebo kombinačních obvodů.

Prezentace této práce se umístila na prvním místě v kategorii magisterských projektů Počítačové systémy na konferenci a soutěži STUDENT EEICT 2013. Výsledky práce budou rovněž publikovány v příspěvku *Coevolutionary Cartesian Genetic Programming in FPGA*, který je v současné době v recenzním řízení na konferenci ECAL 2013, *12th European Conference on Artificial Life*.

Literatura

[1] Canny, J. A Computational Approach to Edge Detection. *IEEE Trans. Pattern Anal. Mach. Intell.* červen 1986, roč. 8, č. 6. S. 679–698. Dostupné na: <http://dx.doi.org/10.1109/TPAMI.1986.4767851>. ISSN 0162-8828.

[2] Ciletti, M. *Advanced Digital Design with the Verilog HDL.* [b.m.]: Prentice Hall, 2010. Prentice Hall Xilinx design series. ISBN 978-01-3601-928-2.

[3] Cramer, N. L. A Representation for the Adaptive Generation of Simple Sequential Programs. In Grefenstette, J. J. (ed.). *Proceedings of the International Conference on Genetic Algorithms and their Applications.* 1985.

[4] Dobai, R. a Sekanina, L. Towards Evolvable Systems Based on the Xilinx Zynq Platform. In *2013 IEEE International Conference on Evolvable Systems (ICES).* [b.m.]: IEEE Computational Intelligence Society, 2013. S. 89–95. Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI). ISBN 978-1-4673-5869-9.

[5] Durstenfeld, R. Algorithm 235: Random permutation. *Commun. ACM.* červenec 1964, roč. 7, č. 7. S. 420. ISSN 0001-0782.

[6] Flegr, J. *Zamrzlá evoluce, aneb, Je to jinak, pane Darwin.* 1. vyd. [b.m.]: Academia, 2011. 326 s. ISBN 978-80-200-1526-6.

[7] Gonzalez, R. C. a Woods, R. E. *Digital Image Processing.* 3rd. [b.m.]: Addison-Wesley Longman Publishing Co., Inc., 2001. ISBN 0201180758.

[8] Gonzalez, R. C., Woods, R. E. a Eddins, S. L. *ImageProcessingPlace.com: "Standard"test images.* ImageProcessingPlace.com. 2009. Dostupné na: <http://www.imageprocessingplace.com>.

[9] Hrbáček, J., Hrbáček, R. a Věchet, S. Modular Control System Architecture for a Mobile Robot. In *Proceedings of the 17th international conference Engineering Mechanics.* 2011. S. 211–214. ISBN 978-80-87012-33-8.

[10] Jaśkowski, W. *Algorithms for Test-Based Problems.* Poznan University of Technology – Faculty of Computing and Information Science, 2011. Disertační práce.

[11] Kolouch, J. *Programovatelné logické obvody.* [b.m.]: Vysoké učení technické v Brně, 2009.

[12] Koza, J. R. *36 Human-Competitive Results Produced by Genetic Programming.* prosinec 2003. Dostupné na: <http://www.genetic-programming.com/humancompetitive.html>.

[13] MacDonald, M. *Pro WPF 4.5 in C#: Windows Presentation Foundation in .NET 4.5.* 4. vyd. [b.m.]: Apress, 2012. 1112 s.

[14] Mařík, V., Štěpánková, O., Lažanský, J. et al. *Umělá inteligence (3).* [b.m.]: Academia, 2001.

[15] Marsaglia, G. Xorshift RNGs. *Journal of Statistical Software.* 2003, roč. 8, č. 14. S. 1–6. ISSN 1548-7660.

[16] Popovici, E., Bucci, A., Wiegand, R. et al. *Handbook of Natural Computing.* [b.m.]: Springer Berlin Heidelberg, 2012. Coevolutionary Principles, s. 987–1033. ISBN 978-3-540-92909-3.

