VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POLYMORFNÍ OBRAZOVÉ FILTRY

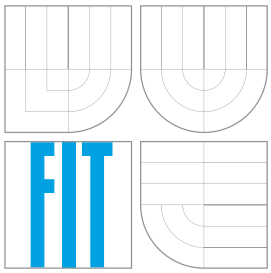
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH SALAJKA

BRNO 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

POLYMORFNÍ OBRAZOVÉ FILTRY

POLYMORPHIC IMAGE FILTERS

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE
AUTHOR

Bc. VOJTĚCH SALAJKA

VEDOUCÍ PRÁCE
SUPERVISOR

Doc. Ing. LUKÁŠ SEKANINA, Ph.D.

BRNO 2011

Vysoké učení technické v Brně - Fakulta informačních technologií

Ústav počítačových systémů Akademický rok 2010/2011

Zadání diplomové práce

Řešitel: **Salajka Vojtěch, Bc.**
Obor: Bioinformatika a biocomputing
Téma: **Polymorfní obrazové filtry**
Polymorphic Image Filters
Kategorie: Umělá inteligence

Pokyny:

1. Seznamte se s problematikou polymorfních číslicových obvodů.
2. Seznamte se s metodou evolučního návrhu obrazových filtrů.
3. Vypracujte studii na témata definovaná v bodech 1 a 2.
4. Navrhněte a implementujte algoritmus pro evoluční návrh polymorfních obrazových filtrů.
5. Na zvolených úlohách demonstrujte činnost algoritmu.
6. Shrňte dosažené výsledky.

Literatura:

- Dle pokynů vedoucího.

Při obhajobě semestrální části diplomového projektu je požadováno:

- Splnění bodů 1 až 3 zadání.

Podrobné závazné pokyny pro vypracování diplomové práce naleznete na adrese http://www.fit.vutbr.cz/info/szz/

Technická zpráva diplomové práce musí obsahovat formulaci cíle, charakteristiku současného stavu, teoretická a odborná východiska řešených problémů a specifikaci etap, které byly vyřešeny v rámci ročníkového a semestrálního projektu (30 až 40% celkového rozsahu technické zprávy).

Student odevzdá v jednom výtisku technickou zprávu a v elektronické podobě zdrojový text technické zprávy, úplnou programovou dokumentaci a zdrojové texty programů. Informace v elektronické podobě budou uloženy na standardním nepřepisovatelném paměťovém médiu (CD-R, DVD-R, apod.), které bude vloženo do písemné zprávy tak, aby nemohlo dojít k jeho ztrátě při běžné manipulaci.

Vedoucí: **Sekanina Lukáš, doc. Ing., Ph.D.**, UPSY FIT VUT
Datum zadání: 20. září 2010
Datum odevzdání: 25. května 2011

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
Fakulta informačních technologií
Ústav počítačových systémů a sítí
612 66 Brno, Božetěchova 2
L.S.

doc. Ing. Zdeněk Kotásek, CSc.
vedoucí ústavu

Studium přerušeno ze zdravotních důvodů od 1. června 2011 do 31. května 2012.
Termín obhajoby: 20. června 2012

Abstrakt

Tato práce se zabývá evolučním návrhem polymorfních obrazových filtrů. Studie zahrnuje polymorfní obvody, jejich teoretický základ a praktické oblasti nasazení. Dále se zabývá kartézským genetickým programováním, které je použitelné pro evoluční návrh některých typů obrazových filtrů. V dalších částech je specifikován evoluční algoritmus pro návrh polymorfních obrazových filtrů. Je popsána implementace tohoto algoritmu ve dvou verzích – neakcelerovaná běžící pouze na CPU a akcelerovaná využívající pro svůj běh také GPU. Pomocí algoritmu je navrženo několik polymorfních obrazových filtrů.

Abstract

This thesis deals with the polymorphic image filter design. The study includes polymorphic circuits, their theoretical base and practical applications. It further focuses on the cartesian genetic programming that can be used for an evolutionary design of some types of image filters. The thesis continues with the specification of the evolutionary algorithm to be used for the design of the polymorphic image filters. The implementation of the algorithm is described in two versions – a standard one running only on a CPU and an accelerated one that partially uses the GPU. Several polymorphic image filters are designed by means of the algorithm.

Klíčová slova

polymorfní obvody, kartézské genetické programování, obrazové filtry, evoluční návrh

Keywords

polymorphic circuits, cartesian genetic programming, image filters, evolutionary design

Citace

Vojtěch Salajka: Polymorfní obrazové filtry, diplomová práce, Brno, FIT VUT v Brně, 2011

Polymorfní obrazové filtry

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Doc. Ing. Lukáše Sekaniny, Ph.D.

.....
Vojtěch Salajka
21. května 2011

Poděkování

Tímto chci poděkovat panu Doc. Ing. Lukáši Sekaninovi, Ph.D. za jeho trpělivé vedení při tvorbě této diplomové práce. Dále chci poděkovat panu Simonu Hardingovi za poskytnutí testovacího obrázku.

Obsah

Kapitola 1

Úvod

Obrazové filtry v kontextu této práce jsou algoritmy pro zpracování digitálního obrazu. Jako takové nacházejí uplatnění v mnoha oblastech, jako je věda a výzkum, průmysl, lékařství, spotřební elektronika, umění a mnohé další. Mezi konkrétní příklady uplatnění obrazových filtrů patří redukce šumu, úprava kontrastu a barev, komprese obrazových dat, zvláštní efekty jak ve statických obrázcích, tak ve videu a filmu, segmentace obrazu, rozpoznávání textu (OCR) atd.

Z celé množiny obrazových filtrů si všimneme zejména skupiny lokálních obrazových filtrů, které používají pro výpočet výsledného pixelu obrazu zdrojový pixel a jeho blízké okolí. Návrh některých filtrů tohoto typu je dostatečně netriviální a zároveň prohledávací prostor je rozumně velký, takže je vhodné a prakticky možné pro jejich vývoj nasadit evoluční algoritmy.

Další oblastí, kterou se tento projekt zabývá, jsou polymorfní obvody. Tyto obvody mění na základě vnějších podmínek své vlastnosti dobře definovaným způsobem. Mají své využití v prostředí, kde se očekávají výrazné změny vnějších podmínek, ať už je to poměrně odtažitá oblast vesmírného průzkumu nebo naopak snadno představitelná možnost signalizace nízké energie baterií ve spotřební elektronice.

Polymorfní obrazové filtry jsou obrazové filtry, které obsahují polymorfní výpočetní elementy. Díky tomu je možné jedním takovým filtrem provádět jednu ze dvou různých operací v závislosti na režimu výpočetních elementů. Tyto operace mohou být podobné, ale i naprosto odlišné. Cílem tohoto projektu je navrhnout odvozenou metodu kartézského genetického programování pro návrh polymorfních obrazových filtrů. V rámci tohoto úkolu je třeba specifikovat a implementovat tento algoritmus a zejména ověřit, že pomocí něj lze skutečně polymorfní obrazové filtry navrhovat. Praktickým výsledkem testů by mělo být několik konkrétních polymorfních filtrů vytvořených tímto evolučním algoritmem.

Na první kapitolu, kterou právě čtete, navazuje druhá kapitola zabývající se polymorfními obvody, oblastmi jejich využití, teoretickým základem jejich návrhu a praktickými příklady konkrétních realizací. Třetí kapitola se zabývá kartézským genetickým programováním, což je zásadní evoluční algoritmus jak pro evoluční návrh logických obvodů (konvenčních i polymorfních), tak pro evoluční návrh obrazových filtrů. Čtvrtá kapitola pojednává o metodách filtrace obrazu. Pátá kapitola shrnuje metody využívané v evolučním návrhu konvenčních (nepolymorfních) obrazových filtrů.

V šesté kapitole je specifikován evoluční algoritmus pro návrh polymorfních obrazových filtrů. Sedmá kapitola popisuje implementaci tohoto algoritmu. Osmá kapitola popisuje průběh testů a výsledné navržené filtry. Závěrečné shrnutí práce je provedeno v deváté kapitole.

Diplomová práce navazuje na semestrální projekt, ze kterého přebírá kapitoly 2, 3, 4, 5 a části kapitol 1 a 6.

Kapitola 2

Polymorfní obvody

Polymorfní obvody jsou nekonvenční logické komponenty, které mohou měnit své logické funkce v závislosti na měnícím se prostředí [13]. Těmito změnami prostředí mohou být například změny teploty, napájecího napětí, osvětlení atd. Polymorfní obvod má specifikovány typicky dvě, ale někdy i více logických funkcí. První je definována pro první stav prostředí, druhá je definována pro druhý stav. Takto jsou vlastně definovány dva režimy, ve kterých může polymorfní hradlo pracovat. Pokud je prostředí ve stavu, který v rámci tolerance nedosahuje hodnot, pro které je polymorfní obvod specifikován, jeho výstup je nedefinovaný.

2.1 Praktické využití

Mezi praktické aplikace polymorfních obvodů patří automatické řízení spotřeby při poklesu napětí baterie, dále tvorba watermarků v hardware, které za běžných podmínek nejsou uživateli viditelné. Možné je i využití k ochraně proti reverznímu inženýrství. Dalšími aplikacemi je použití v inteligentních senzorech, adaptivních systémech a v diagnostce a testování logických obvodů [13]. Další oblastí využití polymorfních čipů je tvorba odolných systémů pro nasazení ve vesmírných nebo mořských hlubinách [5].

2.2 Teoretické výpočetní možnosti

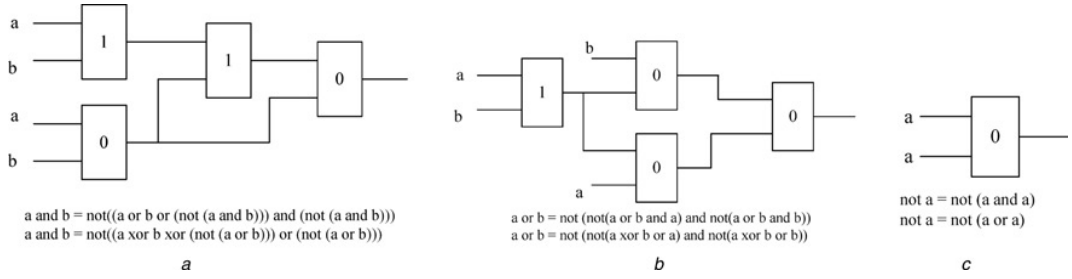
Při práci s polymorfními hradly může být důležité nejen vytvořit obvod, který mění své vlastnosti na základě měnícího se prostředí, ale i obvod, který si zachová stejné vlastnosti v obou stavech prostředí.

Uvažujme sadu tří polymorfních hradel [5]:

- polymorfní hradlo 0: NAND/NOR [1]
- polymorfní hradlo 1: OR/XOR
- polymorfní hradlo 2: XOR/((NOT a) AND b)

V [5] je dokázáno, že je možné vytvořit buňky AND, OR a NOT pouze pomocí polymorfních hradel 0 a 1, a tyto buňky si zachovávají svoji funkci jak v režimu 1, tak v režimu 2. Ukázka takových buněk je na obrázku 2.1.

[1] Zápis NAND/NOR značí, že polymorfní hradlo má v režimu 1 vlastnosti hradla NAND a v režimu 2 vlastnosti hradla NOR. Ostatní polymorfní hradla jsou zapisována obdobně.



a b c

Obrázek 2.1: Příklady buněk AND (a), OR (b) a NOT (c), které zachovávají svoji funkcionalitu v obou režimech (převzato z [5]).

Jakmile máme k dispozici prvky AND, OR a NOT, můžeme pomocí nich vytvořit libovolný kombinační obvod [5].

Schopnost polymorfních obvodů modelovat dvě různé funkce je minimálně stejně důležitá jako jejich schopnost zachovat si stejnou funkci v obou režimech. V [5] je rovněž dokázáno, že pomocí polymorfních hradel 0 a 1 je možné vytvořit logický obvod, který modeluje dvě libovolné funkce f_1 a f_2.

Tyto poznatky jsou velmi důležité při návrhu polymorfních obvodů. Pokud ovšem chceme dosáhnout efektivních výsledků, je nutné použít evoluční přístup [5]. V takovém případě se typicky použije algoritmus CGP popsaný v kapitole 3. Pokud je cílem minimalizovat počet hradel, výpočet hodnoty fitness probíhá podle následujícího algoritmu zveřejněného v článku [10]:

1. Všechna hradla se nastaví do režimu 1.
2. Na vstupy se přivedou všechny možné vstupní kombinace a spočítá se počet korektních bitů B_1, kde korektnost se posuzuje porovnáním s výstupem požadované funkce f_1.
3. Všechna hradla se nastaví do režimu 2.
4. Na vstupy se přivedou všechny možné vstupní kombinace a spočítá se počet korektních bitů B_2, kde korektnost se posuzuje porovnáním s výstupem požadované funkce f_2.
5. Proběhne výpočet $F_1 = B_1 + B_2$.

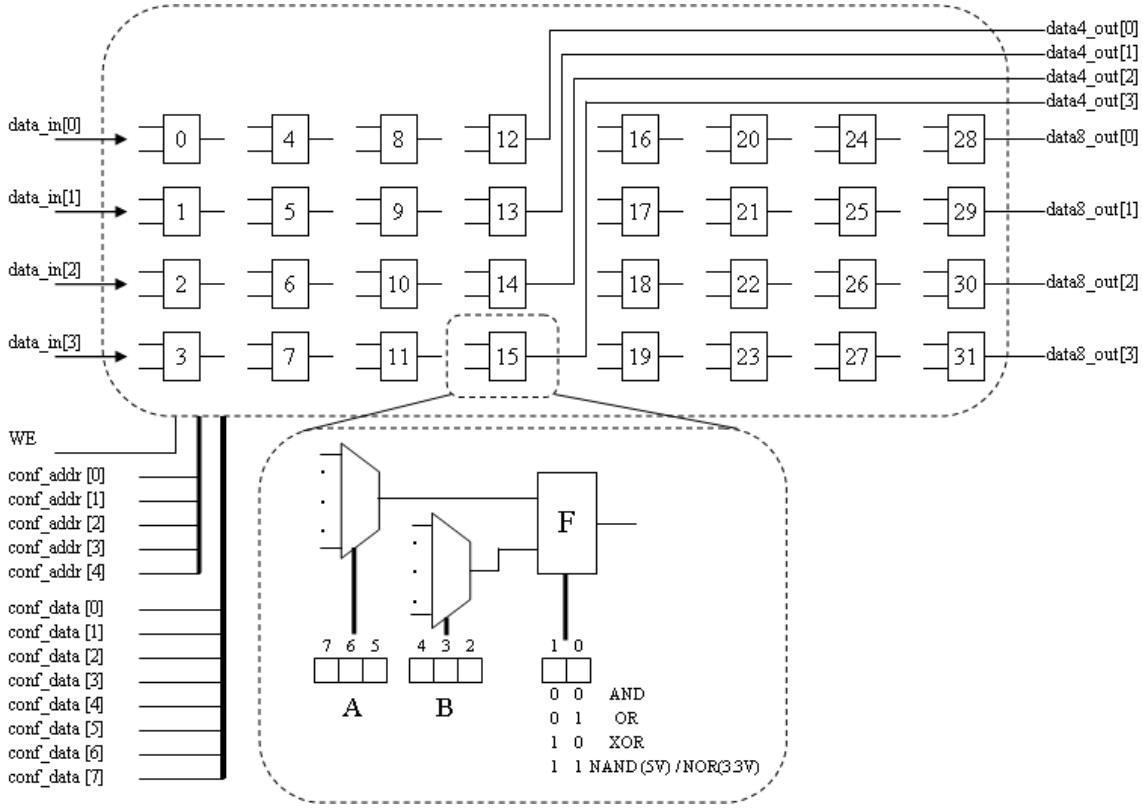
Pokud je dosaženo požadovaného chování (tzn. pokud funkce obvodu v režimu 1, resp. režimu 2 odpovídá f_1, resp. f_2), vypočítá se hodnota

$$F_2 = F_1 + u - g, \tag{2.1}$$

kde g je počet využitých hradel a u je maximální počet hradel.

2.3 Příklady

Polymorfní obvody mají velký potenciál, avšak množství fyzických realizací za teoretickým výzkumem v této oblasti pokulhává. Nicméně podle [12] existuje polymorfní AND/OR hradlo, kde aktuální funkce závisí na teplotě prostředí. Hradlo je v režimu AND při teplotě 27 °C a v režimu OR při teplotě 127 °C.



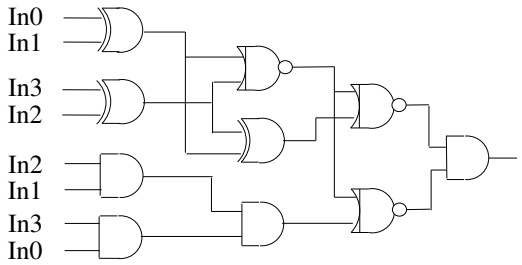
Obrázek 2.2: Architektura čipu REPOMO32 (převzato z [12]).

