VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

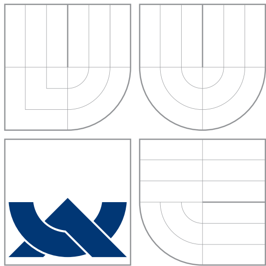
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH PRO APROXIMACI OBVODŮ

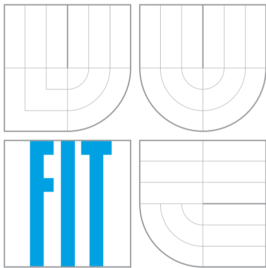
DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR DVOŘÁČEK
AUTHOR

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

EVOLUČNÍ NÁVRH PRO APROXIMACI OBVODŮ

EVOLUTIONARY DESIGN FOR CIRCUIT APPROXIMATION

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. PETR DVOŘÁČEK
AUTHOR

VEDOUCÍ PRÁCE — Prof. Ing. LUKÁŠ SEKANINA, Ph.D.
SUPERVISOR

BRNO 2015

Abstrakt

V posledních letech klademe stále větší důraz na energetickou úspornost integrovaných obvodů. Můžeme vytvořit aproximační obvody, které nesplňují specifikovanou logickou funkci, a které jsou cíleně navrženy ke snížení velikosti, doby odezvy a příkonu. Tyto nepřesné obvody lze využít v mnoha aplikacích, kde lze tolerovat chyby, obzvláště v aplikacích ve zpracování signálů a obrazu, počítačové grafiky a strojového učení. Tato práce popisuje evoluční přístup k návrhu aproximačních aritmetických obvodů a dalších složitějších obvodů. Díky paralelnímu výpočtu fitness byl evoluční návrh osmibitových násobiček urychlen až 170 krát oproti standardnímu přístupu. Pomocí inkrementální evoluce byly vytvořeny různé aproximační aritmetické obvody. Vyvinuté obvody byly použity v různých typech detektorů hran.

Abstract

In recent years, there has been a strong need for the design of integrated circuits showing low power consumption. It is possible to create intentionally approximate circuits which don't fully implement the specified logic behaviour, but exhibit improvements in term of area, delay and power consumption. These circuits can be used in many error resilient applications, especially in signal and image processing, computer graphics, computer vision and machine learning. This work describes an evolutionary approach to approximate design of arithmetic circuits and other more complex systems. This text presents a parallel calculation of a fitness function. The proposed method accelerated evaluation of 8-bit approximate multiplier 170 times in comparison with the common version. Evolved approximate circuits were used in different types of edge detectors.

Klíčová slova

aproximace, přibližné počítání, evoluční návrh obvodů, kartézské genetické programování, měření chyby, optimalizace obvodů

Keywords

approximation, approximate computing, evolutionary circuit design, cartesian genetic programming, error measurement, circuit optimization

Citace

Petr Dvořáček: Evoluční návrh pro aproximaci obvodů, diplomová práce, Brno, FIT VUT v Brně, 2015

Evoluční návrh pro aproximaci obvodů

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana Lukáše Sekaniny. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Petr Dvořáček
27. května 2015

Poděkování

Rád bych poděkoval prof. Lukáši Sekaninovi za trpělivost, vedení a odbornou pomoc při řešení této práce.

© Petr Dvořáček, 2015.
Tato práce vznikla jako školní dílo na Vysokém učení technickém v Brně, Fakultě informačních technologií. Práce je chráněna autorským zákonem a její užití bez udělení oprávnění autorem je nezákonné, s výjimkou zákonem definovaných případů.

Obsah

1 Úvod 5

2 Číslicové obvody 7
2.1 Parametry číslicových obvodů . . . 8
2.1.1 Plocha . . . 8
2.1.2 Zpoždění . . . 8
2.1.3 Příkon . . . 9
2.2 Návrh aritmetických číslicových obvodů . . . 9
2.2.1 Sčítačka . . . 10
2.2.2 Odčítačka . . . 12
2.2.3 Násobička . . . 12

3 Aproximační obvody 15
3.1 Metriky pro stanovení chyby aproximačních obvodů . . . 15
3.2 Konvenční návrh aproximačních aritmetických obvodů . . . 17
3.2.1 Přiřazení konstanty na výstup . . . 17
3.2.2 Zanedbání nejméně významných bitů . . . 18
3.2.3 Karnaughovy mapy v aproximaci . . . 18
3.2.4 Aritmetické aproximační obvody jako stavební bloky . . . 19
3.2.5 SALSA . . . 20
3.2.6 ABACUS . . . 20

4 Evoluční návrh pomocí kartézského genetického programování 22
4.1 Vícekriteriální optimalizace . . . 23
4.2 Fitness funkce pro aproximační obvody . . . 24
4.2.1 Suma Hammingových vzdáleností . . . 24
4.2.2 Vážená suma Hammingových vzdáleností . . . 24
4.2.3 Suma absolutních diferencí . . . 25
4.3 Evoluční aproximace komplexnějších aritmetických obvodů . . . 25
4.3.1 Evoluční aproximace obvodů podle různých chybovostních metrik . . 25
4.3.2 Evoluční návrh aproximačních obvodů pomocí heuristické inicializace 26

5 Detekce hran 27
5.1 Metody založené na první derivaci . . . 27
5.2 Cannyho detektor hran . . . 29

6 Akcelerace ohodnocení aproximovaného kandidátního řešení **31**
6.1 Paralelní simulace . . . 32
6.2 Simulace fenotypu a přeskočení neutrálních mutací . . . 32
6.3 Předkompilace chromozomu a fitness funkce . . . 32
6.3.1 Experimentální vyhodnocení JIT kompilací . . . 32
6.4 Paralelní simulace výpočtu fitness hodnoty pro aritmetické obvody . . . 33
6.4.1 Experimentální vyhodnocení paralelního výpočtu . . . 34

7 Evoluce aproximačních aritmetických obvodů **36**
7.1 Návrh přibližných sčítaček a jejich vyhodnocení . . . 36
7.2 Návrh přibližných násobiček a jejich vyhodnocení . . . 36
7.3 Graf rozptylu chyb pro aritmetické obvody . . . 36

8 Použití navržených přibližných obvodů v detekci hran **40**
8.1 Přibližná Sobelova detekce hran . . . 40
8.1.1 Hardwarová implementace a návrh aproximace . . . 40
8.1.2 Experimentální vyhodnocení . . . 41
8.2 Přibližná Cannyho detekce hran . . . 43
8.2.1 Návrh aproximace . . . 43
8.2.2 Tvorba přibližných násobiček podle částečné specifikace . . . 44
8.2.3 Experimentální vyhodnocení . . . 44