[17] Sekanina, L., Vašíček, Z., Růžička, R. et al. *Evoluční hardware: Od automatického generování patentovatelných invencí k sebemodifikujícím se strojům.* [b.m.]: Academia, 2009. 328 s. Edice Gerstner. ISBN 978-80-200-1729-1.

[18] Sekanina, L., Harding, S. L., Banzhaf, W. et al. *Cartesian Genetic Programming.* [b.m.]: Springer Berlin Heidelberg, 2011. Image Processing and CGP, s. 181–215. ISBN 978-3-642-17310-3.

[19] Šťastný, J. *FPGA prakticky: Realizace číslicových systémů pro programovatelná hradlová pole.* [b.m.]: BEN – technická literatura, 2011. ISBN 978-80-7300-261-9.

[20] Šikulová, M. a Sekanina, L. Acceleration of Evolutionary Image Filter Design Using Coevolution in Cartesian GP. In *Proceedings of the 12th International Conference on Parallel Problem Solving from Nature.* [b.m.]: Springer Verlag, 2012. S. 163–172. LNCS 7491. ISBN 978-3-642-32936-4.

[21] Šikulová, M. a Sekanina, L. Coevolution in Cartesian Genetic Programming. In *Proceedings of the 15th European Conference on Genetic Programming.* [b.m.]: Springer Verlag, 2012. S. 182–193. LNCS 7244. ISBN 978-3-642-29138-8.

[22] Xilinx. *Dual Processor Reference Design Suite.* 2008.

[23] Xilinx. *LogiCORE IP Mailbox (v1.00a) – Product Specification.* 2010.

[24] Xilinx. *7 Series FPGAs Overview – Advance Product Specification.* 2012.

[25] Xilinx. *LogiCORE IP AXI Timer (v1.03a) – Product Guide.* 2012.

[26] Xilinx. *LogiCORE IP AXI UART 16550 (v1.01a) – Product Specification.* 2012.

[27] Xilinx. *MicroBlaze Processor Reference Guide – Embedded Development Kit EDK 14.2.* 2012.

[28] Xilinx. *Vitex-6 Family Overview – Product Specification.* 2012.

[29] Xilinx. *Zynq-7000 All Programmable Soc Overview – Preliminary Product Specification.* 2013.

[30] Zelinka, I., Oplatková, Z., Šeda, M. et al. *Evoluční výpočetní techniky: Principy a aplikace.* [b.m.]: BEN – technická literatura, 2009.

Seznam použitých zkratek

ASIC	Application Specific Integrated Circuit
AXI	Advanced eXtensible Interface
CGP	Kartézské genetické programování
CLB	Configurable Logic Block
CPLD	Complex Programmable Logic Device
EA	Evoluční algoritmus
ES	Evoluční strategie
FCS	Fitness Cases Subset (podmnožina případů fitness)
FPGA	Field Programmable Gate Array
GA	Genetický algoritmus
GP	Genetické programování
HDL	Hardware Description Language
LCM	Lightweight Communications and Marshalling
LUT	Look-Up Table
MDPP	Mean Difference Per Pixel (střední odchylka pixelů)
MSE	Mean Squared Error (střední kvadratická chyba)
PSNR	Peak Signal to Noise Ratio (špičkový poměr signálu k šumu)
SoC	System on Chip
SSE	Streaming SIMD Extensions
VHDL	Very High Speed Integrated Circuit HDL
VRC	Virtuální rekonfigurovatelný obvod

Seznam příloh

Příloha A

Výpisy kódu

A.1 Softwarové rozhraní komponenty CGP Unit

Výpis kódu A.1: Deskriptor komponenty CGP Unit.

```
typedef struct
{
    uint32_t baseAddr;
    uint16_t inputCnt;
    uint16_t colCnt;
    uint16_t rowCnt;
    uint16_t vrcCnt;
    uint32_t memSize;
    uint16_t fitCnt;
    uint16_t fcnCnt;
    uint16_t geneCnt;
    uint16_t chrWords;
    uint16_t chrBytes;
    uint16_t chrOff;
    uint16_t outMuxOff;
} cgp_unit_t;
```