Článek [12] informuje o existenci dalších dvou polymorfních hradel, jejichž aktuální funkce je určena hodnotou napájecího napětí. První z nich prezentovala Stoicova skupina. Hradlo má režimy NAND/NOR a hodnoty napájecího napětí jsou Vdd = 1,8 V v NOR režimu a Vdd = 3,3 V v NAND režimu. Při výrobě byla použita HP 0,5 mikronová CMOS technologie.

Druhé hradlo je rovněž NAND/NOR. Bylo vyvinuto na FIT VUT v rámci skupiny zabývající se problematikou Evolvable Hardware. Při výrobě byla použita AMIS 0,7 mikronová CMOS technologie. Napájecí napětí nabývá hodnot 3,3 V a 5 V.

Stejná výzkumná skupina rovněž vytvořila rekonfigurovatelný polymorfní čip nazvaný REPOMO32 [12]. Je vyroben 0,7 mikronovou AMIS CMOS technologií. Obsahuje 32 rekonfigurovatelných logických elementů (CLE), které jsou uspořádány do mřížky 8×4. Každý z nich může mít jednu z funkcí AND, OR, XOR a polymorfní NAND/NOR. Tato polymorfní funkce jako jediná mění vlastnosti v závislosti na napájecím napětí, ostatní jsou na hodnotě napájecího napětí nezávislé. Vstup jednotlivých CLE může být připojen na výstup kteréhokoli CLE z těsně předcházejících 2 sloupců. Čip má 4 primární vstupy a 8 primárních výstupů. Vstupy CLE, které jsou v prvních 2 sloupcích, mohou být připojeny k primárním vstupům. Čtyři primární výstupy jsou pevně připojeny k výstupům CLE ve čtvrtém sloupci a další čtyři k výstupům v osmém sloupci. Schéma čipu je zobrazeno na obrázku 2.2.

REPOMO32 je první vyvinutý rekonfigurovatelný polymorfní čip a umožňuje zkoumání vlastností polymorfních čipů novými způsoby, například evolučními postupy [12].



Obrázek 2.3: Detektor sudé parity při napájecím napětí 1,2 V a indikátor majority při napájecím napětí 3,3 V (převzato z [13]).

Jako zástupce polymorfních obvodů realizujících dvě různé funkce uveďme obvod, který při napájecím napětí 1,2 V detekuje sudou paritu a při napájecím napětí 3,3 V indikuje majoritu [13]. Schéma tohoto obvodu je na obrázku 2.3.

Kapitola 3

Kartézské genetické programování

Kartézské genetické programování (CGP) vychází ze standardního genetického programování (GP). Mají společný účel, pro který vznikly, a tím je evoluční návrh výpočetních funkcí. Liší se od GP například způsobem reprezentace genotypu, převodem na fenotyp nebo používaným genetickým operátorem [6].

CGP představili Miller s Thomsonem v článku [6]. Snaží se odstranit nevýhody standardní varianty genetického programování, kterými jsou proměnná délka chromozomu a nízká redundance.

V klasické variantě GP má fenotyp tvar parsovacího stromu LISPu a hlavním používaným genetickým operátorem je křížení [6]. Naproti tomu v CGP je fenotypem acyklický orientovaný graf a hlavním používaným genetickým operátorem je mutace.

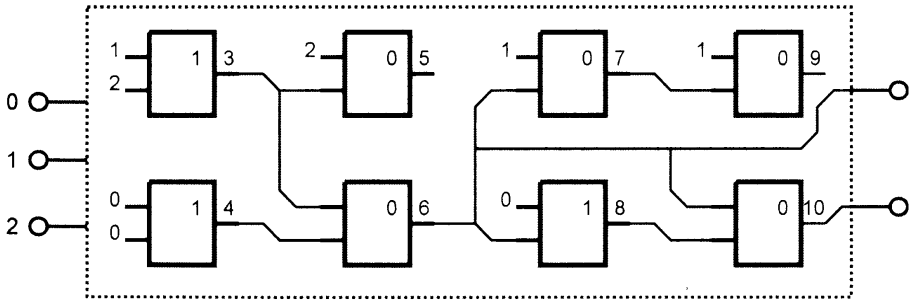
3.1 Vztah genotypu a fenotypu v CGP

Obvod reprezentovaný CGP je tvořen n_i primárními vstupy, n_o primárními výstupy a polem programovatelných elementů velikosti $n_c \times n_r$, kde n_c je počet sloupců a n_r je počet řádků. Elementy tedy tvoří mřížku a jsou adresovány v kartézských souřadnicích (odtud je odvozen název metody) [6]. Každý element má přiřazenu jednu z množiny předdefinovaných funkcí Γ, jejichž počet je n_f. Každá z těchto funkcí má až n_a argumentů [14]. Pokud má funkce méně argumentů než n_a, nepotřebné vstupy elementu se ignorují.

Vstupy programovatelných elementů mohou být připojeny na primární vstupy obvodu nebo na výstupy jiných elementů. Primární výstupy obvodu reprezentované CGP mohou být připojeny přímo na primární vstupy nebo na výstupy programovatelných elementů. Propojení však nemůže být úplně libovolné, podléhá několika omezením.

Při propojování elementů lze připojit každý ze vstupů výpočetního elementu buď na výstupy výpočetního elementu v některém z předchozích sloupců, nebo na primární vstupy obvodu. Proto nemohou vznikat zpětné vazby a zpoždění obvodu je tímto způsobem shora omezeno počtem sloupců n_c.

Parametr L určuje další omezení, a to maximální vzdálenost sloupců, ve kterých lze propojovat programovatelné elementy. Pokud například $L = 1$ (což je minimální hodnota tohoto parametru), lze vstup elementu připojit pouze na výstup elementu v těsně předchozím sloupci a primární vstupy lze připojit pouze na elementy v prvním sloupci. Pokud naopak $L = n_c$, je možné vstupy každého elementu připojovat na výstupy elementu ve kterémkoliv z předchozích sloupců. Extrémním případem propojitelnosti je CGP reprezentace s parametry $L = n_c$ a $n_r = 1$. Tím je umožněno propojovat elementy libovolně v rámci



Obrázek 3.1: Příklad kandidátního obvodu v CGP (převzat z [14]) s parametry $n_r = 2$, $n_c = 4$, $n_i = 3$, $n_o = 2$, $n_a = 2$, $n_f = 2$, $L = 3$, $\Gamma = \{AND(0), OR(1)\}$. Uzly 5 a 9 nejsou ve fenotypu využity. Chromozom: 1,2,1, 0,0,1, 2,3,0, 3,4,0, 1,6,0, 0,6,1, 1,7,0, 6,8,0, 6,10. Poslední dvě hodnoty chromozomu určují zapojení výstupů.

ostatních omezení (připomeňme, že například není možné použití zpětné vazby) [14]. Pro doplnění lze uvést, že existují i jiné varianty CGP, kde je použití zpětné vazby možné [6].

Jak již bylo řečeno, genotyp CGP se vyznačuje pevnou délkou chromozomu. Chromozom je tvořen vektorem celočíselných hodnot. Počet prvků tohoto vektoru je Λ_{CGP} [14], kde

$$\Lambda_{CGP} = n_r n_c (n_a + 1) + n_o. \tag{3.1}$$

Význam hodnot chromozomu je ten, že první programovatelný element je nakonfigurován n_a hodnotami, které určují, odkud jsou přivedeny jeho vstupy. Pro první element pak následuje ještě jedna hodnota, která určuje, jakou funkci z množiny Γ element představuje. Toto je zopakováno pro všechny další programovatelné elementy, které se berou postupně po sloupcích.

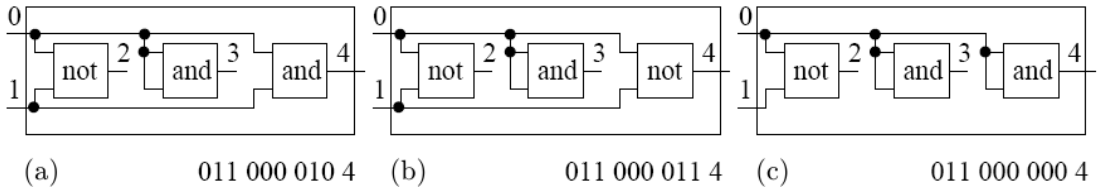
Místa, kam lze připojit vstupy programovatelných elementů, jsou indexována tak, že nejprve jsou indexovány primární vstupy obvodu. Následně jsou indexovány výstupy programovatelných elementů, a to po sloupcích.

Následuje n_o hodnot, které určují, kam jsou připojeny primární výstupy obvodu. Jejich význam je stejný jako význam konfigurace vstupů programovatelných elementů. Tím chromozom končí.

Vztah mezi genotypem a fenotypem ilustruje obrázek 3.1 [14].

Redundance v CGP má význam při neutrálních změnách genotypu. Tyto změny se totiž mohou projevit až později při větším množství a může se tak najít lepší řešení, které je zároveň výrazněji odlišné od aktuálně nalezeného [6].

Obvod reprezentovaný CGP může obsahovat několik forem redundance. Prvním typem je redundance uzlů, jejichž výstup se neprojeví na primárních výstupech obvodu. Další forma redundance je funkcionální redundance. K té dochází v případě, že několik propojených uzlů tvoří část složitější funkce, kterou ale lze vytvořit jinak pomocí menšího počtu uzlů. Třetím typem redundance je tzv. vstupní redundance. Ta se projevuje u funkcí, které reálně vyžadují méně vstupů, než je počet vstupů programovatelného elementu. V takovém případě se zbytek vstupů neprojeví [6].



Obrázek 3.2: Rodičovský obvod před mutací (a), mutace jednoho hradla (b), mutace vstupu hradla (c). Parametry CGP jsou následující: $\Gamma = \{and, not\}$, $n_c = 3$, $n_r = 1$, $n_i = 2$, $n_o = 1$, $n_a = 2$, $n_f = 2$ (převzato z [9]).

3.2 Operátor mutace

Operátor mutace pracuje tak, že při generování chromozomu potomka z chromozomu rodiče je v chromozomu nahrazen jeden náhodně vybraný gen hodnotou, která je náhodně vybrána z rozsahu platného pro daný gen. Někdy se používá vícenásobná mutace, kdy se typicky nahrazují dva nebo tři geny [9].

Ukázky mutace chromozomu a odpovídající změny ve fenotypu jsou na obrázku 3.2.

3.3 Prohledávací algoritmus

Prohledávací algoritmus pracuje tak, že na počátku náhodně vygeneruje počáteční populaci o velikosti $\lambda+1$ jedinců. Obvykle $\lambda = 4$ [14]. Následně se jedinci ohodnotí (vypočítá se fitness funkce). Vybere se nejlepší jedinec a přesune se do další generace. Dále se vygeneruje λ jeho potomků, kteří jsou od něj odvozeni operátorem mutace. Tím je vytvořena nová populace.

V případě, že dva či více jedinců dosáhne shodného ohodnocení, které je zároveň nejlepší, vybere se z nich ten, který nebyl rodič v minulé populaci. Tím je zajištěna genetická diverzita [14]. Celý postup se opakuje tak dlouho, až je dosaženo požadované kvality řešení nebo předem zvoleného maximálního počtu cyklů algoritmu.

Kapitola 4

Filtrace obrazu

V počítačové grafice, počítačovém vidění, zpracování obrazových dat, umělé inteligenci a v mnoha dalších oblastech (fyzika, lékařství) je nutné zpracovávat rastrová data. Během vývoje informatiky bylo objeveno mnoho postupů, které umožňují nějakým způsobem obraz poloautomaticky zpracovat. Vždy závisí na typu úlohy, jaký konkrétní postup se použije. Při zpracovávání obrazových dat zachycených z kamer, fotoaparátů a skenerů se uplatňují například filtry pro vylepšení obrazu, kam patří redukce šumu, zvýraznění barev, kontrastu atd. V oblasti počítačového vidění a umělé inteligence se používají např. filtry pro detekci hran, úseček, segmentaci obrazu atd. Mezi obrazové filtry patří například i filtry jako dolní a horní propust, které mají použití ve velkém množství různých oblastí.

Obecně lze obrazový filtr, tedy funkci pro zpracování rastrového obrazu, formálně zapsat jako

$$f(o) : I \to I, \tag{4.1}$$

kde o je vstupní rastrový obraz a I je množina všech možných rastrových obrazů [18].

Existuje alternativní popis filtrační funkce, a to po pixelech. Zapíše se jako

$$f(o, x, y) : I \times \mathbb{N}^0 \times \mathbb{N}^0 \to X, \tag{4.2}$$

kde o je vstupní obraz, x a y jsou souřadnice výstupního pixelu, I je množina všech možných rastrových obrazů a X je obor hodnot výstupního pixelu [18].

Obrazové filtry se dělí podle mnoha vlastností. Jednou z nich je linearita.

4.1 Lineární filtry

Lineární filtry jsou takové filtry, které splňují princip superpozice. To znamená, že filtr aplikovaný na součet dvou signálů dá stejný výsledek jako součet výsledků filtrace každého signálu zvlášť [15]. Formálně tedy lze zapsat, že pro lineární filtry platí

$$f(o_1) + f(o_2) = f(o_1 + o_2), \tag{4.3}$$

kde o_1 a o_2 jsou obrazové signály a f je lineární filtrační funkce [18].

Ještě je vhodné podotknout, že filtry obecně, a tedy i lineární filtry, mohou zpracovávat i jiný než jen dvourozměrný obrazový signál, a to například jednorozměrný zvukový, elektromagnetický, dokonce například trojrozměrný videosignál.

Existuje několik obecných druhů lineárních filtrů [16]:

- dolní propust,

- horní propust,
- pásmová propust,
- pásmová zádrž

a další.

Mezi konkrétnější případy lineárních filtrů používaných pro zpracování dvourozměrného rastrového obrazu patří (příklady) [3]:

- konvoluce,
- derivační filtry,
- řiditelné (steerable) filtry a
- Wienerův filtr.

4.1.1 Konvoluce

Dvourozměrná konvoluce je přirozeným rozšířením konvoluce jednorozměrné. Formálně lze zapsat takto [3]:

$$g(x,y) = \sum_{k=-\infty}^{\infty} \sum_{l=-\infty}^{\infty} f(k,l)h(x-k, y-l), \tag{4.4}$$

kde f je obrázek a h je dvourozměrný filtr.

Mezi nejběžnější konvoluční operace patří rozostření a zaostření. Tyto dvě konkrétní operace mohou být zrealizovány pomocí 2D konvoluční matice, popřípadě pomocí dvou 1D konvolučních vektorů – horizontálního a vertikálního. Výpočet pomocí dvou jednorozměrných konvolučních vektorů je efektivnější. Jako praktický příklad uveďme rozostřující konvoluční filtr používající matici

$$\begin{bmatrix} 0{,}0625 & 0{,}125 & 0{,}0625 \\ 0{,}125 & 0{,}25 & 0{,}125 \\ 0{,}0625 & 0{,}125 & 0{,}0625 \end{bmatrix}.$$

Oba jednorozměrné vektory, kterými by jej bylo možné nahradit, by měly tvar $\begin{bmatrix} 0{,}25 & 0{,}5 & 0{,}25 \end{bmatrix}$ [3].

4.1.2 Derivační filtry

Definice derivace spojitého signálu $f(x)$ [3] je

$$D\{f(x)\} = \lim_{\varepsilon \to 0} \frac{f(x+\varepsilon) - f(x)}{\varepsilon}. \tag{4.5}$$

Rastrový obrázek je ale z tohoto pohledu signál nespojitý, který je samplovaný v místech středů jednotlivých pixelů z původně spojitého signálu.

Jeden ze způsobů, jak převést nespojitý signál na signál spojitý, je popsán vztahem

$$f(x) = f[x] * h(x), \tag{4.6}$$

kde $f(x)$ představuje spojitý signál, $f[x]$ diskrétně samplovaný signál,

$$h(x) = \frac{\sin(\frac{\pi x}{T})}{\frac{\pi x}{T}} \tag{4.7}$$

a $*$ je operátor konvoluce [3].

4.1.3 Řiditelné (steerable) filtry

Nyní víme, jak vypočítat parciální derivaci dvourozměrného signálu v horizontálním nebo vertikálním směru. Parciální derivace v libovolném směru se vypočte pomocí řiditelných filtrů. Vztah pro její výpočet je

$$f_\alpha(x,y) = \cos(\alpha) f_x(x,y) + \sin(\alpha) f_y(x,y), \tag{4.8}$$

kde $f_\alpha(x,y)$ je parciální derivace v bodě (x,y) ve směru daném úhlem α, $f_x(x,y)$ je parciální derivace v horizontálním směru a $f_y(x,y)$ parciální derivace ve vertikálním směru [3].

4.1.4 Wienerův filtr

Wienerův filtr je určen pro odstraňování náhodného šumu. Je to filtr typu dolní propusti [3].

4.2 Nelineární filtry

Nelineární filtry jsou ty, které nesplňují princip superpozice. Nelinearita může vznikat například vlivem saturace, mocnin, řazením nebo výběrem hodnot [18].