9 Závěr **47**

A Obsah CD **50**

B Použití implementovaných programů **51**

Seznam obrázků

2.1 Multiplexor na úrovni hradel . . . 7
2.2 Zřetězená sčítačka . . . 10
2.3 Struktura sčítačky s logikou generování přenosu . . . 11
2.4 Logika přenosu pětibitové Kogge-Stone sčítačky . . . 12
2.5 Čtyřbitová násobička . . . 12
2.6 Optimalizovaná čtyřbitová násobička . . . 13
2.7 Wallaceův strom . . . 14

3.1 Graf rozptylu chyb . . . 17
3.2 Plně funkční a přibližná násobička . . . 20

4.1 Zakódovaní grafu v CGP s příkladem mutace . . . 23

5.1 Gradient hrany . . . 27
5.2 Detekce pomocí Sobelových operátorů . . . 28
5.3 Kroky Cannyho detekce hran . . . 29

6.1 Paralelní výpočet fitness hodnoty SAD . . . 34

7.1 Grafy chyb vyvinutých přibližných sčítaček . . . 38
7.2 Grafy chyb vyvinutých přibližných násobiček . . . 39

8.1 Návrh hardwarové implementace Sobelova detektoru hran . . . 41
8.2 Použití různě aproximovaných sčítaček v Sobelově detekci hran . . . 42
8.3 Přibližné úhly hran . . . 43
8.4 Využití operandů násobení v procesu ztenčení hran . . . 44
8.5 Porovnání chybovosti násobiček s použitím v detekci hran . . . 45
8.6 Aproximace ztenčení hran za použití přibližných násobiček . . . 46

Seznam tabulek

2.1 Pravdivostní tabulka multiplexoru . . . 7
2.2 Plocha a zpoždění hradel podle NAND logiky VLSI technologie . . . 9

3.1 Aproximace logických funkcí pomocí Karnaughovy mapy . . . 18
3.2 Upravená Karnaughova mapa pro aproximační násobičku 2x2 bity . . . 19
3.3 Chybovosti složených přibližných násobiček . . . 19

5.1 Prewittové operátory . . . 28
5.2 Sobelovy operátory . . . 28

6.1 Časová analýza algoritmu CGP . . . 31
6.2 Vyhodnocení akcelerace CGP pomocí JIT kompilace fitness funkce . . . 33
6.3 Porovnání rychlostí výpočtu funkce SAD. . . . 35

7.1 Vlastnosti evolučně navržených sčítaček . . . 37
7.2 Vlastnosti evolučně navržených násobiček . . . 37

8.1 Parametry hardwarové implementace přesného Sobelova detektoru hran . . . 41

Kapitola 1

Úvod

V roce 1965 Gordon E. Moore, spoluzakladatel společnosti Intel, napsal článek, ve kterém předpovídal, že se každé dva roky dvojnásobně zvýší počet tranzistorů v integrovaných obvodech. Věřil, že tento trend zůstane neměnný po dobu deseti let. Po padesáti letech, kdy jsou integrované obvody vyráběny technologií 14 nm a s příchodem mobilních zařízení, řešíme zejména jiný požadavek než hustotu integrace – energetickou úspornost. Položme si tedy otázku, jak navrhnout energeticky úsporné obvody?

Aproximační počítání se jeví jako perspektivní řešení. Vychází z předpokladu, že můžeme vytvořit výkonné a energeticky nenáročné systémy za cenu vyšší chybovosti. Byla představena řada aplikací, ve kterých lze tolerovat chyby [5]. Lidské smysly, zejména zrak a sluch, nemusí chyby rozpoznat. Tento fakt je řadu let využíván v kompresi multimediálních dat. Tento princip může být také použit ve filtraci anebo počítačovém vidění. Další toleranci vůči chybám můžeme pozorovat v získávání znalostí z databází, kde je prakticky nemožné získat prokazatelně optimální výsledek. Najdeme ji také v aplikacích, kde člověk chyby očekává a toleruje, ku příkladu v předpovědi počasí, či klasifikaci.

Aproximační systémy můžeme navrhnout jak v softwaru (např. rychlý výpočet funkce $f(x) = 1/\sqrt{x}$ [3]), tak i v hardwaru na všech úrovních. Byly představeny přibližné sekvenční a aritmetické obvody (např. SRAM [7], sčítačky [26], násobičky [11], či komparátory [14]) a složitější obvody (např. perceptron [15], FFT, či diskrétní kosinová transformace [23]). Byly navrženy přibližné maticové procesory, ale i programovací jazyky určené k aproximačnímu počítání. Nejpoužívanější aproximací na moderních počítačích je však interpretace čísel v pohyblivé řádové čárce, která reprezentuje reálná čísla [1].

V této práci se zabývám evolučním návrhem aproximačních obvodů. Tímto názvem označujeme číslicové obvody, které nesplňují svou logickou funkci definovanou specifikací, a které jsou cíleně navrženy k snížení velikosti, spotřeby nebo latence. Evoluční návrh umožňuje vytvořit různá inovativní řešení, která mohou být lepší než konvenční řešení navržená člověkem. Příkladem může být evolučně navržená anténa, která byla použita v několika vesmírných misích, dále pak návrh optických systémů, či analogových obvodů [17]. Bylo ukázáno, že můžeme evolučně navrhnout a optimalizovat i logické obvody [13]. V současnosti se zjistilo, že můžeme pomocí evolučních algoritmů navrhovat i aproximační obvody [16], [20], [21].

Cílem této práce je navázat na předchozí výzkum v oblasti evolučního návrhu aproximačních obvodů [16], [20], [21]. V práci [16] se navržená fitness funkce pro každý trénovací vektor počítala zvlášť. Tato diplomová práce představuje nový paralelní výpočet fitness, který umožňuje významně urychlit celý evoluční návrh přibližně pracujících obvodů. Optimalizovaný algoritmus použijeme v návrhu přibližných 8-bitových násobiček a sčítaček.

Následně demonstrujeme použitelnost vyvinutých aproximačních obvodů v detekci hran.

Tato práce je členěna následovně. Ve 2. kapitole jsou vysvětleny základní principy číslicových obvodů, z čeho se skládají, jak je navrhnout a jaké jsou jejich parametry pro posuzování. Kapitola 3 prezentuje aproximační obvody a jejich konvenční návrh. Kapitola 4 se zabývá evolučním návrhem aproximačních obvodů pomocí kartézského genetického programování. V kapitole 5 jsou popsány principy detekce hran. Kapitola 6 popisuje různé akcelerace ohodnocení zároveň prezentuje paralelní výpočet fitness. V kapitole 7 jsou vyhodnoceny nalezené přibližné obvody a zároveň je prezentován nový typ grafu chyb pro aritmetické obvody. V kapitole 8 je popsáno použití aproximačních obvodů v detekci hran. Kapitola 9 shrnuje dosažené výsledky a uzavírá tuto práci.