A.2 Knihovna pro evoluční algoritmy

Výpis kódu A.2: Datový typ určený pro předávání parametrů evoluce filtrů.

```
typedef struct
{
    bool reportFitness;
    u16  populationSize;
    u32  generationCnt;
    u16  maxMutations;
    u16  lBack;
    u8   fitSel;
    u32  fcnMask;
} cgp_params_t;
```

Výpis kódu A.3: Datový typ určený pro předávání parametrů evoluce FCS.

```
typedef struct
{
    bool enabled;
    bool reportFitness;
    u16  minCGPGenerations;
    u16  populationSize;
    u16  archiveSize;
    u32  vectorCnt;
    u32  imageSize;
    u16  tournamentSize;
    u16  crossoverCnt;
    u16  eliteCnt;
    u16  maxMutations;
    u8   fitSel;
} ga_params_t;
```

A.3 Definice zpráv pro knihovnu LCM

Výpis kódu A.4: Ukázka definice zprávy pro knihovnu LCM.

```
struct cgp_params_msg_t
{
    boolean   reportFitness;
    int16_t   populationSize;
    int32_t   generationCnt;
    int16_t   lBack;
    int16_t   maxMutations;
    int8_t    fitSel;
    int32_t   fcnMask;
}
```

Výpis kódu A.5: Vygenerovaná zpráva v jazyce C#.

```
using System;
using System.Collections.Generic;
using System.IO;
using LCM.LCM;

namespace Messaging
{
    public sealed class cgp_params_msg_t : LCM.LCM.LCMEncodable
    {
        public bool reportFitness;
        public short populationSize;
        public int generationCnt;
        public short lBack;
        public short maxMutations;
        public byte fitSel;
        public int fcnMask;

        ...
    }
}
```

Výpis kódu A.6: Vygenerovaná zpráva v jazyce C.

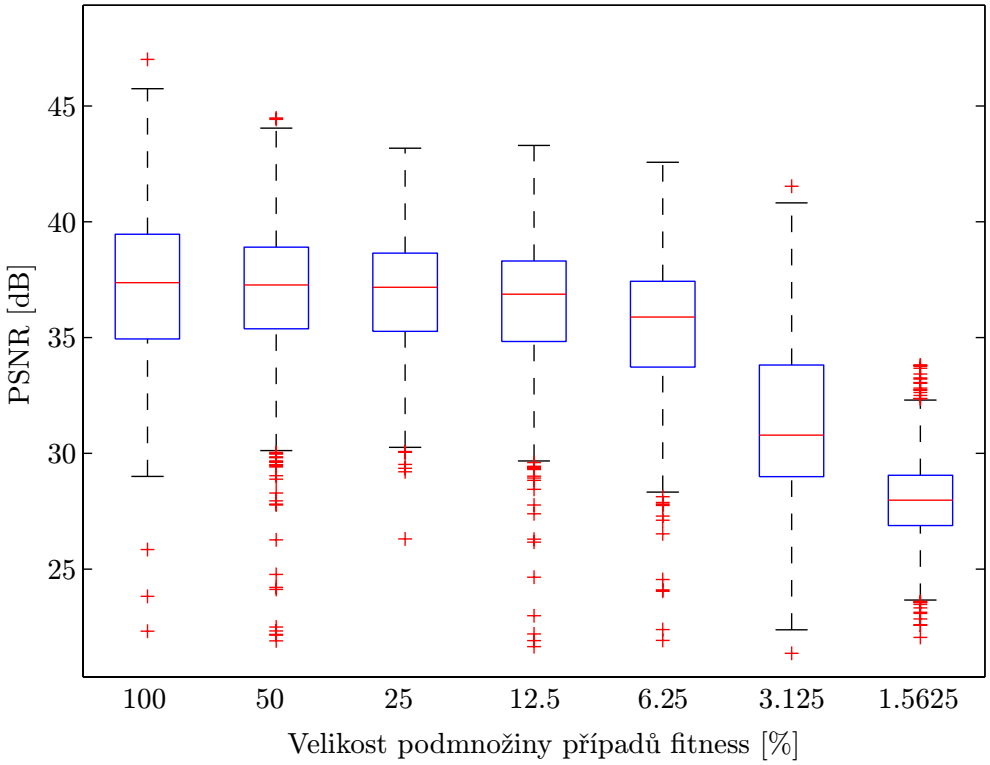
```
#include <stdint.h>
#include <stdlib.h>
#include <lcm/lcm_coretypes.h>

#ifndef _cgp_params_msg_t_h
#define _cgp_params_msg_t_h

#ifdef __cplusplus
extern "C" {
#endif

typedef struct _cgp_params_msg_t cgp_params_msg_t;
struct _cgp_params_msg_t
{
    int8_t      reportFitness;
    int16_t     populationSize;
    int32_t     generationCnt;
    int16_t     lBack;
    int16_t     maxMutations;
    int8_t      fitSel;
    int32_t     fcnMask;
};

...

#ifdef __cplusplus
}
#endif

#endif
```