Jedním ze způsobů tvorby nelineárních filtrů je použití nelineární operace na výsledek několika lineárních filtrů. Mezi tyto operace patří výběr maxima, součet absolutních hodnot a délka vektoru tvořeného výsledky filtrů. Příkladem tohoto druhu nelineárního filtru je Sobelův filtr, který používá poslední uvedenou operaci [18].

Zde je několik konkrétních příkladů různých typů nelineárních filtrů [3, 18]:

- všesměrová detekce hran,
- Sobelův filtr,
- mediánový filtr,
- rank-order filtr,
- prahování a
- dither.

4.2.1 Všesměrová detekce hran

Derivace je základem pro další aplikaci, kterou je detekce hran. Hrana je zhruba definována jako oblast s velkou změnou intenzity v nějakém směru. Je to tedy délka vektoru gradientu $\nabla(x,y)$, kterou lze vypočítat [3] jako

$$|\nabla(x,y)| = \sqrt{f_x^2(x,y) + f_y^2(x,y)}. \tag{4.9}$$

Je důležité podotknout, že všesměrová detekce hran prováděná uvedeným způsobem je nelineární filtr, avšak pro část výpočtů používá lineární filtry.

4.2.2 Sobelův filtr

Sobelův filtr je konkrétním příkladem všesměrové detekce hran. Jeho základ tvoří lineární filtr f_1 definovaný konvoluční maskou

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

a lineární filtr f_2 definovaný konvoluční maskou

$$\begin{bmatrix} -1 & -1 & -1 \\ 0 & 0 & 0 \\ 1 & 1 & 1 \end{bmatrix}.$$

Jejich výsledky se složí pomocí vzorce

$$S = \sqrt{f_1^2 + f_2^2} \text{ [18].} \tag{4.10}$$

4.2.3 Mediánový filtr

Výsledek pro mediánový filtr se vypočte tak, že se spočte medián pixelů v okolí daného pixelu, tzn. hodnota uprostřed seřazené posloupnosti. Využívá se například ke zjemnění obrázku. Jiné využití je odstranění šumu typu sůl a pepř (nazývaný také výstřelový šum) [3].

4.2.4 Rank-order filtr

Vypočítá se podobně jako u mediánového filtru, ale výsledek se vybírá z obecně jiného místa seřazené posloupnosti než ze středu [18] (rank-order filtr je zobecněním mediánového filtru).

4.2.5 Prahování

Prahování je jeden z algoritmů pro snížení barevné hloubky obrázku. Jako příklad poslouží převod obrázku ve stupních šedi na černobílý obrázek. Nejprve se zvolí práh. Pokud je hodnota původního pixelu menší než tento práh, bude barva nového pixelu černá, jinak bílá [18].

4.2.6 Dither

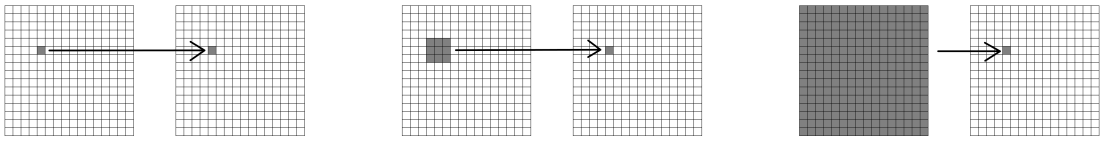
Dither je další z postupů používaných ke snížení barevné hloubky obrázků. Pracuje se záměrným vložením šumu pro náhodnou distribuci kvantizační chyby [17].

Dither je obecné označení několika různých algoritmů, mezi které patří dither náhodný, polotónový, Bayer, Floyd-Steinberg, Burkes, Sierra, Atkinson a další [17].

4.3 Lokalita algoritmů

Algoritmy pro zpracování obrazu lze dělit podle lokality na tři skupiny [18]:

- bodové,



Obrázek 4.1: Bodový, lokální a globální algoritmus – schématické znázornění výpočtu hodnoty jednoho pixelu.

- lokální a
- globální.

Většina obrazových filtrů uvedených v sekcích 4.1 a 4.2 spadá do kategorie lokálních algoritmů.

4.3.1 Bodové algoritmy

U tohoto typu algoritmů se pro získání výsledného pixelu použije pouze jeden zdrojový pixel [18]. Mezi tyto algoritmy patří například úprava jasu a kontrastu nebo také prahování. Schématické znázornění toho, jak bodový algoritmus pracuje, je vidět na obrázku 4.1 vlevo.

4.3.2 Lokální algoritmy

Tento typ filtrů používá pro výpočet cílového pixelu zdrojový pixel a jeho malé pevně stanovené okolí [18]. Do této skupiny patří například konvoluce, Sobelův filtr a mediánový filtr. Schématické znázornění toho, jak lokální algoritmus pracuje, je vidět na obrázku 4.1 uprostřed.

4.3.3 Globální algoritmy

Jiné algoritmy než bodové a lokální se nazývají globální [18]. Pro výpočet výsledného pixelu výstupního obrazu mohou být využity až všechny pixely vstupního obrazu. Schématické znázornění toho, jak globální algoritmus pracuje, je vidět na obrázku 4.1 vpravo.

Kapitola 5

Evoluční návrh konvenčních obrazových filtrů

Evoluční návrh polymorfních obrazových filtrů musí logicky vycházet z návrhu filtrů nepolymorfních. Shrňme si dosavadní výsledky poměrně rozsáhlého výzkumu, který v této oblasti probíhá.

5.1 Evoluční návrh obrazových filtrů v FPGA

Hlavní důvod, proč používat hradlová pole FPGA v evolučních algoritmech, je akcelerace výpočtů. Existuje několik způsobů, jak evoluční algoritmus akcelerovat v FPGA.

5.1.1 Rekonfigurace FPGA

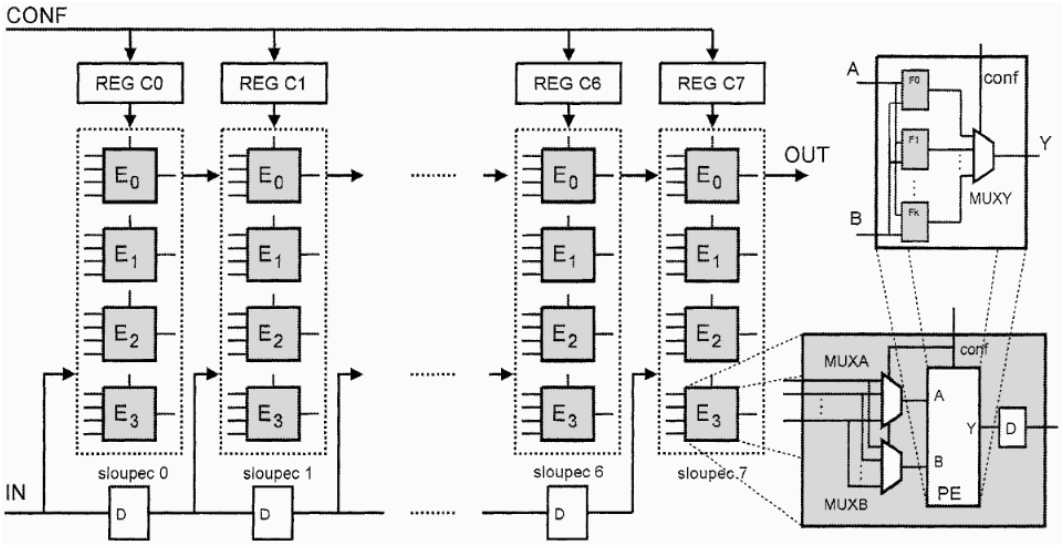
Prvním způsobem je použití FPGA pouze pro evaluaci kandidátních řešení, protože tato operace patří zejména v oblasti obrazových filtrů k výpočetně nejnáročnějším, ale zároveň implementačně poměrně jednoduchým. Tyto dvě vlastnosti ji tedy k výpočtu na FPGA předurčují. Ostatní části algoritmu v tomto případě běží na externím procesoru. Na FPGA se zašle pouze konfigurační řetězec, jsou z něj přečteny datové výstupy, ale hodnota fitness se vypočítá opět v externím procesoru [14].

Jinou možností je počítat na FPGA nejen výstup nakonfigurovaného filtračního obvodu, ale také hodnotu fitness. V některých případech si může FPGA samo generovat i vstupní testovací vektory, ale tato možnost se využívá zejména při návrhu logických obvodů a řadicích sítí, ale ne obrazových filtrů.

Pokud bychom chtěli provádět evoluční návrh kompletně v FPGA, narazíme na problémy s interní rekonfigurací FPGA čipů. Většinou ji buď nepodporují, nebo tato možnost není dostatečně zdokumentovaná. Například chybí popis formátu konfiguračního souboru a je nutné ho zjišťovat metodou reverzního inženýrství. Tak je tomu například u obvodu Virtex-II firmy Xilinx [14].

5.1.2 Virtuální rekonfigurovatelné obvody

Další možností využití FPGA pro evoluční návrh nejen obrazových filtrů jsou virtuální rekonfigurovatelné obvody (VRC). V FPGA je vytvořen obvod, který se dá rekonfigurovat. Je třeba rekonfigurovat výběr výpočetních funkcí a propojovací sítě. Děje se to zpravidla pomocí multiplexorů a vyhledávacích tabulek (LUT). To, jak jsou prvky uspořádány a



Obrázek 5.1: VRC pro evoluční návrh obrazových operátorů (převzato z [14]).

jaké funkce jsou k dispozici, je specifické pro konkrétní aplikaci. Rekonfigurace samotného FPGA se tedy děje jen jednou při inicializaci a poté už není potřeba. Konfigurační řetězec VRC je uložen v registrech FPGA a rekonfigurace se provádí jejich přepisem. Tím pádem je velmi rychlá. VRC mají ovšem nevýhodu v tom, že jejich vytvoření v FPGA vyžaduje desetinásobný až stonásobný objem logiky v porovnání s přímou rekonfigurací FPGA [14].

Virtuální rekonfigurovatelný obvod s architekturou znázorněnou na obrázku 5.1 byl použit pro evoluční návrh několika různých obrazových filtrů [14]. Obsahuje dvourozměrné pole programovatelných elementů vytvořených pomocí ALU, která je osmibitová, má jeden vstup a dva výstupy. Propojovací síť tvoří multiplexory. Propojitelnost je taková, že lze každý vstup programovatelného elementu připojit buď na výstup elementu v předchozím sloupci, nebo přímo na primární vstup. Architektura tedy částečně připomíná CGP s parametrem L-back = 1.

Tato architektura je zajímavá i volbou funkcí, kterých může programovatelný element nabývat. Jsou to takové funkce, které se hodí při návrhu konkrétních obrazových filtrů. V tomto případě je to filtr pro redukci výstřelového šumu a pro detekci hran.

Celá kompletní architektura tohoto typu kromě rekonfigurovatelné části obsahuje i další jednotky, jako je fitness jednotka (která počítá fitness) a genetická jednotka (která provádí evoluční algoritmus).

Jedna konkrétní implementace popisované architektury byla implementována pomocí FPGA čipu Virtex-II Pro. Vzhledem k tomu, že obvod Virtex-II Pro obsahuje dva procesory PowerPC, bylo možné využít pro část úkolů evolučního algoritmu s výhodou i je. Konkrétně byly využity pro implementaci prohledávacích algoritmů, protože jejich implementace je náročnější, ale výpočetní náročnost menší [14].

Pomocí architektury VRC byly navrženy konkrétní obrazové filtry pro redukci výstřelového šumu a detekci hran. Vzhledem k možnostem flexibilní volby prohledávacího algoritmu byly zkoumány tři prohledávací algoritmy, konkrétně náhodné prohledávání, horolezecký algoritmus a genetický algoritmus. Nejlepší výsledky dával právě genetický algoritmus [14].

5.2 Evoluční návrh obrazových filtrů na GPU

Výkon a možnosti grafických karet se postupně zvyšují. Jsou proto používány i pro jiné než grafické výpočty. Využívají se i pro genetické algoritmy. Evoluční návrh obrazových filtrů pomocí GPU umožňuje použití floating point operací, což je rozdíl oproti FPGA akceleraci, kde se využívaly pouze celočíselné operace [4].

5.3 Fitness funkce

Evolučně navržené obrazové filtry většinou neřeší zadaný problém naprosto dokonale, ale ve výsledném obrázku zůstávají určité chyby (na kvalitě nalezeného řešení záleží, jak jsou výrazné). Nazvěme je šum. Úkolem evolučního algoritmu je minimalizovat průměrnou vzdálenost dosažené výstupní hodnoty pixelu od požadované výstupní hodnoty pixelu (minimalizovat tento šum). Velikost vstupního obrázku je 256×256 pixelů (rozměr označme jako N, tedy $N = 256$), ale vzhledem k filtračnímu jádru 3×3 pixelů se místa, kde nejsou dostupné všechny pixely, ignorují. Jsou to jednopixelové okraje. Výstupní obrázek má tedy velikost 254×254 pixelů. Hodnota fitness se získá odečtením aktuálního rozdílu obrázků od maximální možné odchylky obou obrázků [8]:

$$FitnessValue = 255(N-2)^2 - DIFF, \tag{5.1}$$

kde

$$DIFF = \sum_{i=1}^{N-2} \sum_{j=1}^{N-2} |B(i,j) - C(i,j)|. \tag{5.2}$$

Proměnná B je vstupní obrázek přefiltrovaný kandidátním filtrem. Proměnná C značí ideální výstupní obrázek, kterého se snažíme pomocí navrhovaného filtru dosáhnout.

5.4 Banka filtrů

Pokud jsme evolučním návrhem získali několik řešení daného problému (například filtry pro odstranění výstřelového šumu), můžeme je využít jako banku filtrů. Při filtraci se využije k vybraných filtrů. Z dosažených mezivýsledků se deterministicky vybere jeden konečný výsledek (například výpočtem mediánu z k hodnot). Před aplikací těchto filtrů je možné obrázek ještě předzpracovat [11].

5.5 Rozšířené filtrační jádro

Pro některé typy vad v obrazu (například šum nazvaný impulse burst noise) je potřeba pro kvalitní výsledky použít větší filtrační jádro (například velikosti 5×5 pixelů). Takto velké filtrační jádro dává celkem 25 vstupů, kvůli čemuž vzniká příliš velký prohledávací prostor pro evoluci pomocí CGP. Proto byl vytvořen ještě selekční operátor, který z 25 pixelů vybere 9 vstupů, které jsou připojeny na vstupy filtru navrženého pomocí CGP. Selekční operátor je rovněž navržen evolučně [11].

5.6 Víceprůchodový přístup

Doposud popsané evolučně navržené filtry prováděly výpočet jedním průchodem. Pro některé typy problémů lze použít jednodušší evolučně navržený filtr, který se však pro cílový výsledek musí aplikovat v předem stanoveném počtu průchodů. Mezi takto navržené filtry patří dilatace, eroze, emboss, různé způsoby detekce hran a motion blur [11].

Kapitola 6

Návrh polymorfních obrazových filtrů

Úkolem této práce je navrhnout a implementovat algoritmus pro evoluční návrh polymorfních obrazových filtrů.

6.1 Metoda

Metodou evolučního návrhu je kartézské genetické programování popsané v kapitole 3. Výpočetní elementy jsou v tomto případě polymorfní, mají dva režimy, čímž je umožněno, aby navržený filtr prováděl dvě různé operace. Vzniklý polymorfní obrazový filtr tedy provádí jednu ze dvou různých operací v závislosti na polymorfním režimu jeho výpočetních elementů. Propojení výpočetních elementů a připojení na výstup je shodné pro oba tyto filtry virtuálně obsažené v jednom polymorfním filtru. Z toho vyplývá, že topologie výrazového stromu je u obou filtračních funkcí obsažených v polymorfním filtru identická a liší se pouze některé výpočetní funkce.

Obvod může obsahovat jak funkce polymorfní, tak funkce nepolymorfní. Evoluční metoda byla navržena tak, aby bylo možné volbou parametrů evoluce ovlivňovat konečnou skladbu použitých funkcí. Je tedy např. možné nastavit strop pro počet polymorfních funkcí v navrženém filtru, popřípadě některé funkce, ať už polymorfní, nebo standardní, úplně vynechat.

Při návrhu algoritmu bylo nutné určit parametry volitelné uživatelem, ale neměnné po celý jeden evoluční běh. Od těchto parametrů bylo nutné odlišit parametry, které podléhají evoluci a které tedy tvoří genotyp.

6.2 Vstupní parametry evoluční metody

V této sekci jsou blíže popsány vstupní parametry této formy kartézského genetického programování, které nejsou součástí genotypu.

Velikost populace – Hodnota p představuje počet jedinců v populaci a je v ní započítán i rodič, který je zkopírován z předešlé populace beze změny. Typická hodnota tohoto parametru je 5.

Počet generací – Hodnota g představuje počet evolučních cyklů algoritmu, tedy generování potomků z nejlepšího rodiče pro následující generaci a jejich následné ohodnocování. Pohybuje se v řádech desetitisíců.

Počet mutací – Hodnota m představuje počet mutovaných genů.

Šířka × výška pole CGP – Rozměry pole $w \times h$ rekonfigurovatelných a propojitelných elementů.

L-back – Parametr L, bližší popis v kapitole 3.