Kapitola 2

Číslicové obvody

Hradla jsou nejjednodušší číslicové obvody, které realizují Booleovské funkce. Každé hradlo má jeden nebo více vstupů, podle nichž se na základě jejich funkce vytváří výstup. Vstupy a výstupy jsou modelovány jako digitální signály nabývající právě dvou diskrétních hodnot, které nazýváme 0 a 1 (či nepravda a pravda). Ve skutečnosti však pracují na úrovni analogové, kde signál je podmíněn napětím a proudem [24].

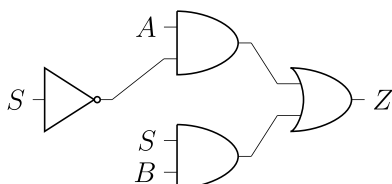
Tři základní hradla jsou dvouvstupové hradlo AND realizující konjunkci, OR provádějící disjunkci a jednovstupové NOT, které realizuje negaci. Pomocí těchto tří hradel můžeme provést libovolnou složitější Booleovskou funkci (např. sčítačky, násobičky, filtry). Kombinačními obvody nazýváme takové číslicové obvody, jejichž výstupy závisí pouze na dané vstupní kombinaci. Oproti tomu u sekvenčních obvodů záleží i na stavu vyvolaném předchozími vstupními kombinacemi. Jinými slovy, sekvenční obvody obsahují paměť.

Tradiční metodou návrhu jednoduchých kombinačních obvodů je specifikace podle pravdivostní tabulky, ve které pro každou vstupní kombinaci najdeme výstupní kombinaci. Například multiplexor o třech vstupech má celkem 2^3 (tedy 8) vstupních kombinací, jak je naznačeno v tabulce 2.1. Tuto tabulku můžeme vyjádřit pomocí Booleovy algebry v disjunktní normální formě (zkráceně DNF) a výsledný vztah minimalizovat pomocí optimalizačního algoritmu. Pro multiplexor tedy získáme vztah ve tvaru

$$Z = \overline{S} \cdot A + S \cdot B. \tag{2.1}$$

V dalším kroku můžeme tento vztah převést na odpovídající kombinační obvod, viz obrázek 2.1. Pro automatizovaný návrh složitých číslicových obvodů existuje řada metod, které jsou implementovány v návrhových softwarech.

Integrované obvody se skládají z jednoho či více hradel a jsou integrovány do samotného celku, kterému říkáme čip. V minulosti se tyto obvody dělily podle velikostí na malé (obsa-



Obrázek 2.1: Multiplexor na úrovni hradel.

S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1

Tabulka 2.1: Pravdivostní tabulka multiplexoru.

hující $1 - 20$ hradel), střední ($20 - 200$ hradel) a velké ($200 - 200000$ hradel). V současnosti integrované obvody čítají více než 50 miliónů tranzistorů a řadíme je mezi VLSI z angl. very large-scale integration [24].

Cíleně navrženým integrovaným obvodům, které plní jistou aplikaci (např. mikrokontroléry), říkáme ASIC. Tyto obvody jsou po všech stránkách optimalizovány pro konkrétní použití. Jejich hlavní nevýhodou je výrobní cena. Kompromisem může být návrh obvodu pro rekonfigurovatelná zařízení [24].

Rekonfigurovatelné obvody FPGA, z angl. Field-Programmable Gate Array, jsou tvořeny maticí konfigurovatelných logických bloků nazývané CLB (z angl. Configurable Logic Block). Bloky CLB jsou ještě děleny na menší bloky, které se nazývají slices. Každý slice obsahuje funkční generátory, multiplexory, hradla, registry, logiky přenosu atd. Funkční generátory jsou implementované jako vyhledávací tabulky LUT (z angl. Look-Up Table), které jsou tvořeny SRAM pamětí a mohou být využity i jiným způsobem (paměť, posuvný registr) [17]. Návrh obvodů pak řeší speciální vývojové nástroje, které pro FPGA vygenerují konfigurační data.

2.1 Parametry číslicových obvodů

Při vývoji číslicových obvodů klademe důraz na několik parametrů, které se snažíme optimalizovat. Mezi ně patří plocha, zpoždění a příkon. Při měření těchto parametrů musíme také určit úroveň abstrakce, se kterou pohlížíme na obvody. Platí fakt, že čím nižší úroveň abstrakce zvolíme, tím přesnější výsledky dostaneme. Mimo to musíme vzít v potaz cílovou technologii, pro kterou obvody vyvíjíme. Pro obvody ASIC používáme jiné metriky než pro rekonfigurovatelné obvody FPGA.

2.1.1 Plocha

Plochu obvodu můžeme definovat podle celkového počtu hradel či tranzistorů, ze kterých se dané obvody skládají. Například díky pravidlům Booleovy algebry můžeme daný obvod složit pouze z NAND hradel a použít počet NAND hradel jako jednu z metrik. Chceme-li být přesnější, můžeme určit celkovou relativní plochu hradel λ pro VLSI integrované obvody. V tabulce 2.2 jsou uvedeny tyto tři parametry pro jednotlivá hradla.

Mnohem složitější je určit plochu požadovaného obvodu se zahrnutím informace o propojích. Můžeme určit celkovou délku propojů, či lépe počet jejich křížení.

2.1.2 Zpoždění

Dobu zpoždění číslicových obvodu, v práci označováno jednotkou Δ, lze vypočítat podle velikosti kritické cesty obvodu. Jedná se o nalezení maximálního počtu navštívených uzlů v dopředném orientovaném grafu, který reprezentuje obvod. Vstupy resp. výstupy grafu jsou v korespondenci se vstupy resp. výstupy obvodu. Uzly grafu pak představují hradla.

Velikost hradel je v nepřímé úměře k jejich zpoždění. Ku příkladu invertor je rychlejší než hradlo XNOR, které je samozřejmě větší. Zpoždění může být rovněž určeno pro hradla sestavených podle NAND logiky [24] anebo také na úrovni tranzistorů.