Příloha B

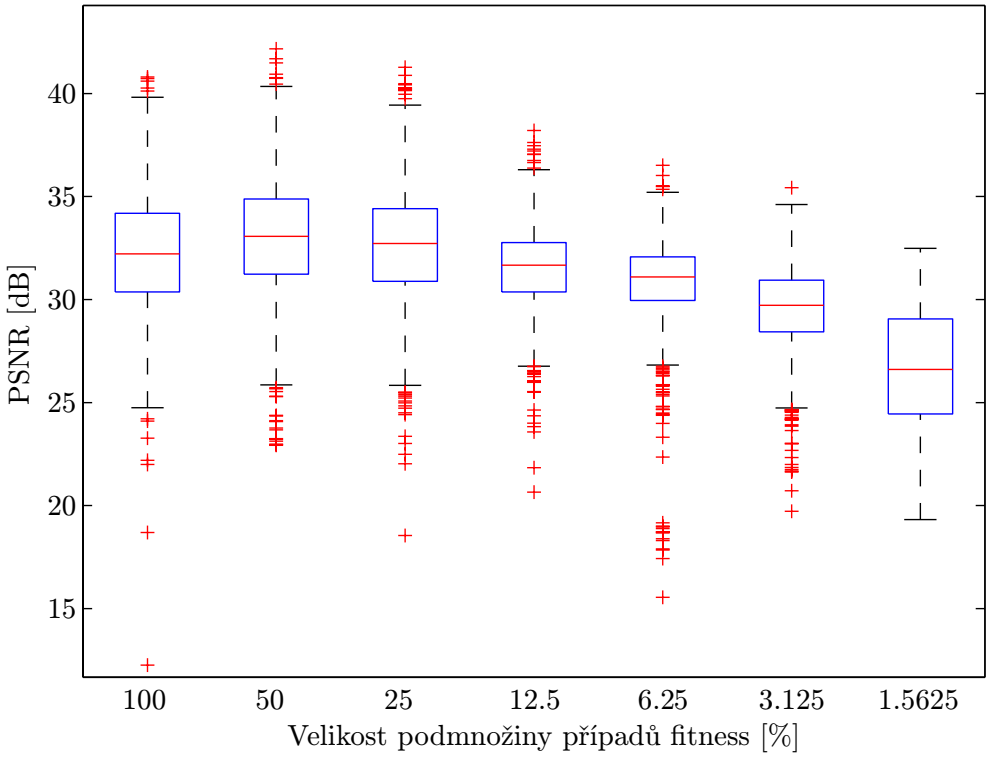
Grafy

B.1 Statistiky PSNR

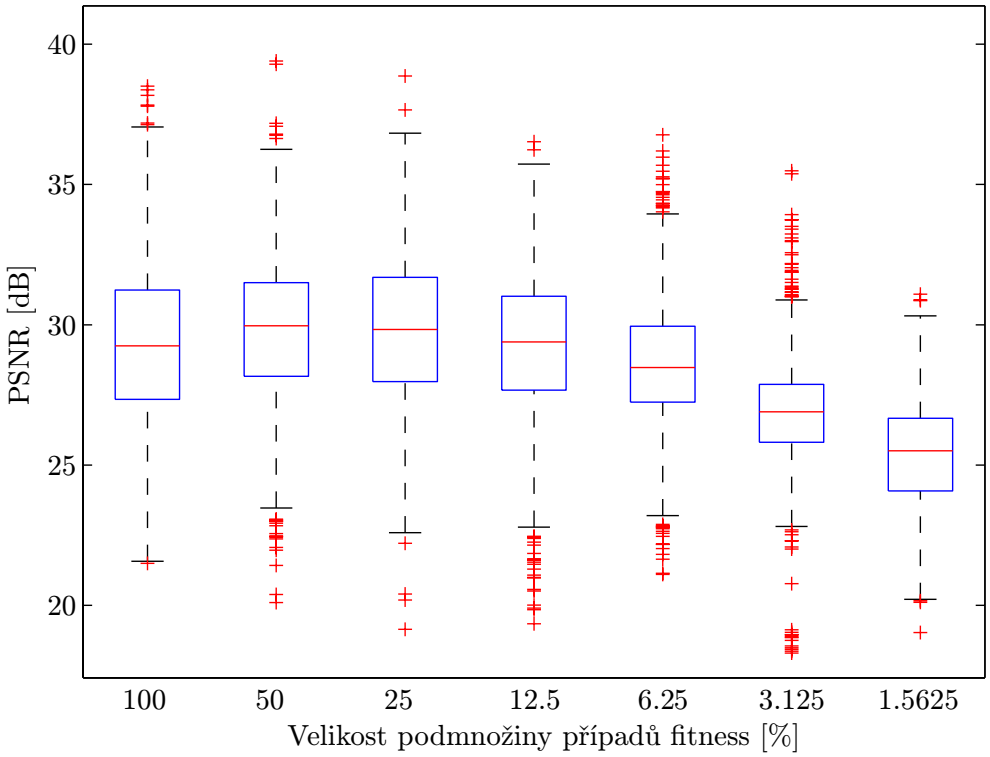
Následující grafy zobrazují statistiky PSNR spočítané pro 100 evolvovaných filtrů (pro každou konfiguraci úlohy bylo spuštěno 100 běhů evoluce) na sadě 14 testovacích obrazů. Testovací