První množina polymorfních funkcí – Tuto množinu si označme R. Je potřeba volby podmnožiny z množiny všech možných polymorfních funkcí vzniklých kombinacemi elementárních funkcí. Tato množina je realizována jako seznam polymorfních funkcí, kde se žádná polymorfní funkce neopakuje. Jedna polymorfní funkce je vždy dvojice dvou klasických funkcí. Každá je určená pro jeden ze dvou režimů. Nepolymorfní funkce má obě tyto funkce shodné, je to tedy speciální případ polymorfní funkce.

Druhá množina polymorfních funkcí – Tuto množinu si označme S. Je možné mít požadavek na horní hranici počtu funkcí použitých v obrazovém filtru. V takovém případě přichází ke slovu druhý seznam polymorfních funkcí, ze kterého se evolučně vybere podmnožina funkcí.

Množiny R i S lze při návrhu vhodně kombinovat. Konkrétní použití je například takové, že první množina bude obsahovat pouze nepolymorfní funkce, zatímco tato množina bude obsahovat pouze funkce polymorfní, jejichž počet ve výsledném obvodu může být vhodné omezit.

Počet prvků podmnožiny druhé množiny polymorfních funkcí – Parametr s určuje maximální počet funkcí, které je možné evolučně přiřadit do obvodu. Je zřejmé, že nemá smysl, aby tato hodnota byla větší než počet prvků druhé množiny polymorfních funkcí.

Implementace podmnožiny byla zjednodušena pro použití v evoluční metodě na seznam, takže se od skutečné množiny liší v tom, že může jeden prvek obsahovat vícekrát.

Trénovací data – Trénovacími daty jsou v případě obrazových filtrů trénovací obrázky. Sada trénovacích dat je představována pro standardní obrazový filtr dvěma obrázky – vstupním a ideálním výstupním. Protože je evolučně navrhován polymorfní filtr, jsou potřeba dvě tyto dvojice, tedy celkem čtyři obrázky. Vzhledem k tomu, že chceme, aby se při výpočtu fitness oba obrázky projevily stejnou měrou, musí mít všechny obrázky stejné rozměry. Jako rozumný kompromis mezi kvalitou a výkonem se ukázal rozměr 256×256 pixelů, ovšem algoritmus musí umět pracovat s libovolným rozměrem, a to například z důvodu výkonnostních testů.

6.3 Elementární výpočetní funkce

Každý výpočetní element realizuje nějakou elementární výpočetní funkci a všechny dohromady tvoří základ obrazového filtru. Vzhledem k tomu, že obrazové filtry jsou tvořeny evolučním návrhem, určuje základní sada elementárních funkcí zásadním způsobem výpočetní

název	funkce	význam
`const`	`c`	konstanta
`ident`	`x`	identita
`or`	`x | y`	bitový OR
`nor`	`~(x | y)`	inverze bitového OR
`and`	`x & y`	bitový AND
`nand`	`~(x & y)`	negace bitového AND
`xor`	`x ^ y`	bitový XOR
`nxor`	`~(x ^ y)`	negace bitového XOR
`_or`	`(~x) | y`	bitový OR s inverzním prvním operandem
`inv`	`~x`	inverze
`div2`	`x >> 1`	celočíselné dělení dvěma
`div4`	`x >> 2`	celočíselné dělení čtyřmi
`add`	`x + y`	součet
`adds`	`min(((unsigned)x) + y, 255)`	součet se saturací
`mean`	`(x + y) >> 1`	průměr se zaokrouhlením dolů
`max`	`max(x, y)`	maximum
`min`	`min(x, y)`	minimum

Tabulka 6.1: Sada elementárních funkcí

možnosti těchto obrazových filtrů. Elementární výpočetní funkce proto musí být dostatečně univerzální, aby co nejlépe pokryly potřeby řešených problémů. Dalším faktorem výběru elementárních funkcí byla také jejich efektivní řešitelnost jak na CPU, tak na GPU.

V tomto případě není na škodu ani redundance. Nevadí, když sada funkcí obsahuje např. inv, and i nand. Naopak poskytuje evolučnímu algoritmu více možností, jak se dobrat nejlepšího výsledku. Sada elementárních funkcí byla z velké části převzata z [14] a je popsána v tabulce 6.1. Funkce jsou popsány pomocí operátorů a funkcí standardní knihovny jazyka C++. Funkce const má speciální význam, protože konstanta není nastavena na pevnou hodnotu 255 jako v [14], ale její hodnota je uložena v genotypu.

Každá funkce má dva celočíselné osmibitové operandy. Některé jsou v případě nepotřebnosti ignorovány. Výsledek je také osmibitový, přebývající bity jsou ořezány.

6.4 Genotyp

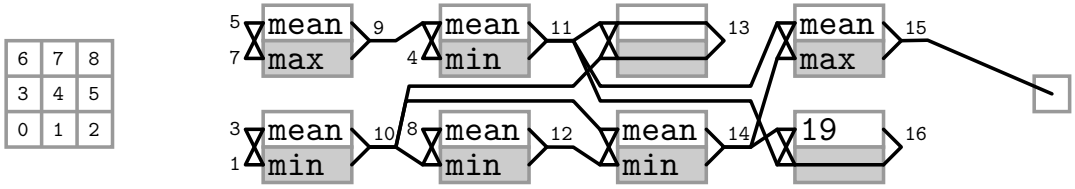
Genotyp obvodu je tvořen jedním chromozomem G, což je vektor celočíselných hodnot. Délka tohoto vektoru i význam a rozsahy jednotlivých jeho prvků vycházejí ze vstupních parametrů evoluční metody popsaných v sekci 6.2.

Prvním prvkem chromozomu je vždy hodnota, kterou vrací funkce const. Rozsah hodnot tohoto prvku chromozomu je $\langle 0; 255 \rangle$.

Podmnožina druhé množiny polymorfních funkcí je řešena seznamem indexů o délce s. Přímo jsou jimi indexovány prvky množiny S. Jak již bylo uvedeno, dochází zde k odchylce od definice podmnožiny, protože indexy některých funkcí se mohou objevit v seznamu vícekrát. To ale v kontextu algoritmu není problém. Rozsah hodnot těchto prvků chromozomu je $\langle 0; |S| \rangle$.

Následuje seznam výpočetních elementů. Elementy jsou indexovány postupně po sloupcích shora dolů zleva doprava. Každý výpočetní element je v chromozomu reprezentován třemi prvky. První dva prvky určují, kam jsou připojeny vstupy výpočetního elementu.

Index	0	1	2	3	4	5
Funkce	const/ident	ident	add	max	mean/min	mean/max



Obrázek 6.1: Příklad kandidátního polymorfního filtru s parametry $w = 4$, $h = 2$, $L = 2$, $R = \{const/ident, ident, add, max\}$, $S = \{mean/max, mean/min, and/or, and\}$, $s = 2$. Uzly 13 a 16 nejsou ve fenotypu využity. Chromozom: 19, 1,0, 5,7,5, 3,1,4, 9,4,4, 8,10,4, 11,10,1, 10,12,4, 11,14,5, 14,11,0, 15. První gen je hodnota konstanty. Další dva geny představují podseznam množiny S. Následují trojice hodnot představující výpočetní elementy. Poslední hodnota určuje zapojení výstupu. Tabulka ukazuje způsob indexace funkcí.

Rozsah hodnot závisí na parametru L a na sloupci, ve kterém se element nachází. Parametr L totiž určuje, o kolik sloupců před aktuálním je nejvzdálenější element, kam lze ještě vstupy připojit.

Rozsah hodnot pro oba vstupy konkrétního elementu je stejný. Jestliže $L > e_x$, pak je rozsah

$$\langle 0; 9 + he_x), \tag{6.1}$$

jinak je rozsah

$$\langle 9 + h(e_x - L); 9 + he_x), \tag{6.2}$$

kde e_x je souřadnice x elementu v CGP poli a ostatní parametry jsou určeny podle sekce 6.2.

Výstupy elementů jsou indexovány od hodnoty 9, tzn. následují těsně po indexech z rozsahu $\langle 0; 8\rangle$, které jsou vyhrazeny pro indexaci filtračního jádra, což je vstupní pixel a jeho osmiokolí. Samotné filtrační jádro je indexováno po řádcích zleva doprava odspoda nahoru. Středový pixel jádra má tedy index 4.

Třetí hodnota je index polymorfní funkce, kterou bude výpočetní element realizovat. Tímto prvkem je indexován myšlený seznam složený z množiny R, za kterým následuje podmožina množiny S. Rozsah hodnot těchto prvků je tedy $\langle 0; |R| + s)$.

Posledním prvkem chromozomu je index výstupu. Výstup lze připojit na výstup kteréhokoli prvku, případně i přímo na některý z devíti vstupů. Hodnota L zde nehraje roli. Rozsah hodnot tohoto prvku je $\langle 0; 9 + wh)$.

Počet prvků chromozomu je dán vztahem

$$|G| = s + 3wh + 2. \tag{6.3}$$

Vztah fenotypu a genotypu ilustruje obrázek 6.1.

6.5 Fitness funkce

Fitness funkce pro polymorfní filtry je odvozena od fitness funkce používané při evolučním návrhu konvenčních obrazových filtrů popsané v sekci 5.3. Hodnota fitness F kandidátního řešení polymorfního filtru se tedy spočítá takto:

$$F = 510(N-2)^2 - (D_1 + D_2), \tag{6.4}$$

kde

$$D_1 = \sum_{i=1}^{N-2} \sum_{j=1}^{N-2} |B_1(i,j) - C_1(i,j)| \tag{6.5}$$

a

$$D_2 = \sum_{i=1}^{N-2} \sum_{j=1}^{N-2} |B_2(i,j) - C_2(i,j)|. \tag{6.6}$$

Proměnná C_1 značí ideální výstupní obrázek, kterého se snažíme dosáhnout polymorfním filtrem v režimu 1. Proměnná C_2 značí ideální výstupní obrázek, kterého se snažíme dosáhnout polymorfním filtrem v režimu 2. Proměnná B_1 je vstupní obrázek přefiltrovaný polymorfním filtrem v režimu 1 a proměnná B_2 je vstupní obrázek přefiltrovaný polymorfním filtrem v režimu 2. Proměnná N je rozměr obrázku. Pomocné proměnné D_1 a D_2 představují celkový rozdíl mezi cílovým a kandidátním řešením v příslušném režimu.

Kapitola 7

Implementace

V této kapitole je popsána implementace speciálního případu algoritmu kartézského genetického programování určeného pro evoluční návrh polymorfních obrazových filtrů, jehož specifikace je popsána v kapitole 6.

Tento algoritmus byl nejprve implementován v jazyce C++. Další snahou byl pokus o akceleraci algoritmu na GPU. Akcelerace pomocí GPU již byla použita podobným způsobem při návrhu konvenčních obrazových filtrů [2, 4]. V případě této práce byla funkcionalita GPU zpřístupněna pomocí knihovny OpenGL verze 2.1 a GLSL shaderů. Akcelerován nebyl celý evoluční algoritmus, ale jen některé jeho části, které byly pro akceleraci vhodné.

Implementace je provedena s důrazem na výkon. Alokace paměťového prostoru a inicializace většiny dat je provedena v přípravné fázi algoritmu. V nejnáročnější fázi, kterou je evoluční návrh a především výpočty fitness hodnot, již k alokacím paměti nedochází a stejně tak je omezeno zbytečné kopírování dat v operační paměti.

Velká část funkcionality je obsažena v konstruktorech. Vzhledem k tomu, že konstruktory nevracejí žádnou hodnotu, musí se chybové stavy předávat pomocí výjimek. V tomto projektu nejsou výjimky nijak zachytávány, každá je fatální a způsobí ukončení programu. Všechny konstruktory ale proběhnou pouze na začátku celého programu, takže nehrozí, že by byl výjimkou přerušen dlouhý výpočet. V konstruktorech jsou takto ošetřována vstupní data, je kontrolována jejich smysluplnost. Pokud je ve vstupních parametrech nebo v datech vstupujících do programu chyba, nemá smysl provádět evoluční návrh.

7.1 Třída `CGPEvo`

Základní třídou implementující evoluční algoritmus a zapouzdřující všechny ostatní potřebné objekty je třída `CGPEvo`.

7.1.1 Datové členy

Všechny datové členy jsou soukromé. Dělí se na dvě skupiny. První z nich jsou parametry běhu algoritmu. Jsou inicializovány v konstruktoru, uvolněny v destruktoru a během samotné evoluce je z nich pouze čteno. Následuje jejich výčet a stručný popis. Podrobná specifikace většiny z nich je v sekci 6.2.

`m_population_size` – počet jedinců v populaci.

`m_generations_count` – počet generací.

`m_fitness_function` – metoda výpočtu fitness hodnoty, nabývá hodnot `FITN_CPU`, resp. `FITN_GPU_GL` a označuje volbu výpočtu na CPU, resp. na GPU pomocí knihovny OpenGL a GLSL shaderů.

`m_mutation_count` – počet mutací při přenosu genotypu z rodiče na potomka.

`m_width`, `m_height` – rozměry CGP pole.

`m_lback` – parametr L-back.

`m_function_pair_list`, `m_function_pair_count` – první seznam polymorfních funkcí. Obsahuje dvojice indexů výpočetních funkcí. Parametr `m_function_pair_count` vynásobený dvěma dá reálnou délku pole `m_function_pair_list`, protože pole ukládá dvojice indexů základních funkcí.

`m_selectable_pair_list`, `m_selectable_pair_count` – druhý seznam polymorfních funkcí. Struktura dat je shodná s prvním seznamem.

`m_max_selected_count` – Délka podseznamu druhého seznamu.

`m_img_orig0`, `m_img_target0` – Trénovací dvojice zdrojového a cílového obrázku pro první polymorfní mód obvodu.

`m_img_orig1`, `m_img_target1` – Trénovací dvojice zdrojového a cílového obrázku pro druhý polymorfní mód obvodu.

Druhou skupinu datových členů třídy tvoří proměnné odrážející aktuální stav evoluce. Patří sem následující členy:

`m_generation_index` – index aktuální generace.

`m_best_index` – index nejlepšího jedince v aktuální generaci.

`m_best_fitness` – fitness hodnota nejlepšího jedince v aktuální generaci.

`m_generation` – vektor jedinců, který obsahuje dvě poslední generace. Funguje jako dvojitý buffer. To, ve které polovině bufferu je uložena aktuální generace, závisí na tom, jestli je index aktuální generace sudý nebo lichý. Datový typ tohoto členu je `std::vector<CGPInd>`, přičemž je alokován v konstruktoru na velikost `m_population_size*2` a používá se jako neměnné úložiště pro běh evolučního algoritmu. Za běhu evolučního algoritmu k realokacím nedochází.

Ukazatel `m_evogl` na objekt typu `CGPEvoGL` je někde na pomezí těchto dvou skupin. Obsahuje pomocná data pro GPU akceleraci, detailní popis v sekci 7.4.1. Objekt je alokován a inicializován pouze v případě, že člen `m_fitness_function` je nastaven na `FITN_GPU_GL`. V opačném případě je ukazatel nastaven na `NULL`.

7.1.2 Konstruktory, destruktor, operátory a členské funkce

Třída implementuje jen operace, které jsou reálně využity a které tedy mohou být rozumně otestovány. Z toho důvodu není implementován implicitní konstruktor, ale pouze jeden explicitní konstruktor. Mezi parametry konstruktoru jsou jednak základní parametry CGP (sekce 6.2), a dále také volba metody mezi standardním výpočtem na CPU a GPU akcelerací. Poslední faktický parametr složený z pole `geno_init` a jeho délky `geno_init_size`,

který obsahuje počáteční genotyp, je volitelný. Pokud ho nechceme uvést, předáme jako délku `geno_init_size` hodnotu -1. S tímto parametrem je naloženo příslušným způsobem až ve třídě `CGPInd`. V případě uvedení počátečního genotypu se tento použije pro inicializaci počátečních genotypů jedinců v populaci místo jejich náhodného vygenerování.

Činnost konstruktoru spočívá v načtení trénovacích obrázků a ve zkopírování dalších parametrů včetně seznamů funkcí. Dále konstruktor inicializuje dvojitý buffer jedinců `m_generation`. Pokud je zvolena GPU akcelerace, je inicializována pomocná třída `CGPEvoGL`. Pomocnou funkcí `internalFindBest` jsou inicializovány členy `m_best_index` a `m_best_fitness`.

Implicitní konstruktor, kopírovací konstruktor a operátor přiřazení jsou ve třídě deklarovány, ale nejsou implementovány. Deklarace je důležitá, aby v případě jejich použití neznalým uživatelem nebyly automaticky vygenerovány implicitní verze. Docházelo by totiž k obtížně laditelným chybám. Místo toho se v takovém případě vyvolá chyba sestavování (link error) a program se nepřeloží.

Jediná soukromá funkce – `internalFindBest` – provádí aktualizaci hodnot členů `m_best_index` a `m_best_fitness` s využitím funkce `GCPInd::getFitness`.