Logical effort je jedna z možných metrik pro měření zpoždění pro obvody sestavených z tranzistorů [2]. Logical effort hradla je definován jako poměr mezi vstupní kapacitou (množství elektrického náboje ve vstupním vodiči) invertoru, kterou je nutno převést na výstup, a vstupní kapacitou daného hradla tak, aby jeho výstupní kapacita byla shodná

	NAND logika		VLSI		Logical effort		
	Hradel	Zpoždění	Tranzistorů	λ	A	B	Avg.
NOT	1	1	2	24	1,00		1,00
NAND	1	1	4	32	1,29	1,35	1,32
NOR	4	3	4	32	1,67	1,63	1,65
AND	2	2	6	40	1,81	1,71	1,76
OR	3	2	6	40	2,09	1,99	2,04
XOR	4	3	10	72	1,50	1,60	1,55
XNOR	5	4	12	72	2,76	2,77	2,77

Tabulka 2.2: Tabulka uvádí počet tranzistorů, relativní plochu λ hradel ve VLSI (pro standard vsclib) a logical effort pro vstup A resp. vstup B jednotlivých hradel. Převzato z publikace [2]. Dále je znázorněn počet hradel a zpoždění v NAND logice.

k výstupní kapacitě invertoru. Lze jej definovat podle vztahu $g = C_b/C_{inv}$, kde C_b je vstupní kapacita daného hradla a C_{inv} je vstupní kapacita invertoru. Logical effort závisí nejen na topologie obvodu, ale také na výrobní technologii tranzistoru (zda se jedná o nMOS či pMOS). Hodnoty logical effort pro vybraná hradla lze nalézt v tabulce 2.2.

2.1.3 Příkon

Příkon obvodů závisí zejména na velikosti obvodu, použité technologii tranzistorů a charakteru vstupních dat. Logické obvody TTL, které jsou sestaveny z bipolárních tranzistorů, se v dnešních výpočetních systémech používají velmi zřídka. Jejich nástupce jsou unipolární logické obvody sestavené podle technologie CMOS, která se snaží potlačit nevýhody TTL. Jednou z těchto výhod je, že CMOS tranzistory mají velmi nízký statický příkon. Dynamický příkon se projevuje tehdy, když se překlápí tranzistory. Lze jej vyjádřit vztahem

$$P_{dyn} = C_{dyn} f U^2, \tag{2.2}$$

kde C_{dyn} je zátěžová kapacita. Frekvence přepínání je značena jako f a U představuje napájecí napětí. Je zřejmé, že zejména snižovaní napájecího napětí vede k výraznému snižování příkonu obvodu CMOS. U moderních technologií (65 nm a níže) však roste podíl statického příkonu vzhledem k příkonu dynamickému, a proto se statický příkon nesmí zanedbat. Můžeme jej získat dle vztahu

$$P_{stat} = U I_{leak}, \tag{2.3}$$

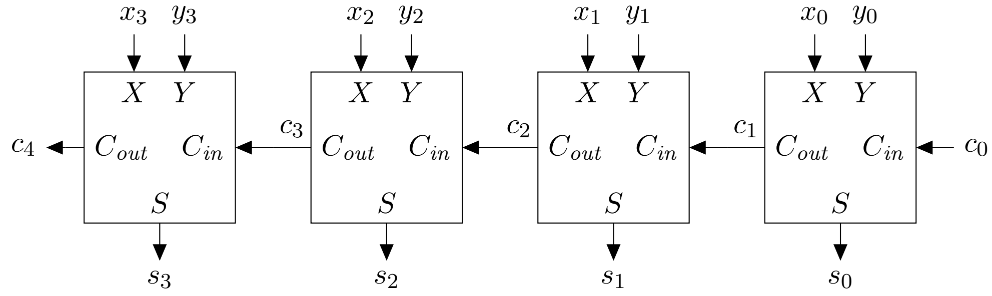
kde I_{leak} znamená zbytkový proud a U je vstupní napětí [17].

Zjištění příkonu obvodu je časově drahá záležitost, protože je nutné simulovat daný obvod. Existují různé simulátory obvodů, většina z nich je bohužel komerčních. Z nekomerčních můžeme použít například SIS, který je bohužel zastaralý a nepřesný. Nejvhodnější a také přesnou metodou je změřit příkon obvodu v reálném prostředí.

2.2 Návrh aritmetických číslicových obvodů

Aritmetické číslicové obvody implementují výpočty v binární soustavě. n-bitové číslo označíme

$$b_{n-1} b_{n-2} \dots b_1 b_0, \tag{2.4}$$



Obrázek 2.2: Sestavení sčítačky s postupným přenosem pomocí úplných sčítaček. Hodnota c_0 se nastaví na logickou 0.

kde $b_i \in \{0, 1\}$ a n je stanovený počet bitů. Nejlevější číslice odpovídá nejvíce významnému bitu (zkráceně MSB) a nejpravější je nejméně významný bit (zkráceně LSB). Hodnotu v decimální soustavě můžeme získat podle vzorce

$$B = \sum_{i=0}^{n-1} b_i 2^i. \tag{2.5}$$

V dalším textu této kapitoly $+$ značí konjunkci, $\cdot$ disjunkci a $\oplus$ exkluzivní konjunkci.

2.2.1 Sčítačka

Sčítání je jedna z nejpoužívanějších aritmetických operací v obvodech. Nejjednodušší sčítačkou je poloviční sčítačka, která sčítá dva jednobitové operandy x a y a vytváří sumu s a přenos c. Můžeme je zapsat pomocí vztahů

$$s = x \oplus y = x \cdot \overline{y} + \overline{x} \cdot y \tag{2.6}$$

$$c_{out} = x \cdot y \tag{2.7}$$

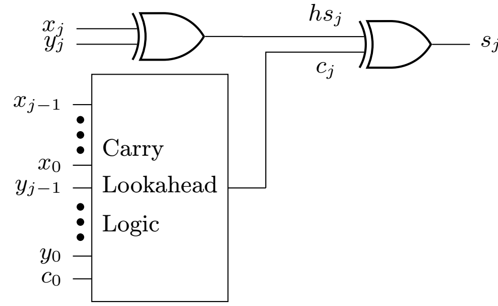
Pro součet tří bitů nám slouží úplná sčítačka, která má na vstupu navíc vstupní přenos c_{in}. Úplnou sčítačku můžeme vyjádřit podle vztahů

$$s = x \oplus y \oplus c_{in} \tag{2.8}$$

$$c_{out} = x \cdot y + x \cdot c_{in} + y \cdot c_{in} \tag{2.9}$$

Sčítačka s postupným přenosem

Máme-li několik úplných sčítaček sestrojených podle vzorců 2.8 a 2.9, potom je můžeme snadno propojit a vytvořit tak několikabitovou sčítačku. Vytvoříme ji tak, že propojíme c_{out} sčítačky se vstupem c_{in} ze sousední sčítačky. Hodnota c_{in} na LSB (nejméně významný bit) je přitom nastavena na logickou 0, viz obrázek 2.2. Problém spočívá ve zpoždění této sčítačky způsobeném nutností přenést bit přenosu do MSB (nejvíce významného bitu). Takže vygenerování výstupu s_3 uvedeném na obrázku 2.2 má zpoždění 7Δ, kde Δ značí jednotku zpoždění na úrovni hradel.



Obrázek 2.3: Struktura sčítačky s logikou generování přenosu.