úlohou byl návrh filtrů šumu typu sůl a pepř o intenzitách 5–30 %. Statistika v každém sloupci je určena na základě celkem 1 400 hodnot fitness. Velikost FCS 100 % značí standardní evoluční přístup.



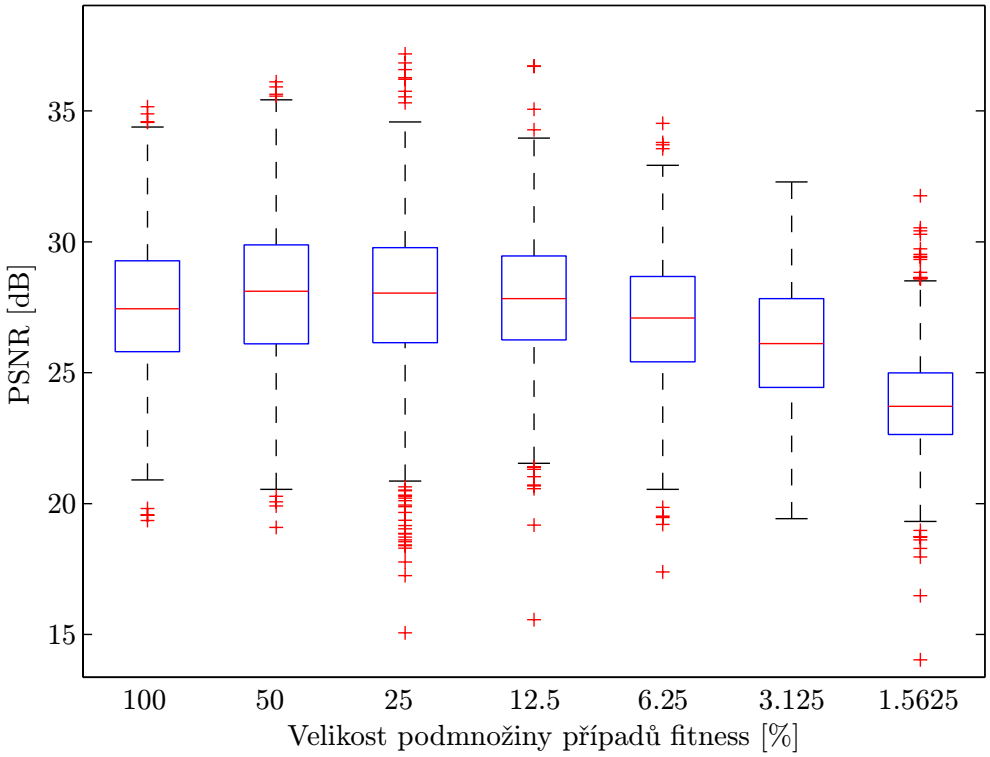
Obrázek B.1: Statistiky PSNR pro šum typu sůl a pepř 5 %.