Veřejné členské funkce `getGenerationIndex`, `getBestFitness`, `getBestMEPP`, `getBestIndex` a `getInd` jsou funkce pro přístup k datům třídy. Za zmínku stojí funkce `getBestMDPP`, která vypočítá průměrnou absolutní odchylku mezi pixely trénovacích cílových a reálně přefiltrovaných obrázků z jejich rozměrů a z hodnoty fitness. Dále je zajímavá funkce `getInd`, která odstiňuje dvojitý buffer od uživatele a vrací vždy ukazatel na jedince aktuální generace podle zadaného indexu.

Veřejná členská funkce `nextGeneration` představuje jeden iterační krok evolučního algoritmu. Pokud nebylo dosaženo zadaného počtu generací, tato funkce vytvoří další generaci jedinců z aktuální, inkrementuje číslo generace a vrátí hodnotu `true`. Další generace je vytvořena tak, že její první jedinec je inicializován kopií nejlepšího jedince aktuální generace a další jedinci mutovanou kopií. Pokud bylo dosaženo maximálního počtu generací, další generace se netvoří a funkce ihned vrací `false`.

7.2 Třída `CGPInd`

Třída `CGPInd` představuje jednoho jedince v populaci. Je určena pro práci v součinnosti s třídou `CGPEvo`, kde se nachází vektor objektů této třídy.

7.2.1 Datové členy

Všechny datové členy třídy `CGPInd` jsou soukromé. Členy `m_fitness_function` až `m_max_selected_count` jsou shodné se členy ve třídě `CGPEvo` (sekce 7.1.1). Jsou to jejich prosté kopie, které jsou inicializovány v konstruktoru a nemění se po celou dobu života objektů.

Členy `m_img_orig0` a `m_img_target0` jsou ukazatele na trénovací obrázky prvního módu obvodu. Ukazují na obrázky ve třídě `CGPEvo`. Pokud by měl každý jedinec vlastní kopie obou trénovacích obrázků, docházelo by ke zbytečnému plýtvání pamětí, což by se mohlo negativně odrazit i na výkonu. Členy `m_img_orig1` a `m_img_target1` jsou ukazatele na trénovací obrázky druhého módu obvodu. Důvod pro volbu ukazatelů místo kopií je shodný se členy `m_img_orig0` a `m_img_target0`.

Další datové členy jsou následující:

`m_fitness` – hodnota fitness daného jedince.

`m_genes` – pole genů – genotyp.

Každému jedinci jsou ještě přiřazeny pomocné datové členy představující předalokovaný paměťový prostor nutný pro výpočty. Pro výpočty na CPU jsou určeny tyto členy:

`m_fitn_cpu_aux` – datový prostor pro výpočty na CPU. Bližší popis v sekci 7.3.1.

`m_img`, `m_img_width` a `m_img_height` – pomocný obrazový buffer a jeho rozměry.

Pro akcelerované výpočty na GPU každý jedinec obsahuje ukazatel na objekt třídy `CGPEvoGL`, který byl v případě potřeby vytvořen v konstruktoru objektu `CGPEvo`. Všichni jedinci tedy obsahují tento jeden shodný ukazatel. Každý jedinec do něj může určitým řízeným způsobem zasahovat, konkrétně při akcelerovaném výpočtu fitness hodnoty. Fitness hodnotu při akceleraci nemůže počítat více jedinců najednou, ale vždy jen jeden.

7.2.2 Konstruktory, destruktor, operátory a členské funkce

Třída neimplementuje implicitní konstruktor. Ten je pouze deklarován. Implementován je jeden explicitní konstruktor. Je volán pouze z konstruktoru třídy `CGPEvo`, předává parametry CGP a vytváří jejich kopii. Výjimkou jsou trénovací obrázky, kde předává a kopíruje pouze jejich ukazatele. Dále předává a kopíruje genotyp, pokud je uveden. Jinak tento genotyp náhodně vygeneruje. Nakonec vypočítá fitness hodnotu funkcí `computeFitness` a uloží ji do `m_fitness`.

Kopírovací konstruktor je deklarován i implementován. Je důležitý při vytváření vektoru jedinců `m_generation` v konstruktoru třídy `CGPEvo`.

Dále je kvůli způsobu implementace kontejneru `std::vector` ve standardní knihovně nutné implementovat i operátor přiřazení. Tento operátor se však nikdy nevolá. Proto je jeho implementace omezena na pouhé vyvolání výjimky, kterým se předchází obtížně laditelným chybám v případě, že by byla třída použita jinak, než bylo zamýšleno.

Destruktor pracuje inverzně ke konstruktorům – uvolňuje alokovanou paměť. Objekty, které se nekopírovaly, což jsou trénovací obrázky a pomocná třída `CGPEvoGL`, ponechává bez povšimnutí.

Další část tvoří veřejné členské funkce. Jsou využívány jednak třídou `CGPEvo`, ale mohou být využity i mimo ni, protože k jedinci se lze dostat přístupovou funkcí `CGPEvo::getInd`.

`from` – tato funkce je příbuzná kopírovacímu konstruktoru. Kopíruje jedince, ale na rozdíl od kopírovacího konstruktoru nerealokuje paměť. Místo toho využívá paměť, která již byla alokována pro jiného jedince. Je to ale typicky vhodné pouze u jedinců stejných parametrů, protože jen u nich je zřejmé, že genotyp má stejný počet genů a že nedojde k zápisu mimo alokovanou paměť. Tato funkce je určena a využívá se pouze pro kopírování dat ve dvojitém bufferu datového členu `CGPEvo::m_generation`, tzn. vytváří potomka z rodiče. Proto se nekopírují neměnné parametry CGP, které jsou pro všechny členy stejné a byly nastaveny již explicitním, popř. kopírovacím konstruktorem. Kopíruje se jen genotyp. Funkce umožňuje navíc i mutaci takže je `m_mutation_count`-krát přepsán náhodný gen novou hodnotou. Následně je přepočítána, resp. zkopírována hodnota fitness při mutovaném, resp. prostém kopírování. Případný výpočet nové hodnoty fitness provádí funkce `computeFitness`.

`print` – tato funkce vypíše do zadaného souboru (typicky na standardní výstup) textovou reprezentaci CGP pole představujícího obrazový filtr.

`writeGeno` – tato funkce zapíše genotyp do zadaného souboru (typicky na pevný disk). Výsledný textový soubor je určen k použití dohromady s konfigurací CGP, protože sám o sobě nedává dostatečnou informaci pro rekonstrukci obrazového filtru.

`getFitness` – tato funkce vrací hodnotu soukromého členu `m_fitness`.

`processImage` – toto je základní funkce pro filtraci obrázku na CPU. Bližší popis v sekci 7.3.1.

`processTex` – toto je základní funkce pro filtraci obrázku na GPU. Obrázek je zadán ve formě textury a musí mít rozměry shodné se obrázky, kterými byl inicializován objekt typu `CGPEvoGL`.

`processSaveImage`, `processSaveTex` – pomocné funkce, které interně volají jednu z funkcí `processImage` a `processTex`.

Soukromé členské funkce jsou tyto:

`getGenesCount` – tato funkce vrací počet genů v genotypu. Výpočet určuje vzorec 6.3.

`getGeneRange` – získá rozsah platných hodnot pro daný gen. Specifikace rozsahů hodnot genů v genotypu je definována v sekci 6.4.

`replaceGene` – v genotypu nahradí gen zadaného indexu náhodnou hodnotou z povoleného rozsahu.

`computeD` – výpočet části fitness hodnoty na CPU. Bližší popis v sekci 7.3.

`computeD_GL` – výpočet části fitness hodnoty na GPU. Bližší popis v sekci 7.4.2.

`computeFitness` – výpočet fitness hodnoty zvolenou metodou, tzn. buď pomocí CPU, nebo akcelerovaně na GPU.

7.3 Výpočet fitness hodnoty na CPU

7.3.1 Zpracování obrázku

Základní funkcí pro výpočet fitness hodnoty je funkce `CGPInd::processImage`, která zpracuje vstupní obrázek na výstupní. Polymorfní obrazový filtr má dva režimy, tato funkce tedy umožňuje parametrem zvolit jeden z těchto režimů.

Jako pomocný výpočetní prostor je již v konstruktoru třídy `CGPInd` alokováno pole `CGPInd::m_fitn_cpu_aux`. Velikost tohoto prostoru je dána vztahem $c_i + wh + c_o$, kde $c_i = 9$ a představuje počet vstupů, w a h jsou rozměry CGP pole a $c_o = 1$ a představuje počet výstupů. Po dobu života jedince ani jedinců shodných parametrů se jeho velikost nemění, a proto toto pole není během evoluce realokováno.

Funkce pracuje tak, že postupně pro každý pixel výstupního obrázku (kromě okrajových) vypočte jeho hodnotu. To provede tak, že nejprve inicializuje prvních devět prvků pole `CGPInd::m_fitn_cpu_aux` odpovídajícím bodem vstupního obrázku a jeho devítiokolím. Pak postupně po sloupcích prochází výpočetní elementy. Odkaz na polymorfní výpočetní funkci a oba vstupy každého z elementů získává přímo z genotypu. Z polymorfní výpočetní funkce, což je v praxi dvojice dvou standardních výpočetních funkcí, na základě zvoleného módu zjistí standardní výpočetní funkci, kterou má použít. Indexy vstupů přímo indexuje

pole `CGPInd::m_fitn_cpu_aux`. Výsledek zapíše zpět do pole `CGPInd::m_fitn_cpu_aux` na pozici odpovídající výpočetnímu elementu, přičemž tyto jsou číslovány od indexu 9 po sloupcích. Nakonec do pixelu cílového obrázku zkopíruje výsledek, který je v poli `CGPInd::m_fitn_cpu_aux` na pozici, ke které je připojen výstup. Funkce obsahuje jednoduchou optimalizaci, kdy nechává počítat výpočetní elementy pouze po element připojený na výstup, protože elementy za ním již konečný výsledek zcela jistě neovlivní.

7.3.2 Výpočet fitness

Jakmile jsou zpracovány oba vstupní obrázky každý v příslušném polymorfním režimu obrazového filtru, je možné přistoupit k výpočtu rozdílů od ideálních cílových obrázků. Tato operace je i se samotným zpracováním obrázku obsažena ve funkci `CGPInd::computeD`. Tuto funkci obaluje funkce `CGPInd::computeFitness`, která spouští vybranou metodu výpočtu, tedy buď výpočet na CPU, nebo na GPU. Zde předpokládáme, že je vybrán výpočet na CPU. Ze dvou hodnot D_1 a D_2 pro každý z polymorfních režimů funkce vypočte hodnotu fitness podle vztahu 6.4.

7.4 Třída `CGPEvoGL` a výpočet fitness hodnoty na GPU

7.4.1 Třída `CGPEvoGL` a její datové členy

Třída `CGPEvoGL` uchovává data potřebná pro výpočty na GPU. V této třídě je implementován pouze konstruktor a destruktor. Konstruktor třídy `CGPEvoGL` je volán uvnitř konstruktoru třídy `CGPEvo` pouze v případě, že je zvolena akcelerovaná výpočetní metoda.

Všechny datové členy třídy `CGPEvoGL` jsou soukromé. Třída `CGPInd` musí mít k jejím datovým členům přístup, a proto je tato třída uvnitř třídy `CGPEvoGL` označena jako přátelská.

`m_diff_buffer` – buffer pro zápis vypočteného rozdílu dvou textur.

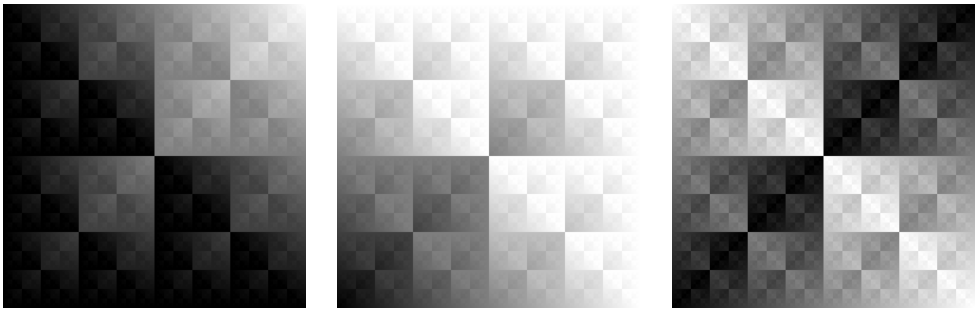
`m_tex` – čtveřice trénovacích obrázků uložených v paměti grafické karty jako textury. Patří sem vstupní obrázek a jeho ideální výstupní protějšek pro první polymorfní režim obvodu a dále vstupní obrázek a jeho ideální výstupní protějšek pro druhý polymorfní režim obvodu.

`m_blacktex` – černá textura pro filtrace obrázků na GPU. Fragment shader počítá absolutní hodnoty rozdílů mezi dvěma texturami, takže pokud chceme vypočíst prostý přefiltrovaný obrázek, musíme jako druhou texturu dát texturu se samými nulami, tedy černou.

`m_img_width`, `m_img_height` – společné rozměry čtveřice textur uložených v `m_tex`.

`m_lookup` – textura o rozměrech 256×256 pixelů, která slouží jako vyhledávací tabulka pro realizaci výpočtu funkcí and, or a xor ve fragment shaderu. Obsahuje tři nezávislé tabulky uložené každou v jednom ze tří barevných kanálů – červeném, modrém a zeleném (obrázek 7.1).

`m_frag_shader_lines`, `m_frag_shader_line_length` a `m_frag_shader_lines_count` – tyto datové členy představují buffer pro uložení zdrojového textu aktuálního fragment shaderu. Bližší popis shaderu je v sekci 7.4.2.



(a) Červený kanál (b) Zelený kanál (c) Modrý kanál

Obrázek 7.1: Vyhledávací textura pro funkce and, or a xor, které jsou po řadě reprezentovány barevnými kanály (a), (b) a (c).

`m_width`, `m_height` – rozměry CGP pole.

`m_vert_shader` – zkompilovaný vertex shader.

`m_framebuffer`, `m_renderbuffer` – další objekty knihovny OpenGL pro offscreen rendering.

`m_linei_const`, `m_linei_a09` a `m_linei_r` – indexy význačných řádků fragment shaderu, bližší popis je v sekci 7.4.2.

7.4.2 Výpočet fitness hodnoty na GPU

Pro implementaci akcelerovaného výpočtu fitness hodnoty byla použita knihovna OpenGL verze 2.1. Kromě základní funkcionality poskytované knihovnou OpenGL uvedené verze bylo využito rozšíření `GL_EXT_framebuffer_object`.

Inicializace

Akceleraci výpočtu na GPU předchází inicializace potřebných dat a alokace objektů uložených v paměti grafické karty a v RAM. Tato počáteční inicializace probíhá v konstruktoru třídy `CGPEvoGL`.

Nejprve jsou načteny trénovací obrázky a uloženy do paměti grafické karty jako textury. Tyto textury mají nastaven interpolační filtr pro zvětšení i zmenšení na `GL_NEAREST`, aby se omezil vliv případné nepřesnosti texturových souřadnic.

Jsou vygenerovány další dvě textury – černá textura o rozměrech 16×16 pixelů (zde nejsou rozměry příliš podstatné) a vyhledávací textura pro funkce and, or a xor o rozměrech 256×256 pixelů. Interpolační filtr je nastaven stejným způsobem jako u trénovacích textur.

Dále je částečně předpřipraven zdrojový text fragment shaderu pro urychlení jeho konstrukce, neboť fragment shader je nejkomplikovanější částí GPU akcelerace. Příklady fragment shaderů jsou v příloze D. Zdrojový text fragment shaderu se musí konstruovat a kompilovat při výpočtu fitness hodnoty každého kandidátního řešení dvakrát pro každý ze dvou režimů polymorfního filtru. Při kompilaci se s výhodou využívá vlastnosti funkce `glShaderSource`, která neomezuje datovou strukturu zdrojového textu na jeden znakový buffer, ale umožňuje zdrojový text zadat jako pole znakových bufferů. Díky tomu lze měnit části o proměnném počtu znaků uprostřed zdrojového textu shaderu bez zbytečných přesunů dat v paměti. Fragment shader v této fázi není kompilován.

Zároveň jsou uloženy indexy význačných řádků zdrojového kódu do datových členů `m_linei_const`, `m_linei_a09` a `m_linei_r`. Ty jsou později využívány k rychlým změnám zdrojového kódu.

Následuje kompilace vertex shaderu, který je velmi jednoduchý. Jeho zdrojový kód je v příloze D.

Jsou inicializovány objekty knihovny OpenGL určené pro offscreen rendering. Nejprve je funkcí `glGenFramebuffersEXT` vygenerován framebuffer. Následně je nutné vytvořit prostor pro vykreslování. Tento prostor je reprezentován renderbufferem, který je vygenerován funkcí `glBindRenderbufferEXT`. Renderbufferu je pomocí funkce `glRenderbufferStorageEXT` nastaven prostor pro vykreslování, tzn. rozměry a barevná hloubka [1]. Rozměry jsou nastaveny na rozměry trénovacích obrázků. Barevná hloubka je nastavena na `GL_RGBA`. Následně je svázán framebuffer s renderbufferem pomocí funkce `glFramebufferRenderbufferEXT`. Framebufferu je přiřazen pouze barvový buffer ve formě renderbufferu. Hloubkový ani jiný buffer se nevyužívá.