Sčítačky s predikcí přenosu

Problém se zpožděním lze vyřešit použitím sčítačky s generováním přenesu zkráceně CLA (z angl. Carry Lookahead Adder). Stačí nám totiž ve vztahu 2.8 předpočítat hodnotu c_{in} a sestrojit obvod podle schématu 2.3. *Carry Lookahead Logic* je obvod, který zjistí c_{in} pro daný bit j. Tento obvod se snaží dozvědět, kdy se příznak přenosu pro určitý bit j generuje, a kdy propaguje. Generování přenosu nastane právě tehdy, když platí

$$g_j = x_j \cdot y_j. \tag{2.10}$$

Dále budeme potřebovat vztah pro propagování přenosu v j-tém bitu

$$p_j = x_j + y_j. \tag{2.11}$$

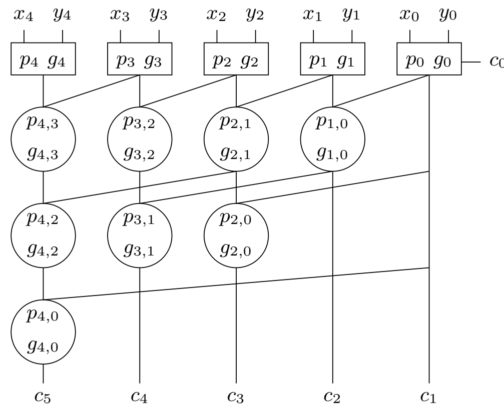
Generování ani propagování není tedy závislé na předchozím přenosu. Samotný přenos c_j můžeme zapsat zobecněným výrazem $c_{j+1} = g_j + p_j \cdot c_j$. Tento výraz rozvedeme pro první čtyři přenosy podle vztahu 2.12. Použijeme-li hradla s více než dvěma vstupy, potom získáme obvod CLA se zpožděním o velikosti 3Δ.

$$\begin{aligned}
c_1 &= g_0 + p_0 \cdot c_0 \\
c_2 &= g_1 + p_1 \cdot g_0 + p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_2 + p_2 \cdot g_1 + p_2 \cdot p_1 \cdot g_0 + p_2 \cdot p_1 \cdot p_0 \cdot c_0 \\
c_3 &= g_3 + p_3 \cdot g_2 + p_3 \cdot p_2 \cdot g_1 + p_3 \cdot p_2 \cdot p_1 \cdot g_0 + p_3 \cdot p_2 \cdot p_1 \cdot p_0 \cdot c_0
\end{aligned} \tag{2.12}$$

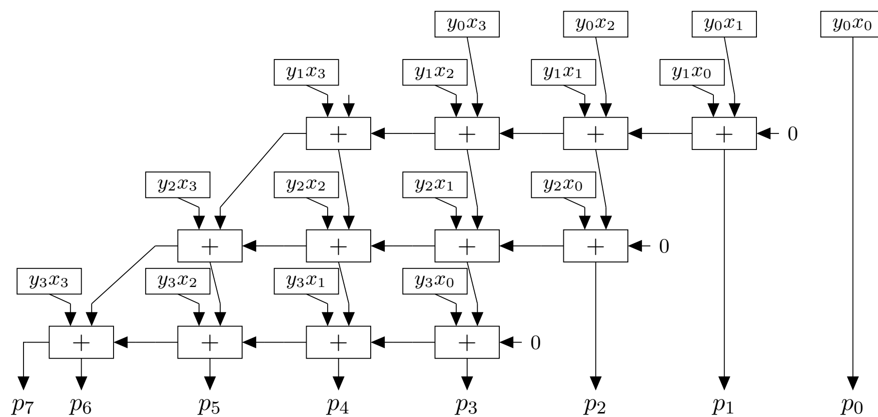
Pro predikci přenosu pomocí dvouvstupých hradel se využívají různé *carry lookahead* sítě. Mezi nejznámější patří Kogge-Stone sčítačka [10], která je pojmenovaná podle svých autorů. Jedná se o sčítačku s časovou složitostí $\mathcal{O}(\log n)$, kde n je bitová šířka operandů. Přenosy jsou vypočteny paralelně za cenu počtu použitých hradel. Princip spočívá ve využití skupinového přenosu a generování podle vztahů 2.13, pomocí kterých vytvoříme logickou síť, která je ukázána na obrázku 2.4.

$$\begin{aligned}
p_{i,j} &= p_{i,k} p_{k-1,j} \\
g_{i,j} &= g_{i,k} + p_{i,k} g_{k-1,j}
\end{aligned} \tag{2.13}$$

Existuje spousta dalších sčítaček pracujících na podobném principu, např. Sklansky, Bret-Kung a další. Tyto sčítačky mohou mít o trochu větší zpoždění než Kogge-Stone, ovšem k jejich sestrojení je potřeba méně hradel a tudíž mohou být cenově výhodnější.



Obrázek 2.4: Struktura Kogge-Stone pětibitové sčítačky s logikou generování přenosu. Obdélníky jsou výpočty první iterace podle vztahů 2.10 a 2.11. Kruhy reprezentují výsledky ze vztahů 2.13. Z hodnoty c_j získáme výsledek podle schématu 2.3.



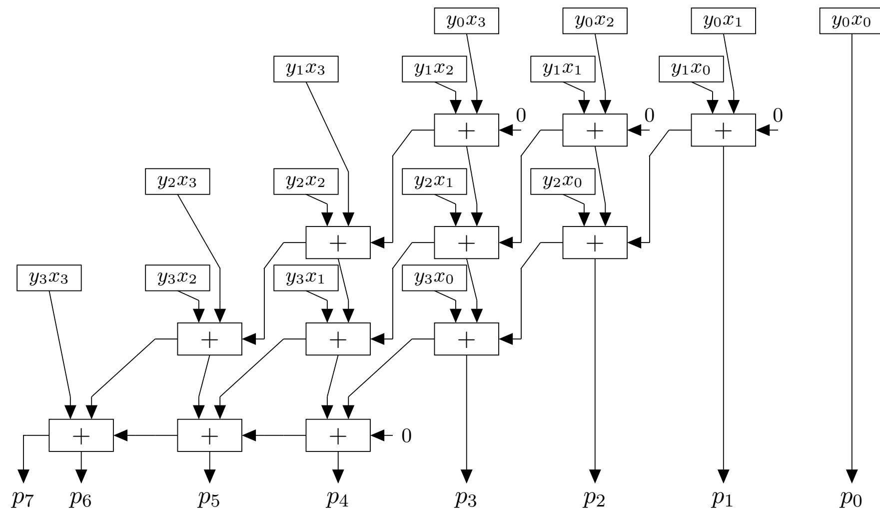
Obrázek 2.5: Propojení čtyřbitové násobičky.

2.2.2 Odčítačka

Binární odčítání ve dvojkovém doplňku je analogické k binárnímu sčítání. Odčítačky se realizují podobně jako sčítačky s tím rozdílem, že vstupní hodnota X (anebo Y) je binárně invertovaná a na vstupu c_0 je hodnota nastavena v logické 1 [24].