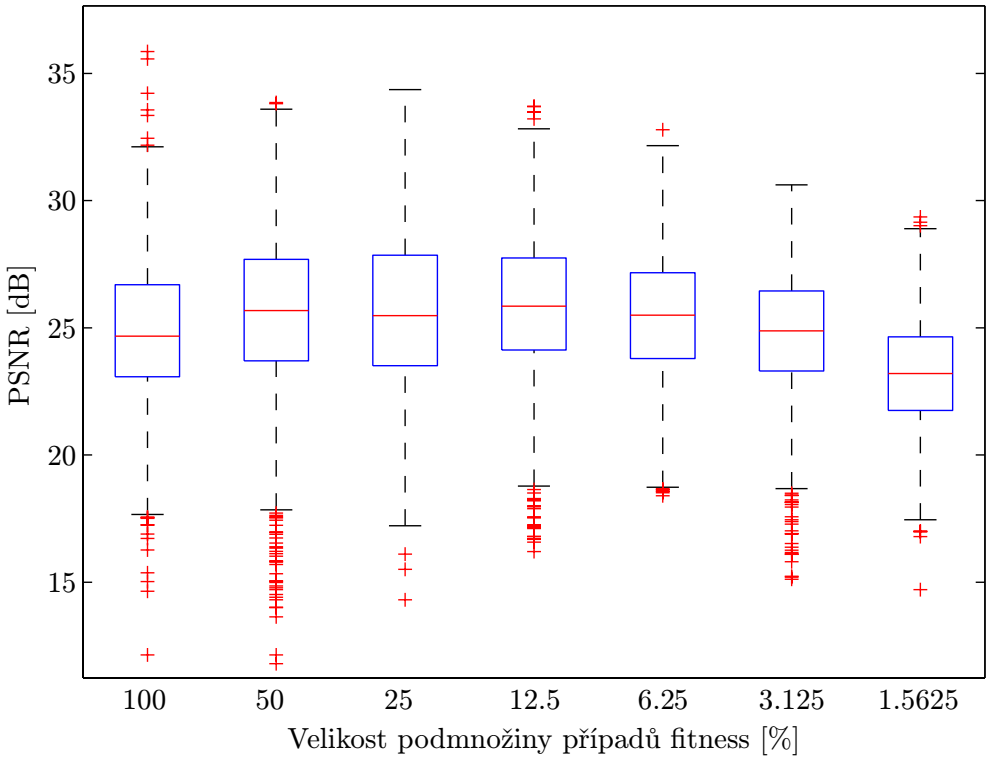
Obrázek B.2: Statistiky PSNR pro šum typu sůl a pepř 10 %.