Nakonec je alokována paměť pro uložení kopie výsledku shaderu (datový člen `m_diff_buffer`).

Výpočet

Samotnou filtraci obrázku ve formě textury provádí funkce `CGPInd::processTex`. Nejprve je doplněn zdrojový kód fragment shaderu, který obsahuje vlastní výpočty. Fragment shader je zkompilován, slinkován s vertex shaderem, čímž vznikne shader program. Shader programu jsou nastaveny proměnné skupiny `uniform` na příslušné hodnoty texturových jednotek a rozměrů textur.

Výpočet probíhá tak, že jsou přiřazeny textury příslušným texturovým jednotkám a je vykreslen jeden čtverec po celé ploše framebufferu. Funkcí `glutSwapBuffers` se vykreslení dokončí a následně jsou přečteny výsledky funkcí `glReadPixels`.

Poté jsou odstraněny objekty OpenGL alokované ve funkci `CGPInd::processTex` a tato funkce končí.

Funkce `CGPInd::computeD_GL`, která volá funkci `CGPInd::processTex`, získané výsledky sečte. Výsledná suma je přiřazena proměnné D_1, popř. D_2 v závislosti na polymorfním režimu obrazového filtru. Funkce `CGPInd::computeFitness`, která nyní pracuje v režimu GPU akcelerace, vypočte s pomocí D_1 a D_2 fitness hodnotu F.

Fragment shader

Pro úplnou představu o GPU výpočtu zbývá popsat fragment shader. Kód fragment shaderu představuje postup zpracování jednoho pixelu výstupního obrazu. O volání fragment shaderu pro každý pixel obrazu se stará rasterizér, který získá informace potřebné pro rasterizaci aktuálního primitiva součinností knihovny OpenGL a vertex shaderu. V tomto konkrétním případě je primitivem obecně obdélník, i když typicky jsou vstupní obrázky programu čtvercové.

Fragment shader je generován pro každé kandidátní řešení znovu ve funkci `CGPInd::processTex`. Pro každý z obou režimů polymorfního filtru je vygenerován příslušný fragment shader. Ukázkové zdrojové texty fragment shaderů byly vygenerovány pro každý ze dvou režimů polymorfního filtru dilatace a eroze, jehož schéma je na obrázku 8.3. Jsou uvedeny v příloze D.

Na prvním řádku fragment shaderu je uvedena verze jazyka GLSL – 1.20. Proměnné `tex0`, `tex1` a `tex2` jsou po řadě vstupní textura, ideální výstupní textura a textura před-

stavující vyhledávací tabulku pro funkce and, or a xor. Proměnná `dims` obsahuje rozměry prvních dvou textur.

Dále jsou obsaženy definice všech elementárních funkcí převedené do GLSL. Jejich implementace není tak triviální jako u výpočtů na CPU. Názvy začínají znakem `f`. Jsou pouze dvoupísmenné, aby se snížila náročnost překladu zdrojového kódu shaderu. Každá funkce má dva parametry typu `float` a vrací výstup typu `float`. Všechny jsou z rozsahu $\langle 0{,}0; 255{,}0\rangle$ a měly by být zaokrouhleny na celá čísla. Tímto způsobem je možné funkce snadno použít pro výpočet výsledků výpočetních elementů CGP.

Funkce `const`, `ident`, `inv`, `div2`, `div4`, `add`, `adds`, `mean`, `max` a `min` jsou implementovány přímo za použití prostředků jazyka GLSL. U některých je využita funkce `floor`, aby se eliminovaly zaokrouhlovací chyby.

Jazyk GLSL z principu poskytuje spíše operace v plovoucí řádové čárce, které se dají využít pro výpočty v pevné řádové čárce. Naopak není obsažena podpora pro bitové operace. Funkce `or`, `nor`, `and`, `nand`, `xor`, `nxor` a `_or` jsou proto implementovány pomocí vyhledávací tabulky – textury `tex2`, jejíž hodnoty byly předpočítány na CPU.

Následuje samotné tělo shaderu. Z vestavěné proměnné `gl_FragCoord` jsou zkopírovány souřadnice pixelu. Pomocí nich jsou indexovány souřadnice pixelů filtračního jádra ze vstupní textury. Definice vstupů a výstupů výpočetních elementů jsou řešeny jako deklarace proměnných typu `float`. Jsou indexovány stejným způsobem jako prvky pole `CGPInd::m_fitn_cpu_aux` (sekce 7.3.1), hodnota indexu je ve zdrojovém kódu shaderu reprezentována v šestnáctkové soustavě a je přímo součástí názvu proměnné, který začíná znakem `a`.

Prvních 9 proměnných (`a00` až `a08`) je inicializováno hodnotami pixelů filtračního jádra ze vstupní textury. Tyto hodnoty jsou původně v rozsahu $\langle 0{,}0; 1{,}0\rangle$, a tak je potřeba je vynásobit hodnotou 255,0. Dalším proměnným jsou již přiřazeny funkce s příslušnými parametry, kterými jsou některé z předešle definovaných proměnných s počátečním znakem `a`. Tato přiřazení přesně odpovídají výpočetním elementům kandidátního řešení CGP pole. Proměnná `n` nemá žádnou úlohu a pouze zjednodušuje generování zdrojového kódu.

Proměnné `r` je přiřazena výstupní proměnná. Na závěr fragment shaderu je vypočítána absolutní hodnota rozdílu mezi výstupem filtru (proměnnou `r`) a odpovídajícím pixelem ideální výstupní textury, přemapována z rozsahu $\langle 0{,}0; 255{,}0\rangle$ na rozsah $\langle 0{,}0; 1{,}0\rangle$ a přiřazena vestavěné proměnné `gl_FragColor.r`, čímž je vykreslena do offscreen bufferu.

Kapitola 8

Experimentální ověření

Tato kapitola popisuje praktické výsledky dosažené při testování vytvořené implementace evolučního algoritmu. Pro každý test byla vybrána dvojice standardních obrazových filtrů a následovaly experimenty s cílem navrhnout co nejlepší polymorfní filtr, který by jen na základě záměny hradel, ale při shodné topologii obvodu dokázal co nejpřesněji replikovat každý z filtrů, tedy mít co nejmenší průměrnou odchylku hodnot pixelů od ideálního řešení. Funkčnost navržených polymorfních filtrů byla dále ověřována pomocí testovacího obrázku (obrázek 8.1), který byl podstatně větší než základní trénovací obrázek (obrázek 8.2).

Téměř veškerý návrh probíhal neakcelerovaně, tedy na CPU, protože byl k dispozici výkonný výpočetní server, na kterém mohl být tento návrh prováděn. Výpočetní server měl 2 čtyřjádrové procesory Intel Xeon 2,66 GHz.

8.1 Metodika

Výsledky evolučního návrhu a rychlost konvergence k nejlepšímu řešení závisí do velké míry na zvolených parametrech kartézského genetického programování. Proto byla nejprve provedena základní série experimentů, na základě jejichž výsledků byly zvoleny parametry konfigurací pro navazující experimenty. Pro sérii základních experimentů byla zvolena sada šesti konfigurací s kombinací různých parametrů. V každé konfiguraci bylo použito 20000 generací, 5 jedinců v populaci, jedna mutace na potomka, CGP pole o velikosti 4×4, 5×5 nebo 6×6 a hodnota L-back 2 nebo 3. K dispozici byla dána celá sada funkcí včetně všech možných polymorfních funkcí vytvořitelných kombinací funkcí základních.

S každou konfigurací bylo provedeno 40 experimentů. Následně byly vybrány konfigurace, u kterých bylo dosaženo nejlepších průměrných výsledků a nejlepších absolutních výsledků. Od nich se odvodily další konfigurace s cílem dosáhnout zlepšení vlastností výsledných polymorfních filtrů, ať už zlepšení přesnosti výpočtu nebo snížení složitosti obvodů, zmenšení sady použitých funkcí apod.

Při pozdějším zkoumání byl v případě úvodní série evolučních běhů zvolen počet mutací větší než 1 a spíše větší počet generací na úkor počtu experimentů s konkrétní konfigurací.

8.2 Návrh obrazových filtrů

8.2.1 Dilatace a eroze

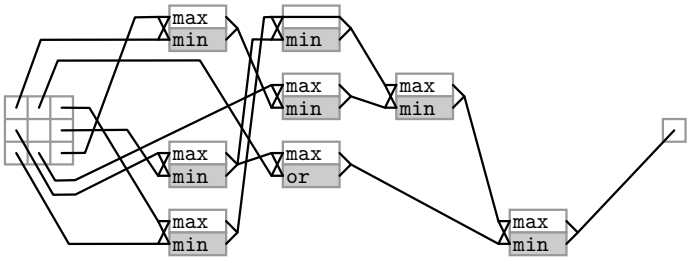
Jako první dvojice obrazových filtrů pro návrh polymorfního obrazového filtru byla zvolena dilatace a eroze. Potřebné trénovací obrázky byly vytvořeny ze základního trénovacího

Obrázek 8.1: Základní testovací obrázek, rozměry 1024×1024 pixelů

Obrázek 8.2: Základní trénovací obrázek, rozměry 256×256 pixelů

Rozměry CGP pole	4×4		5×5		6×6	
L-back	2	3	2	3	2	3
Průměrný výsledek	3,44	**3,37**	4,62	4,59	5,03	4,75
Nejlepší výsledek	**0,25**	0,68	1,18	1,64	1,64	1,60

Tabulka 8.1: Dilatace a eroze – výsledky šesti základních konfigurací CGP.



Obrázek 8.3: Nejlepší nalezený polymorfní filtr pro dilataci a erozi

obrázku pomocí GIMPu.

Výsledky pro úvodní sadu šesti konfigurací CGP jsou uvedeny v tabulce 8.1. Nejlepší hodnoty udávají průměrnou odchylku na pixel nejlepšího filtru. Průměrné hodnoty udávají průměr odchylek na pixel pro výsledné filtry ze všech čtyřiceti pokusů. Je vidět, že nejlepší výsledky (ať už průměrný filtr, nebo nejlepší filtr) vyšly z konfigurace, kdy CGP pole mělo nejmenší rozměry. Bylo by možné experimentovat s ještě menšími rozměry, popřípadě měnit počet mutací. Nalezené řešení (obrázek 8.3) se velmi blíží nejlepšímu známému řešení [7]. Ideální polymorfní filtr realizující tuto funkci by kaskádově vypočetl maximum/minimum ze všech hodnot devítiokolí. Byla by tedy použita jedna polymorfní funkce, a to max/min. Dilatace a eroze fungují velmi podobně, takže překryv topologie je bezproblémový. Prakticky tedy byla ověřena funkčnost evoluční metody.

Podstatně větší výzvu představuje návrh polymorfních filtrů, kde každý z filtrů má funkčnost výrazně odlišnou, tedy filtrů, které nejsou tak blízce příbuzné.

S konfigurací nejlepšího řešení a hodnotou seed[1] nejlepšího pokusu této konfigurace byl proveden akcelerovaný návrh na GPU. Výsledkem návrhu bylo identické řešení, čímž byla úspěšně otestována funkčnost akcelerace evoluční metody.

S tímto modelovým problémem byly dále prováděny srovnávací a výkonnostní testy obou metod výpočtu (sekce 8.3).

8.2.2 Sobelův filtr a odstranění šumu typu sůl a pepř

Dalším evolučně navrhovaným polymorfním filtrem byl Sobelův filtr pro detekci hran a filtr pro odstranění šumu typu sůl a pepř. Trénovací obrázek pro Sobelův filtr byl vytvořen GIMPem a trénovací obrázek poškozený šumem byl vytvořen pomocným jednoúčelovým programem. Šumem bylo poškozeno 10 % originálního obrázku.

Celkové výsledky úvodní série pokusů jsou uvedeny v tabulce 8.2. Z šesti úvodních pokusů měla nejlepší průměr výsledků konfigurace s CGP polem 4×4 a hodnotou L-back

[1] Hodnota seed byla použita pro pseudonáhodný generátor náhodných čísel, který následně generoval shodnou posloupnost náhodných čísel.

Rozměry CGP pole	4×4		5×5		6×6	
L-back	2	3	2	3	2	3
Průměrný výsledek	**8,29**	9,53	9,47	8,63	9,00	9,14
Nejlepší výsledek	6,39	6,49	6,01	5,54	**5,00**	5,28

Tabulka 8.2: Sobelův filtr a odstranění šumu typu sůl a pepř – výsledky šesti základních konfigurací CGP.

Rozměry CGP pole	6×6					
L-back	2					
Max. počet polymorfních funkcí	10			neomezen		
Počet mutací na potomka	1	2	3	1	2	3
Průměrný výsledek	7,13	6,51	**5,71**	8,06	7,13	7,13
Nejlepší výsledek	4,53	4,79	**3,47**	5,30	4,73	3,94

Tabulka 8.3: Sobelův filtr a odstranění šumu typu sůl a pepř – druhá série konfigurací a pokusů.

rovnou 2. Nejlepší filtr byl dosažen konfigurací s CGP polem o rozměrech 6×6 a hodnotou L-back rovnou 2. Také druhý nejlepší filtr byl dosažen v CGP poli o rozměrech 6×6. Tato velikost byla proto zvolena jako základ konfigurace pro další pokusy, přestože časová náročnost evolučního návrhu je u větších CGP polí vyšší.

Následné konfigurace měly tyto parametry: Rozměry CGP pole byly 6×6, počet generací byl dvojnásobný (40000 generací) a počet pokusů s každou konfigurací byl poloviční, tedy 20 pokusů místo 40. Hodnota L-back byla rovna 2 a počet mutací se pohyboval od 1 do 3. Dále byla snaha omezit počet polymorfních funkcí ve výsledném obvodu na maximální počet 10.

Jak je vidět ve shrnující tabulce 8.3, nejlepších výsledků bylo dosaženo při třech mutacích při generování potomka z rodiče. Dále lze pozorovat zajímavý paradox, kdy nejlepší konfigurace byla ta, která měla omezen počet polymorfních funkcí na maximálně 10. I z ostatních výsledků z této sady experimentů vyplývá, že to nebyla náhoda. Tím se podařilo najít řešení, které má jednak nízkou průměrnou chybu na pixel a které zároveň nevyžaduje příliš velký počet polymorfních funkcí. Průměrná chyba nejlepšího řešení byla oproti nejlepšímu řešení vzešlému z úvodních 6 pokusů zmenšena 1,44×, tedy poměrně výrazně.

Dosažená hodnota průměrné chyby na pixel nejlepšího filtru je vztažena k trénovacímu obrázku. Pro velký testovací obrázek mělo nejlepší řešení průměrnou chybu na pixel 3,30, tzn. ještě lepší výsledek než v případě trénovacího obrázku. Na obrázcích 8.4 a 8.5 je vidět porovnání dosažených a ideálních výsledků.

Nejlepší nalezené řešení je na obrázku 8.6. Poslení sloupec CGP pole zůstal nevyužitý, neboť výstup je napojen na výstup výpočetního elementu, který je v předposledním sloupci. Poslední a předposlední řádek by bylo možné sloučit do jednoho. Tento obvod by byl tedy reprezentovatelný i CGP polem o rozměrech 5×5 výpočetních elementů. Ve filtru jsou vidět jisté známky symetrie. Různé dvojice pixelů filtračního jádra jsou připojeny na polymorfní elementy max/min. To se vyskytuje hned 3×. Jinak je ale těžké vidět v navrženém polymorfním filtru nějakou souvislost s jeho funkcí. Podstata jeho fungování je tedy neznámá, což je pro evolučně navržené výpočetní funkce poměrně běžný jev.

(a) Vstup

(b) Výstup filtru v prvním režimu

(c) Výstup Sobelova filtru

(d) Vstup

(e) Výstup filtru v prvním režimu

(f) Výstup Sobelova filtru

Obrázek 8.4: Dva detaily testovacího obrázku a odpovídající výstupy pro první režim polymorfního filtru i pro ideální Sobelův filtr

(a) Poškozený obraz

(b) Výstup filtru ve druhém režimu

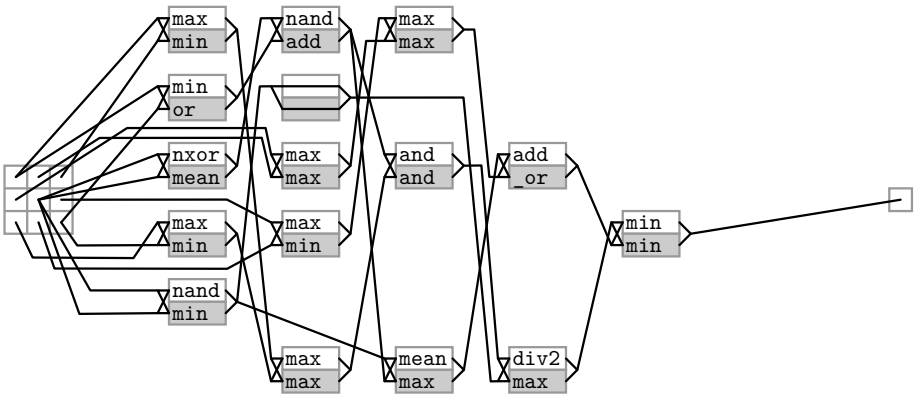
(c) Ideální výstupní obraz

(d) Poškozený obraz

(e) Výstup filtru ve druhém režimu

(f) Ideální výstupní obraz

Obrázek 8.5: Detail testovacího obrázku poškozeného 5% (a) a 10% (d) šumem typu sůl a pepř, zpracovaného polymorfním filtrem ve druhém režimu (b), (e) a ideální výstupy (c), (f)



Obrázek 8.6: Nejlepší nalezené řešení Sobelova filtru a odstranění šumu typu sůl a pepř

Rozměry CGP pole	4×4		5×5		6×6	
L-back	2	3	2	3	2	3
Průměrný výsledek	12,60	10,94	12,67	11,33	13,96	**10,85**
Nejlepší výsledek	8,20	7,68	8,85	7,56	8,85	**7,40**

Tabulka 8.4: Odstranění gaussovského šumu a šumu typu sůl a pepř – výsledky šesti základních konfigurací CGP.