2.2.3 Násobička

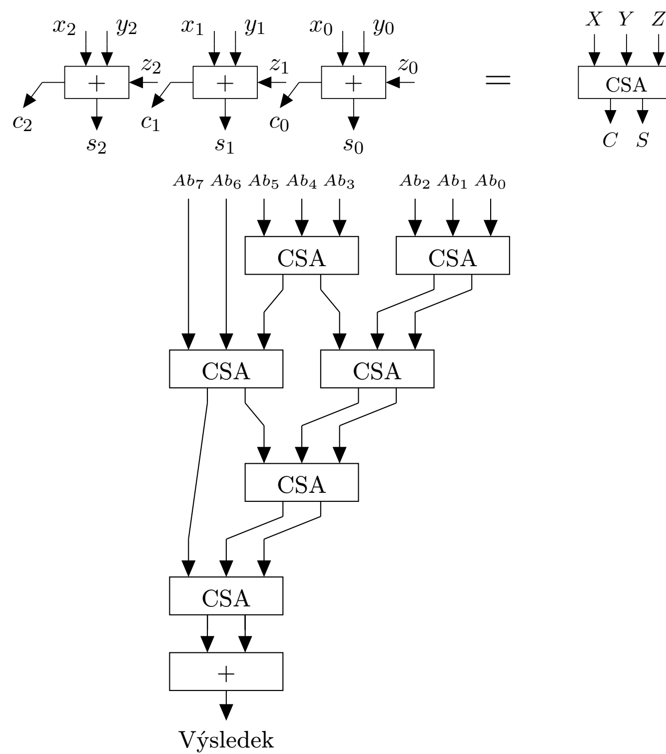
Jedním z nejprimitivnějších způsobů, jak navrhnout násobičku, je použít algoritmus násobení, který se učí na základních školách (násobení pod sebou; posuň a sečti). Jeho princip je postaven na tvorbě částečných součinů. K sestrojení takové násobičky potřebujeme sčítačky a logická hradla AND, propojení provedeme podle obrázku 2.5, kde $p_7p_6 \dots p_0$ je výsledný součin. Hlavní nevýhodou této násobičky je velké zpoždění. Kritická cesta je průchod mezi vstupním uzlem y_0x_1 respektive y_1x_0 po uzel s výstupem p_7. Má-li hradlo AND a poloviční sčítačka zpoždění 1Δ a úplná sčítačka zpoždění 2Δ, potom zpoždění násobičky je 16Δ. Toto zpoždění se s rostoucí délkou operandů dále zvětšuje.



Obrázek 2.6: Optimalizované propojení čtyřbitové násobičky.

Zpoždění můžeme redukovat pomocí tzv. uchovávání přenosu [24]. V řádcích se sčítá bez uvážení přenosů. Avšak přenosy se přičítají v dalším částečném součinu v následujícím kroku (opět bez přenosu). Tyto sčítačky nazýváme sčítačkami s uchováním přenosu. Přičtení posledního operandu se provede pomocí sčítačky s postupným přenosem. Příklad zapojení je uveden na obrázku 2.6. Celkové zpoždění této násobičky je 11Δ.

Podle předchozí metody se provede sekvenčně sedm sčítání pomocí sčítaček s uchováním přenosu. Tento počet lze redukovat pomocí tzv. Wallaceova stromu [25]. Díky nim nejen urychlíme výslednou násobičku, ale i snížíme počet operací sčítání. Osmibitovou násobičku využívající Wallaceův strom můžeme sestrojit podle obrázku 2.7. Je zřejmé, že se díky této metodě docílilo paralelizace, kde je potřeba šesti sčítaček. Zároveň bylo dosaženo menší zpoždění.



Obrázek 2.7: Sestavení násobičky pomocí Wallaceova stromu, kde zkratka CSA znamená sčítačka s uchováním přenosu (z angl. Carry Save Adder).

Kapitola 3

Aproximační obvody

V průběhu vývoje vznikly dva hlavní proudy k aproximaci obvodů. První z nich je aproximace založená na modifikaci elektronických parametrů obvodu (např. napájecí napětí, pracovní frekvence). Byly ukázány metody, kdy se sníží vstupní napětí za účelem snížení příkonu výsledné aplikace. Tento přístup byl využit třeba pro SRAM paměti, jak je uvedeno v článku [7]. Další aproximační metodou je funkční aproximace. Jedná se o cílené změny specifikace obvodu za účelem snížení příkonu, latence nebo velikosti. Tato práce se věnuje právě funkcionální aproximaci.

Podle práce [6] můžeme funkční aproximaci definovat jako převod úplné specifikace logické funkce na částečnou. Nechť máme Booleovu funkci o n vstupech ve tvaru $f : B^n \to B$, kde $B = \{0, 1\}$. Booleova specifikace o n vstupech je funkce $g : B^n \to \{0, 1, -\}$, kde $-$ znamená, že hodnota funkce g je pro danou vstupní kombinaci nespecifikována. Říkáme, že g je plně specifikována tehdy, když každá vstupní kombinace nabývá hodnot právě 0 nebo 1. Specifikace je neúplná, obsahuje-li alespoň jednu hodnotu $-$. Takže Booleova funkce f je plně specifikována Booleova specifikace g. Jinými slovy, vstupní prostor $B^n = \{0, 1\}^n$ specifikační funkce g lze rozdělit do tří částí (tříd). První částí je množina vstupních kombinací, pro které je výstupem 1. Druhou částí je množina vstupních kombinací, pro které je výstupem 0. Třetí částí je množina vstupních kombinací, pro které je výstupem $-$. Podle těchto množin můžeme sestrojit charakteristické funkce g_{on}, g_{off} a g_{dc}. Funkce $g_{dc} : B^n \to \{0, 1\}$ má na výstupu 1 právě tehdy, když g má na výstupu $-$. Funkci g_{on} respektive g_{off} definujeme analogicky podle první respektive druhé množiny. Funkční aproximací rozumíme převod γ úplné specifikace g na částečnou specifikaci $\overline{g}$. Po aplikování aproximace se snažíme zjistit kvalitu aproximace, kterou můžeme definovat třeba jako počet případů, pro které výstupy $\overline{g}$ dává úplná specifikace g rozdílné výstupy. Zároveň se snažíme, aby počet Booleových operátorů funkce získané ze specifikace $\overline{g}$ byl menší než počet Booleových operátorů funkce f. Aproximační obvod pak získáme implementací aproximované Booleovy funkce vytvořené podle specifikace $\overline{g}$ pomocí zvolené konvenční návrhové metody.