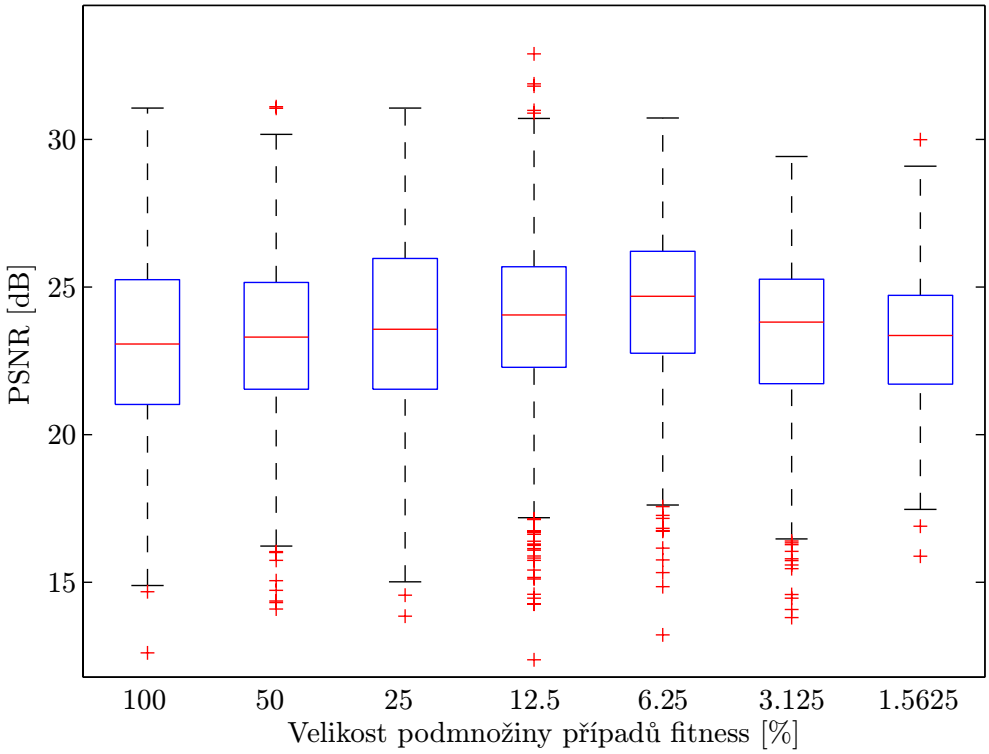
Obrázek B.3: Statistiky PSNR pro šum typu sůl a pepř 15 %.



Obrázek B.4: Statistiky PSNR pro šum typu sůl a pepř 20 %.



Obrázek B.5: Statistiky PSNR pro šum typu sůl a pepř 25 %.



Obrázek B.6: Statistiky PSNR pro šum typu sůl a pepř 30 %.

Příloha C

Obsah CD

Přiložené CD obsahuje veškeré zdrojové kódy včetně pomocných nástrojů a výsledky experimentů v následující adresářové struktuře:

- `archive` – archiv ručně nalezených filtrů,
- `bib` – použitá literatura,
- `bin` – binární soubory (konfigurační řetězce FPGA a program pro procesory MicroBlaze),
- `data` – trénovací obrazy,
- `eval` – výsledky experimentů (veškeré vyevolvované chromozomy, zaznamenané průběhy fitness, MATLAB skript pro vyhodnocení výsledků),
- `img` – sada testovacích obrazů,
- `src` – zdrojové soubory:
  - `fpga` – zdrojové soubory periferií (VHDL, C), projekty v prostředí Xilinx ISE,
  - `messages` – LCM zprávy (definice zpráv, vygenerované implementace v C a C#),
  - `microblaze` – zdrojové kódy softwaru pro procesory MicroBlaze,
  - `misc` – ostatní soubory (časová analýza),
  - `pc` – zdrojové kódy obslužných aplikací (C#),
  - `scripts` – inicializační skripty (Tcl),
- `thesis` – technická zpráva (TEX),
- `tools` – pomocné nástroje,
  - `cgp` – softwarová implementace (ko)evolučního návrhu filtrů (`C++`),
  - `viewer` – prohlížeč chromozomů CGP (autor Zdeněk Vašíček).