8.2.3 Odstranění gaussovského šumu a šumu typu sůl a pepř

Pro první režim tohoto polymorfního filtru byl použit obrázek poškozený gaussovským šumem, jehož parametr $\sigma = 0{,}1$ pro rozsah hodnot pixelů obrázku přemapovaný na $\langle 0{,}0; 1{,}0\rangle$. Pro druhý režim byl použit stejný trénovací obrázek jako u předchozího filtru. Pro úvodních 6 konfigurací byly zvoleny upravené parametry, protože z výsledků předchozích pokusů jsem usoudil, že by mohly vést k nalezení lepších filtrů. Konkrétně počet mutací na potomka byl 2 místo 1 a počet generací byl 40000 místo 20000. S každou konfigurací se provádělo 20 experimentů místo 40.

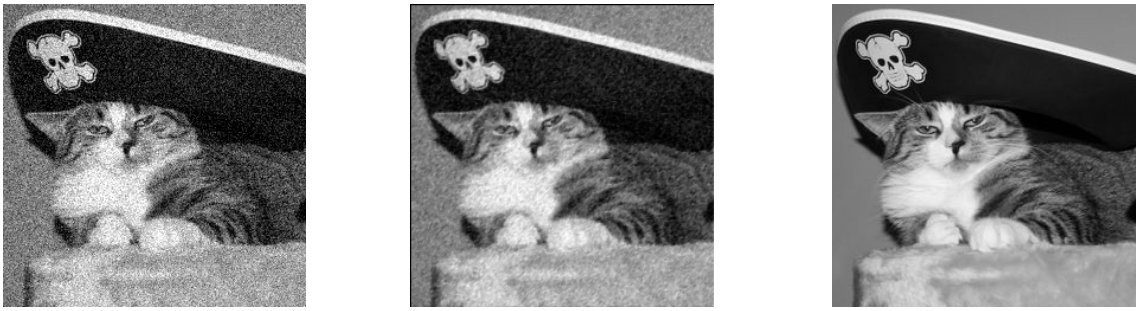
Výsledky pokusů jsou v tabulce 8.4. Nejlepšího řešení i nejlepšího průměrného řešení bylo dosaženo při konfiguraci, kde velikost CGP pole byla 6×6 a hodnota L-back byla 3. Jde o hodnoty, které v rámci těchto šesti konfigurací dávají evoluční metodě největší volnost, takže metoda pravděpodobně narazila na omezení možností evoluce.

Pro další pokusy tedy byla zvyšována hodnota L-back. Aby větší hodnota L-back měla nějaký praktický dopad, bylo použito CGP pole větší do šířky, ale zároveň menší do výšky, aby se zachovala rozumná doba evoluce. Rovněž jsem zkoušel omezit velikost množiny různých polymorfních funkcí na 8. S každou konfigurací bylo provedeno 40 experimentů. Výsledky následných pokusů jsou vidět v tabulce 8.5.

Nejlepšího řešení bylo dosaženo při rozměru CGP pole 7×4 a opět se projevil paradox, kdy omezení počtu různých polymorfních funkcí ve filtru vedlo k lepšímu výsledku než při neomezeném využití těchto funkcí. Na obrázku 8.7 je vidět porovnání dosažených a ideálních výsledků. Průměrná chyba při zpracování testovacího obrázku byla 6,32, tedy přibližně stejná jako v případě trénovacího obrázku. Schéma nejlepšího řešení polymorfního filtru je na obrázku 8.8.

Rozměry CGP pole	7×4		8×5	
L-back	4		5	
Max. počet polymorfních funkcí	8	neomezen	8	neomezen
Počet mutací na potomka	2	2	2	2
Průměrný výsledek	**8,01**	12,14	8,72	13,35
Nejlepší výsledek	**6,40**	7,32	6,63	7,39

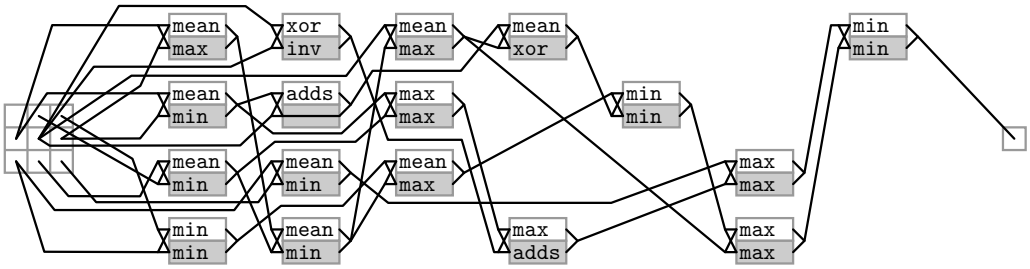
Tabulka 8.5: Odstranění gaussovského šumu a šumu typu sůl a pepř – druhá série konfigurací a pokusů.



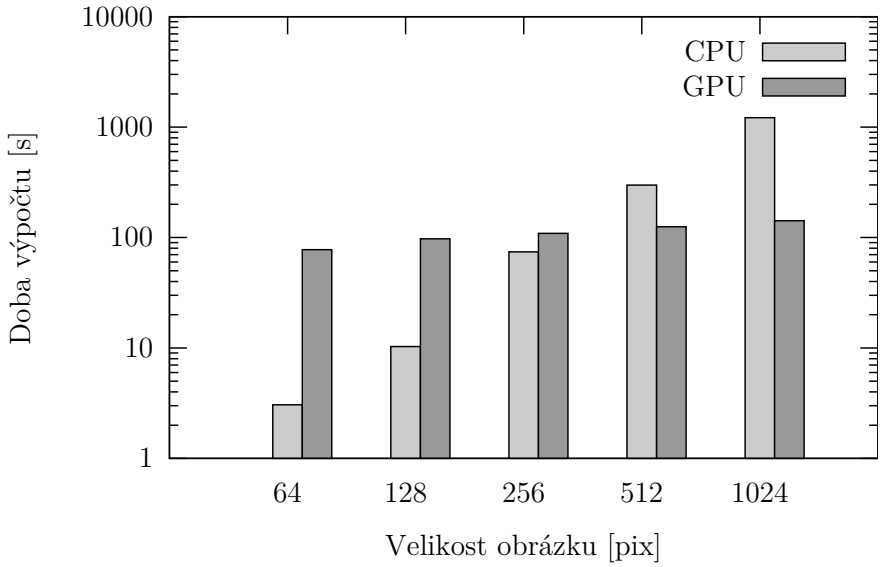
(a) Poškozený obraz (b) Výstup filtru v prvním režimu (c) Ideální výstupní obraz

(d) Poškozený obraz (e) Výstup filtru ve druhém režimu (f) Ideální výstupní obraz

Obrázek 8.7: Detail testovacího obrázku poškozeného gaussovským šumem (a) s odpovídajícím výstupem polymorfního filtru v prvním režimu (b), detail testovacího obrázku poškozeného 5% šumem typu sůl a pepř (d) s odpovídajícím výstupem polymorfního filtru ve druhém režimu. Detail ideálních výstupů (c), (f).



Obrázek 8.8: Nejlepší nalezené řešení odstranění gaussovského šumu a šumu typu sůl a pepř



Obrázek 8.9: Doba výpočtu na CPU a GPU pro různé velikosti obrázků

Navržený polymorfní filtr se v prvním polymorfním režimu pokouší odstranit gaussovský šum odstraněním vysokých frekvencí v obrázku. Lze to zjistit vizuálním posouzením výstupních obrázků, které jsou rozmazanější než originály, a dále ze samotného filtru, který v prvním režimu nejčastěji používá funkci mean, tedy průměr dvou hodnot. Utlumení vysokých frekvencí je nejrozumnější způsob, jak omezit tento typ šumu.

Druhý režim určený pro odstranění šumu typu sůl a pepř pracuje tak, že kaskádově kombinuje nejčastěji funkce min, max a v menší míře některé další. Přesná podstata jeho fungování je však nejasná. Výsledky produkované tímto filtrem ve druhém režimu jsou vizuálně kvalitnější než výsledky dosažené u předchozího polymorfního filtru, který v prvním režimu realizoval přibližný Sobelův filtr.

Na celkové struktuře filtru je zajímavé, že jeden z rohových pixelů filtračního jádra je zcela ignorován.

8.3 Srovnání výkonu standardní a akcelerované metody výpočtu

Vzhledem k tomu, že vedle standardní implementace výpočtu využívající pouze CPU existuje také akcelerovaná verze, je na místě analyzovat její výhodnost. Pro výkonnostní testy byl zvolen evoluční návrh polymorfního filtru pro dilataci a erozi. Další parametry byly následující: Velikost populace byla 5 jedinců, počet generací byl 500, dále 1 mutovaný gen na potomka, parametr L-back byl 2 a velikost CGP pole byla 4×4. Testy byly prováděny na notebooku s CPU Pentium M 1,8 GHz s grafickou kartou GeForce 7300 Go.

Byly vytvořeny trénovací obrázky o velikostech 64×64, 128×128, 256×256, 512×512 a 1024×1024 pixelů. Výsledné doby návrhu pro každý z obrázků jsou znázorněny v grafu na obrázku 8.9.

Z výsledků vyplývá, že GPU akcelerace ve formě, v jaké ji implementuje tento projekt, je výhodná až při velikostech trénovacího obrázku nad 256×256 pixelů. Tato hodnota ovšem

závisí také na konfiguraci testovacího hardware a verzi grafického ovladače. Není vyloučeno, že na jiném hardware a jiné verzi ovladačů by vyšla jiná hodnota.

Pochopení dosažených výsledků vyžaduje hlubší analýzu dob běhu jednotlivých částí algoritmu. Omezíme se na nejnáročnější část, kterou je výpočet fitness funkce, a z tohoto výpočtu konkrétně na operace potřebné pro zpracování vstupního obrázku kandidátním filtrem a výpočet diference výsledku a ideálního řešení. Pro testovací evoluční běh bylo postupně generováno 500 generací po 5 jedincích, ze kterých byl vždy jeden rodič, celkem tedy bylo provedeno $500 \times 4 = 2000$ výpočtů hodnoty fitness. Trénovací obrázky měly velikost 512×512 pixelů.

Celková doba výpočtu všech fitness hodnot na CPU byla 91,81 sekund. Zpracování obrázku filtrem trvalo 92,18 % času a výpočet sumy diferencí od ideálního obrázku 7,82 % času.

Doba výpočtu všech fitness hodnot na GPU byla 93,98 sekund. Úprava zdrojového kódu fragment shaderu trvala 0,11 % času, doba jeho kompilace trvala 52,63 % času. Zpracování obrázku a výpočet rozdílu jednotlivých pixelů se provedl offscreen renderingem, což trvalo 25,28 % času, zkopírování výsledku do paměti RAM z paměti grafické karty trvalo 20,31 % času. Součet rozdílů jednotlivých pixelů trval 1,67 % času.

Na CPU se každý pixel obrázku zpracovává zvlášť a pro tento výpočet není nutná žádná zvláštní příprava, většinu času i výpočetní náročnosti zabírá samotný výpočet výsledků filtrace. Při výpočtu na GPU naopak trvá značnou část doby kompilace fragment shaderu a kopírování mezivýsledku z paměti grafické karty do paměti RAM. Samotné zpracování jednotlivých pixelů probíhá na GPU, ale v případě obrázku o velikosti 512×512 pixelů trvá pouze přibližně čtvrtinu celkového času výpočtu. V případě menších obrázků je rozdíl mezi dobou kompilace fragment shaderu, která je nezávislá na velikosti obrázku, a zpracováním jednotlivých pixelů ještě výraznější. Úzkým hrdlem GPU akcelerace je CPU provádějící kompilaci shaderů, dále potom rychlost systémové sběrnice, přes kterou se kopírují výsledná data z paměti grafické karty do paměti RAM.

Kapitola 9

Závěr

Úkolem diplomové práce bylo navrhnout a implementovat algoritmus pro evoluční návrh polymorfních obrazových filtrů. Následně bylo zapotřebí pomocí algoritmu navrhnout konkrétní obrazové filtry.

Implementace pomocí C++ se ukázala jako velmi účinná a použitelná v praxi, zejména pokud jsou k dispozici výpočetní stroje disponující dostatečným výkonem.

Přínos akcelerované verze algoritmu pro akceleraci evolučního návrhu je diskutabilní zejména z hlediska dosaženého výkonu při zpracování malých obrázků, které se v evolučním návrhu typicky používají. Dalším problémem akcelerované verze algoritmu je to, že pro svůj běh vyžaduje plnohodnotný X server. Na vzdáleném GPU serveru se ji nepodařilo zprovoznit. Na běžné desktopové instalaci systému Linux funguje bez problémů a věrně replikuje funkci standardní implementace.

Evolučním algoritmem byly navrženy tři různé polymorfní obrazové filtry. První z nich, filtr pro dilataci a erozi, dosahoval velmi dobrých výsledků zejména pro jednoduchou až triviální řešitelnost tohoto problému. Druhý polymorfní filtr, který replikoval funkci Sobelova filtru a odstranění šumu typu sůl a pepř, dosáhl poměrně dobré vizuální kvality zejména v prvním režimu. Odstranění šumu dopadlo hůře, dosažené výsledky byly vizuálně horší než výsledky nejlepších nepolymorfních obrazových filtrů evolučně navržených pro tento účel. Třetí navržený polymorfní filtr, filtr pro odstranění gaussovského šumu a šumu typu sůl a pepř, fungoval pro gaussovský šum dle očekávání. Také část pro odstranění šumu typu sůl a pepř dosahovala velmi dobré vizuální kvality.

Ukázalo se, že pro některé dvojice obrazových filtrů funguje návrh polymorfních filtrů lépe, zatímco pro jiné hůře. Není vždy jednoduché určit, které dvojice standardních filtrů jsou vhodné pro propojení do filtrů polymorfních. Může docházet k tomu, že jeden ze dvou polymorfních režimů navrženého filtru zlepšuje své výsledky na úkor druhého, a proto bývají výsledky z hlediska vizuální kvality mírně nesymetrické.

Dalším zajímavým výsledkem pokusů byl paradox, kdy omezení variability polymorfních funkcí vedlo k lepším výsledkům. Mohlo to být zmenšením prohledávacího prostoru, které je důsledkem tohoto omezení. Případně mohla pomoci zvýšená pravděpodobnost použití funkcí nepolymorfních, které v polymorfních obrazových filtrech hrají také významnou roli.

9.1 Možnosti pokračování

Existuje velké množství různých kombinací dvojic obrazových filtrů, pro které má smysl zkoušet evoluční návrh polymorfních obrazových filtrů. Pro každou dvojici mohou být op-

timální parametry evolučního algoritmu odlišné. Zároveň není jednoduché je dopředu odhadnout, takže se tím otevírá obrovský prostor pro nejrůznější pokusy s algoritmem.

Je možné rozšířit algoritmus samotný, a to například rozšířením množiny elementárních funkcí pro výpočetní elementy. Algoritmus je také možné zobecnit tak, aby umožňoval návrh polymorfních filtrů s více než dvěma režimy.

Dále je možné vylepšit akcelerovanou verzi algoritmu. Je zde prostor pro optimalizaci fragment shaderu, aby jeho kompilace trvala kratší dobu, a to zejména eliminací nevyužitých funkcí a elementů ze zdrojového kódu. Bylo by také možné vyzkoušet změnu přístupu a vytvořit složitý fragment shader, který by se zkompiloval pouze jedenkrát a jehož běh by byl řízen změnou vnějších parametrů. Případně je možné místo knihovny OpenGL použít jiné knihovny pro práci s GPU, například CUDA nebo OpenCL.

Literatura

[1] Ahn, S. H.: OpenGL Frame Buffer Object (FBO) [online]. 2008 [cit. 2011-05-12].
URL http://www.songho.ca/opengl/gl_fbo.html

[2] Banzhaf, W.; Harding, S.; Langdon, W. B.; aj.: Accelerating Genetic Programming through Graphics Processing Units. In *Genetic Programming Theory and Practice VI*, editace R. L. Riolo; T. Soule; B. Worzel, Genetic and Evolutionary Computation, Ann Arbor: Springer, 2008, s. 229–249.