V podkapitole 3.1 jsou uvedeny metriky pro měření chyb, bez kterých bychom se v této práci neobešli. V další podkapitole popisuji konvenční návrhy aproximačních obvodů.

3.1 Metriky pro stanovení chyby aproximačních obvodů

Metriky pro stanovení chyby aproximačních obvodů jsou nejdůležitějšími metrikami v aproximačním počítání. Porovnávají nám funkcionalitu aproximačního řešení *approx* vzhledem k plně funkčního řešení *reference*, i-tý výstupní vektor je pak označován jako $approx_i$ respek-

tive $reference_i$. Počet primárních vstupů obvodu je definován jako n_i a počet primárních výstupů jako n_o.

První základní metrikou je počet vstupních kombinací, pro které pracuje obvod špatně:

$$err_{sum} = \sum_{i=1}^{2^{n_i}} \begin{cases} 1 & reference_i \neq approx_i \\ 0 & reference_i = approx_i \end{cases} \tag{3.1}$$

Podobnou přesnější metrikou je počet špatně určených bitů err_{bits}, kde j-tý bit vektoru $approx_i$ resp. $reference_i$ je označen jako $approx_{i,j}$ resp. $reference_{i,j}$. Tato metrika je vhodná pro návrh nearitmetických aproximačních obvodů

$$err_{bits} = err_{shd} = \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} reference_{i,j} \oplus approx_{i,j}. \tag{3.2}$$

Pro aproximační aritmetické obvody můžeme použít vhodnější metriku, která je založena na sumě odchylek od plně funkčního řešení. Jedná se o sumu absolutních hodnot chybných odezev aproximačního řešení od referenčního řešení. Definuje se tedy vztahem

$$err_{sad} = \sum_{i=1}^{2^{n_i}} |reference_i - approx_i|. \tag{3.3}$$

Průměrná chyba je definovaná jako suma odchylek mezi aproximačním a referenčním obvodem. Suma je průměrována počtem vstupních kombinací. Nevýhodou je, že průměrná hodnota je citlivá na krajní hodnoty, které mohou nabývat extrémů. Tento problém může být vyřešen použitím mediánu.

$$err_{avg} = \frac{err_{sad}}{2^{n_i}} = \frac{1}{2^{n_i}} \sum_{i=1}^{2^{n_i}} |reference_i - approx_i| \tag{3.4}$$

Směrodatná odchylka a rozptyl určují, jak jsou hodnoty rozptýleny či odchýleny od průměru hodnot. Rozptyl udává průměr druhých mocnin vzdáleností od průměru. Odchylkou se pak rozumí druhá mocnina rozptylu a vypočte se tedy podle vztahu

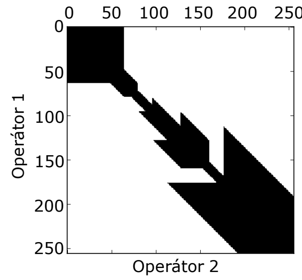
$$err_{\sigma} = \sqrt{\frac{1}{2^{n_i}} \sum_{i=1}^{2^{n_i}} (reference_i - err_{avg})^2} \tag{3.5}$$

Další metrikou je maximální chyba aproximačního řešení definována vztahem 3.6. Jedná se o nejhorší případ, kdy obvod pracuje s chybou. Rovněž se může určit minimální chyba, pro které obvod pracuje s chybou. Viz vztah 3.7.

$$err_{max} = \max_{\forall i \in \langle 1 \dots 2^{n_i} \rangle} |reference_i - approx_i|. \tag{3.6}$$

$$err_{min} = \min_{\forall i \in \langle 1 \dots 2^{n_i} \rangle} \begin{cases} |reference_i - approx_i| & reference_i \neq approx_i \\ undef & reference_i = approx_i \end{cases} \tag{3.7}$$

Výše zmíněné metriky, například err_{max} nebo err_{avg}, mohou být převedeny na procentuální hodnoty. Je zvykem, že se chyba navržených aritmetických aproximačních obvodů



Obrázek 3.1: Příklad grafu rozptylu chyb. Černá barva značí, že pro danou kombinaci nastane chyba. Bílá barva zobrazuje korektní funkčnost.

vyjadřuje právě pomocí těchto hodnot. Ty lze získat tak, že danou metriku podělíme nejvyšší výstupní hodnotou referenčního obvodu:

$$err_{max\%} = \frac{err_{max}}{2^{n_o} - 1} \tag{3.8}$$

$$err_{avg\%} = \frac{err_{avg}}{2^{n_o} - 1} \tag{3.9}$$

$$err_{sum\%} = \frac{err_{sum}}{2^{n_o} - 1} \tag{3.10}$$

Relativní chyba nejlépe zahrnuje informaci o velikosti řádu chyby. Ta se získá takovým způsobem, že se chyba podělí referenční hodnotou

$$err_{relative} = \frac{|reference_i - approx_i|}{reference_i}. \tag{3.11}$$

Všimněme si, že relativní chyba oproti procentuálních hodnot může nabývat hodnot vyšších než 1.

Další metriky mohou zahrnovat různé statistické údaje ku příkladu zjištění mediánu nebo určení prvního či třetího kvartilu chyb. Tyto údaje spolu s minimální a maximální chybou můžeme prezentovat pomocí krabicových grafů. Jiným statistickým údajem je tzv. modus, který odpovídá statisticky nejčastější chybě.

Chyby složitějších aproximačních obvodů můžeme vyjádřit do grafu. Na obrázku 3.1 je zobrazen graf rozptylu chyb, který vychází z práce [14]. Jedná se o grafické vynesení kombinací operandů, pro které obvod chybně pracuje.

3.2 Konvenční návrh aproximačních aritmetických obvodů

3.2.1 Přiřazení konstanty na výstup

Jednou ze základních aproximací je, že se přiřadí na výstup obvodu konstanta [15]. Nevýhodou se jeví chybovost, ale ta může být v některých případech minimální.

Nejvhodnější je zvolit za konstantu nejpravděpodobnější hodnotu. Nechť máme proud dat do systému, který je zobrazitelný Gaussovským rozložením. Potom nejpravděpodobnější hodnota odpovídá střední hodnotě. Velikost chyby přitom záleží na odchylce od střední hodnoty. Mění-li se Gaussovské rozložení v čase, můžeme tuto metodu vylepšit tím, že přepočteme v pravidelných intervalech onu střední hodnotu.

ab \ cd	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	0	1
10	0	1	1	0

ab \ cd	00	01	11	10
00	0	0	0	0
01	0	0	1	1
11	0	1	-	1
10	0	1	1	0

Tabulka 3.1: Aproximace logických funkcí pomocí Karnaughovy mapy. Vlevo je plně specifikovaná funkce, vpravo je zobrazena tatáž funkce, ale aproximovaná. Barevně jsou vyznačeny jednotlivé implikanty.