[3] Farid, H.: Fundamentals of Image Processing. 2008.
URL http://www.cs.dartmouth.edu/farid/tutorials/fip.pdf

[4] Harding, S.: Evolution of Image Filters on Graphics Processor Units Using Cartesian Genetic Programming. In *2008 IEEE World Congress on Computational Intelligence*, editace J. Wang, IEEE Computational Intelligence Society, Hong Kong: IEEE Press, 1-6 Červen 2008.
URL http://ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4631051

[5] Luo, W.; Zhang, Z.; Wang, X.: Designing polymorphic circuits with polymorphic gates: a general design approach. *IET Circuits, Devices & Systems*, ročník 1, č. 6, 2007: s. 470–476.

[6] Miller, J. F.; Thomson, P.: Cartesian Genetic Programming. In *Proc. of the 3rd European Conference on Genetic Programming EuroGP2000*, *LNCS*, ročník 1802, Springer, 2000, s. 121–132.

[7] Myler, H. R.; Weeks, A. R.: *The Pocket Handbook of Image Processing Algorithms in C*. Prentice Hall, Inc., 1993, ISBN 0-13-642240-3.

[8] Sekanina, L.: Evolution of digital circuits operating as image filters in dynamically changing environment. In *Mendel 2002 - 8th International Conference on Soft Computing*, Brno University of Technology, 2002, ISBN 80-214-2135-5, s. 33–38.

[9] Sekanina, L.: *Evolvable Components*. Springer Verlag, 2004, ISBN 3-540-40377-9.

[10] Sekanina, L.: Evolutionary Design of Gate-Level Polymorphic Digital Circuits. In *Applications of Evolutionary Computation*, LNCS 3449, Springer Verlag, 2005, ISBN 978-3-540-25396-9, s. 185–194.

[11] Sekanina, L.; Harding, S.; Banzhaf, W.; aj.: Image Processing and CGP. In *Cartesian Genetic Programming*, editace J. Miller, Springer Verlag, 2011.

[12] Sekanina, L.; Růžička, R.; Vašíček, Z.; aj.: REPOMO32 - New Reconfigurable Polymorphic Integrated Circuit for Adaptive Hardware. In *Proc. of the 2009 IEEE Symposium Series on Computational Intelligence - Workshop on Evolvable and Adaptive Hardware*, IEEE Computational Intelligence Society, 2009, ISBN 978-1-4244-2755-0, s. 39–46.

[13] Sekanina, L.; Stareček, L.; Kotásek, Z.; aj.: Polymorphic Gates in Design and Test of Digital Circuits. *International Journal of Unconventional Computing*, ročník 4, č. 2, 2008: s. 125–142, ISSN 1548-7199.

[14] Sekanina, L.; Vašíček, Z.; Růžička, R.; aj.: *Evoluční hardware*. Academia, 2009, ISBN 978-80-200-1729-1.

[15] Wikipedia: Lineární filtr [online]. 2009-12-30 [cit. 2011-01-02]. URL http://cs.wikipedia.org/wiki/Lineární_filtr

[16] Wikipedia: Linear filter [online]. 2010-11-15 [cit. 2011-01-02]. URL http://en.wikipedia.org/wiki/Linear_filter

[17] Wikipedia: Dither [online]. 2010-12-25 [cit. 2011-01-02]. URL http://en.wikipedia.org/wiki/Dither

[18] Zemčík, P.; Španěl, M.: *Zpracování obrazu - studijní opora*. FIT VUT v Brně, 2006.

Příloha A

Obsah CD

src/ – zdrojové texty programové části.

experiments/ – konfigurační soubory a trénovací data provedených experimentů.

results/ – konfigurační soubory, genotypy a schémata nejlepších nalezených polymorfních filtrů; zpracované testovací obrázky.

tex/ – zdrojové texty technické zprávy.

report/ – technická zpráva ve formátu pdf ve verzi s odkazy a ve verzi pro tisk; zadání.

dip_cd_xsalaj02.7z – všechny předchozí složky komprimované do archivu 7z.

Příloha B

Manuál

B.1 Program `dipproj`

Toto je hlavní program celého projektu. Mezi jeho funkce patří běh evolučního algoritmu na CPU i akcelerovaně pomocí GPU. Dále dokáže pomocí navržených filtrů zpracovávat libovolné obrázky, popřípadě pro ně provádět výpočet průměrné diference. Navržené filtry lze pomocí tohoto programu vizualizovat jednak pomocí textového znázornění schématu, dále potom ukládat vektorové znázornění filtrů jako obrázky ve formátu svg.

Základní schéma parametrů, které `dipproj` přebírá, je toto:

```
dipproj {mode} {mode_option_1} {mode_option_2} ... {mode_option_n}
```

Parametr `{mode}` je vždy jednopísmenné označení režimu programu. Program má následující režimy:

`e` – evoluce pomocí CPU.

`E` – evoluce pomocí GPU.

`a` – zpracování obrázku na CPU v prvním polymorfním režimu.

`b` – zpracování obrázku na CPU ve druhém polymorfním režimu.

`A` – zpracování obrázku na GPU v prvním polymorfním režimu.

`B` – zpracování obrázku na GPU ve druhém polymorfním režimu.

`d` – výpočet průměrné odchylky mezi pixely ideálního a reálného výstupu na CPU.

`D` – výpočet průměrné odchylky mezi pixely ideálního a reálného výstupu na GPU.

`t` – textové znázornění filtru na standardní výstup.

`s` – export schématu filtru do obrázku ve formátu svg.

`C` – výpis jednoho fragment shaderu pro každý ze dvou režimů polymorfního obvodu.

B.1.1 Režimy `e` a `E`

Po spuštění v jednom z těchto režimů program provádí evoluční návrh a průběžně vypisuje informace o výsledcích, kterých bylo dosaženo. Po doběhnutí evoluce zapíše výsledky do souborů.

Program v těchto režimech přebírá následující parametry:

```
dipproj e|E {input_configuration}.conf {seed_number}|time {output_geno-
type}.geno [{initial_genotype}.geno]
```

Parametr `{input_configuration}.conf` je cesta k souboru s konfigurací evolučního návrhu. Příklad konfiguračního souboru je v příloze C. Význam většiny řádků konfiguračního souboru je zřejmý. Zastavme se pouze u řádku začínajícího `functions_fullset`, resp. `functions_subset`, který obsahuje množinu funkcí R, resp. S. Může obsahovat jednak standardní funkce představované jednoduše svým názvem (jejich seznam je v tabulce 6.1). Dále může obsahovat polymorfní funkce, které se zapisují buď jako `fun1.fun2`, nebo jako `fun1:fun2`. Zápis s tečkou znamená, že příslušná množina bude obsahovat polymorfní funkci fun1/fun2. Zápis s dvojtečkou značí, že v této množině bude jak funkce fun1/fun2, tak funkce fun2/fun1.

Další parametr určuje způsob inicializace náhodného generátoru. Pokud je to řetězec `time`, je použit aktuální čas. Pokud obsahuje celé číslo, je náhodný generátor inicializován tímto číslem.

Parametr `{output_genotype}.geno` představuje cestu k výstupnímu souboru, kam se zapíše chromozom ve formě celých čísel oddělených mezerami. Kromě tohoto souboru se zapíše také soubor `{output_genotype}.geno.prm`, který obsahuje parametry nalezeného řešení, jako je hodnota fitness, průměrná chyba na pixel a hodnota seed.

Parametr `[{initial_genotype}.geno]` je volitelný. Je to cesta k počátečnímu genotypu, jehož formát je stejný jako u `{output_genotype}.geno`. Pokud není předán, počáteční chromozom se vygeneruje náhodně.

B.1.2 Režimy `a`, `b`, `A` a `B`

Program v těchto režimech přebírá následující parametry:

```
dipproj a|b|A|B {input_configuration}.conf {input_genotype}.geno {in-
put_image}.png {output_image}.png
```

Pomocí konfigurace `{input_configuration}.conf` a genotypu `{input_genotype}.geno` se sestaví polymorfní filtr. Zpracuje se jím obrázek ze souboru `{input_image}.png` a je uložen do souboru `{output_image}.png`. Zpracování probíhá podle zvoleného režimu buď na CPU, nebo na GPU.

B.1.3 Režimy `d` a `D`

Program zpracuje vstupní obrázky a vypíše průměrnou odchylku mezi dosaženým a ideálním výstupem.

Program v těchto režimech přebírá následující parametry:

```
dipproj d|D {input_configuration}.conf {input_genotype}.geno {input_a}-
.png {input_a_filtered}.png {input_b}.png {input_b_filtered}.png
```

Parametry `{input_configuration}.conf` a `{input_genotype}.geno` definují obrazový filtr. Parametr `{input_a}.png` je vstupní obrázek pro první režim obrazového filtru, `{input_a_filtered}.png` je ideální výstupní obrázek pro první režim obrazového filtru. Parametry `{input_b}.png` a `{input_b_filtered}.png` mají podobný význam a jsou určeny pro druhý režim obrazového filtru.

B.1.4 Režim `t`

Program textově znázorní schéma zadaného obrazového filtru.

Program v tomto režimu přebírá následující parametry:

```
dipproj t {input_configuration}.conf {input_genotype}.geno
```

Parametry `{input_configuration}.conf` a `{input_genotype}.geno` definují obrazový filtr.

B.1.5 Režim `s`

Program v tomto režimu uloží schéma obrazového filtru do vektorového obrázku formátu svg.

Program v tomto režimu přebírá následující parametry:

```
dipproj s {input_configuration}.conf {input_genotype}.geno {output_mode} {output_schematic}.svg
```

Parametry `{input_configuration}.conf` a `{input_genotype}.geno` definují obrazový filtr. Parametrem `{output_mode}` se nastavuje výstup:

`0` – Obrázek bude obsahovat pouze ty výpočetní elementy, které jsou využity. Nebude obsahovat číslování.

`1` – Obrázek bude obsahovat všechny výpočetní elementy. Nebude obsahovat číslování.

`2` – Obrázek bude obsahovat pouze ty výpočetní elementy, které jsou využity. Bude také obsahovat číslování.

`3` – Obrázek bude obsahovat všechny výpočetní elementy. Bude také obsahovat číslování.

Parametr `{output_schematic}.svg` je cesta k výstupnímu souboru pro uložení obrázku.

B.1.6 Režim `C`

Program vypíše oba fragment shadery vygenerované pro obrazový filtr.

Program v tomto režimu přebírá následující parametry:

```
dipproj C {input_configuration}.conf {input_genotype}.geno
```

Parametry `{input_configuration}.conf` a `{input_genotype}.geno` definují obrazový filtr.

B.2 Program `procresults`

Tento program je určen pro souhrnné zpracování výsledků evolučních pokusů. Jako parametry přebírá cesty k souborům `*.geno.prm`. Tyto soubory analyzuje na základě jejich názvů, rozdělí je do skupin a nakonec vypíše průměrné a nejlepší výsledky pro každou skupinu pokusů.

B.3 Program `spnoisegen`

Tento program je určen pro vygenerování šumu typu sůl a pepř. Přebírá následující parametry:

```
spnoisegen {image_in}.png {image_out}.png {percent}
```

Parametr `{image_in}.png`, resp. `{image_out}.png` je vstupní, resp. výstupní obrázek. Parametr `{percent}` je procento zašuměných pixelů.

B.4 Program `gaussnoisegen`

Tento program je určen k vygenerování gaussovského šumu. Přebírá následující parametry:

```
gaussnoisegen {image_in}.png {image_out}.png {sigma}
```

Parametr `{image_in}.png`, resp. `{image_out}.png` je vstupní, resp. výstupní obrázek. Parametr `{sigma}` je parametr σ gaussovského šumu. Je vztažen k hodnotám obrázku přemapovaným na rozsah $\langle 0{,}0; 1{,}0\rangle$.

Příloha C

Příklad konfigračního souboru

```
population_size=5
generations_count=40000
mutated_genes=2
cgp_width=7
cgp_height=4
lback=4
functions_fullset=const ident or nor and nand xor nxor _or inv div2 di...
functions_subset=const:ident const:or const:nor const:and const:nand c...
functions_subset_size=8
imagea_source="indata/sm01-gauss0.1.png"
imagea_target="indata/sm01-orig.png"
imageb_source="indata/sm01-spten.png"
imageb_target="indata/sm01-orig.png"
```

Příloha D

Příklady GLSL shaderů

Zdrojové texty obsažené v této příloze byly přeformátovány pro snazší čitelnost. Vzhledem k tomu, že např. fragment shadery byly generovány programově, obsahovaly méně mezer a ostatních bílých znaků.

D.1 Sdílený vertex shader

```
#version 120
void main() {
    gl_Position = ftransform();
}
```

D.2 První příklad fragment shaderu

Tento fragment shader byl vygenerován pro první režim polymorfního filtru, jehož schéma je na obrázku 8.3.

```
#version 120
uniform sampler2D tex0, tex1, tex2;
uniform vec2 dims;

float f1(float x, float y) {
    return 217;
}
float f2(float x, float y) {
    return x;
}
float f3(float x, float y) {
    return texture2D(tex2,vec2(x+0.1,y+0.1)/256).g*255;
}
float f4(float x, float y) {
    return 255-texture2D(tex2,vec2(x+0.1,y+0.1)/256).g*255;
}
float f5(float x, float y) {
    return texture2D(tex2,vec2(x+0.1,y+0.1)/256).r*255;
}
```

```
float f6(float x, float y) {
    return 255-texture2D(tex2,vec2(x+0.1,y+0.1)/256).r*255;
}
float f7(float x, float y) {
    return texture2D(tex2,vec2(x+0.1,y+0.1)/256).b*255;
}
float f8(float x, float y) {
    return 255-texture2D(tex2,vec2(x+0.1,y+0.1)/256).b*255;
}
float f9(float x, float y) {
    return texture2D(tex2,vec2(255-x+0.1,y+0.1)/256).g*255;
}
float fa(float x, float y) {
    return 255-x;
}
float fb(float x, float y) {
    return floor(x*0.5+0.1);
}
float fc(float x, float y) {
    return floor(x*0.25+0.05);
}
float fd(float x, float y) {
    return floor(mod(x+y+0.6, 256));
}
float fe(float x, float y) {
    return min(x+y, 255);
}
float ff(float x, float y) {
    return floor((x+y+0.4)*0.5);
}
float fg(float x, float y) {
    return max(x,y);
}
float fh(float x, float y) {
    return min(x,y);
}

void main() {
    float x=gl_FragCoord.x;
    float y=gl_FragCoord.y;
    float
        a00=texture2D(tex0, vec2((x-1)/dims.x, (y-1)/dims.y)).r*255,
        a01=texture2D(tex0, vec2(x/dims.x, (y-1)/dims.y)).r*255,
        a02=texture2D(tex0, vec2((x+1)/dims.x, (y-1)/dims.y)).r*255,
        a03=texture2D(tex0, vec2((x-1)/dims.x, y/dims.y)).r*255,
        a04=texture2D(tex0, vec2(x/dims.x, y/dims.y)).r*255,
        a05=texture2D(tex0, vec2((x+1)/dims.x, y/dims.y)).r*255,
        a06=texture2D(tex0, vec2((x-1)/dims.x, (y+1.0)/dims.y)).r*255,
```

```
        a07=texture2D(tex0, vec2(x/dims.x, (y+1)/dims.y)).r*255,
        a08=texture2D(tex0, vec2((x+1)/dims.x, (y+1)/dims.y)).r*255,
        a09=fg(a02,a06),
        a0a=fa(a05,a01),
        a0b=fg(a03,a05),
        a0c=fg(a08,a00),
        a0d=f2(a0c,a0b),
        a0e=fg(a01,a09),
        a0f=fg(a0b,a07),
        a10=fc(a02,a05),
        a11=f2(a0d,a0f),
        a12=fg(a0d,a0e),
        a13=f8(a0e,a0d),
        a14=fd(a0b,a0f),
        a15=fa(a13,a10),
        a16=f6(a13,a10),
        a17=f8(a0d,a11),
        a18=fg(a12,a0f),
        n;
    float r=a18;
    float t=texture2D(tex1, vec2(x/dims.x, y/dims.y)).r*255;
    gl_FragColor.r = abs(r-t)/255;
}
```

D.3 Druhý příklad fragment shaderu

Tento fragment shader byl vygenerován pro druhý režim polymorfního filtru, jehož schéma je na obrázku 8.3. Je identický s předchozím až na řádky začínající `a09` až `a18`. Tyto odlišné řádky jsou uvedeny zde:

```
        a09=fh(a02,a06),
        a0a=f8(a05,a01),
        a0b=fh(a03,a05),
        a0c=fh(a08,a00),
        a0d=fh(a0c,a0b),
        a0e=fh(a01,a09),
        a0f=f3(a0b,a07),
        a10=f9(a02,a05),
        a11=f2(a0d,a0f),
        a12=fh(a0d,a0e),
        a13=fg(a0e,a0d),
        a14=fe(a0b,a0f),
        a15=f3(a13,a10),
        a16=fd(a13,a10),
        a17=fa(a0d,a11),
        a18=fh(a12,a0f),
```