Mějme aritmetický obvod realizující operaci $\circ$, kterou chceme aplikovat na dvě hodnoty o shodné bitové šířce w. Potom výstupní konstantou bude střední hodnota všech výsledků dané operace. Vzorec je definován podle vztahu

$$avg = \frac{\sum_{i=1}^{2^w} \sum_{j=1}^{2^w} i \circ j}{2^{w+1}}, \tag{3.12}$$

kde ve jmenovateli je počet všech vstupních kombinací a v čitateli suma výsledků operace pro dané kombinace.

Stejným způsobem může být realizována hodnota podle mediánu. Je zřejmé, že u sčítání dvou tříbitových čísel je hodnota mediánu shodná se střední hodnotou. Ovšem pokud se zaměříme na násobení tříbitových čísel, potom je střední hodnota $avg = 12$ odlišná od mediánu $median = 8$.

Další možným způsobem, jak přiřadit číslo na výstup, je zkopírovat jeho vstup. To může být vhodné při sčítání velkých čísel s malými. Uplatní se to například při sčítání 32 bitových čísel a 2 bitových čísel [15].

3.2.2 Zanedbání nejméně významných bitů

Když se provádí operace s velkými čísly a dojde k chybě na nejméně významných číslicích, potom je relativní chyba velmi malá. Toho se využívá v práci s čísly v plovoucí řádové čárce. V případě celočíselné aritmetiky můžeme nejméně významné bity (LSB) zanedbat a použít menší sčítačky či násobičky, které jsou napojené na nejvíce významné bity (MSB) [15].

3.2.3 Karnaughovy mapy v aproximaci

Karnaughovy mapy slouží k minimalizaci logických funkcí. Tato technika transformuje pravdivostní tabulku do dvourozměrného prostoru, který nám dovoluje lépe identifikovat booleovskou sousednost. Sousední buňky se liší pouze v jednom bitu a buňky mohou být různě pootočené – nejedná se o tabulku, ale o torus. Souřadnice jsou proto reprezentovány Grayovým kódem a ne binárním.

Minimalizace probíhá tak, že se sestrojí Karnaughova mapa podle pravdivostní tabulky. Potom se vyhledají na mapě oblasti, z nichž se sepíší součinové výrazy. Jejich suma definuje

ab \ cd	00	01	11	10
00	000	000	000	000
01	000	001	011	010
11	000	011	111	110
10	000	010	110	100

Tabulka 3.2: Upravená Karnaughova mapa pro přibližnou násobičku 2x2 bity. Barvou je označena modifikovaná hodnota. Podle článku [11].

Počet bitů	err_{rate}	$err_{avg\%}$	$err_{max\%}$
2	0.0625	1.39%	22.22%
4	0.19	2.60%	22.22%
8	0.46	3.25%	22.22%
12	0.675	3.31%	22.22%
16	0.81	3.32%	22.22%

Tabulka 3.3: Chybovosti složených přibližných násobiček. Převzato z článku [11].

logickou funkci s operacemi logický součin, logický součet a negaci. Oblasti se tvoří tak, aby byly co největší a bylo jich co nejméně. Zároveň oblasti musí splnit podmínku, aby obsáhly všechny jedničky a přitom neobsahovaly žádnou nulu (či naopak). Při aproximaci obvodů tuto podmínku můžeme porušit, vede-li řešení k méně použitým operacím [18]. Příklad aproximace je zobrazen v tabulkách 3.1. Výsledkem je logická funkce v konjunktivní (či disjunktní) normální formě.

3.2.4 Aritmetické aproximační obvody jako stavební bloky

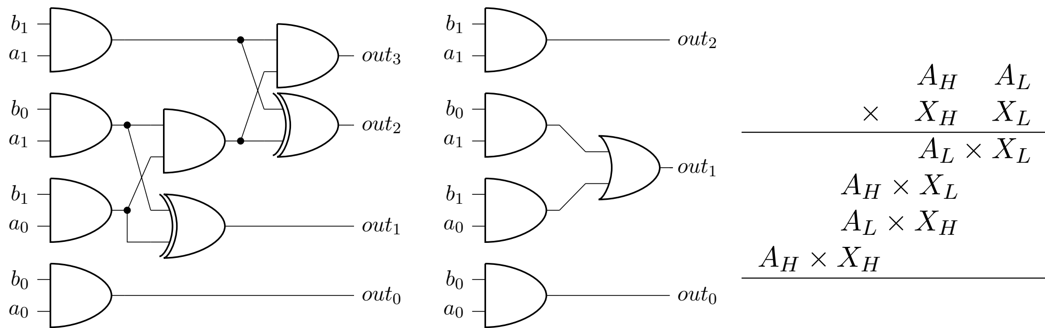
Tato metoda vychází z použití stavebních bloků, např. úplné binární sčítačky, viz vztahy 2.8 a 2.9. Pokud použijeme aproximační verzi přenosu, dle vztahu

$$c_{approx} = (x \oplus y)c + xy, \tag{3.13}$$

docílíme menší plochy, protože výraz $x \oplus y$ už máme vypočtený.

K chybě dochází v jednom případě z šestnácti a to při přenosu $err_{rate} = \frac{1}{16}$ a maximální chyba je $err_{max} = 1$. Tyto aproximační sčítačky můžeme propojit, jak je uvedeno na obrázku 2.2. S propojením přichází propagace chyby, která závisí na počtu aproximačních a úplných sčítaček ve výsledném obvodu. Tato metoda vychází z článku [26]. Výhodou aproximační sčítačky je, že obsahuje o jedno hradlo AND méně a nevyužívá trojvstupový OR, ale dvojvstupový. Zpoždění aproximačního prvku je shodné s přesnou verzí úplné sčítačky.

V článku [11] byla představena metoda návrhu násobiček pomocí menších násobiček a sčítaček. Nejdříve se sestrojí aproximační násobička podle Karnaughovy mapy, jak je zobrazeno v tabulce 3.2. Klasické násobení čísel $3 * 3$ má na výstupu čtyři bity (čili nabývá hodnoty 1001), aproximační násobičce můžeme přidělit pouze tři bity (tedy hodnotu 111). Výsledná chyba se objevuje pouze v jednom případu ze šestnácti ($err_{rate} = \frac{1}{16}$) a její velikost je $err_{max} = 2$. Aproximační obvod pak obsahuje výrazně méně prvků, jak je vidět na obrázku 3.2. Jeho velikost se zmenšila o 30% a jeho kritická cesta se rovněž zmenšila. Naprostou výhodou tohoto aproximačního prvku je možnost zotavení se z chyby. Víme, že chyba nastane pouze u násobení $3 * 3$ a má velikost 2. Proto můžeme vytvořit další obvod, který přičte 2 k výsledku přibližné násobičky, když oba operandy nabývají hodnoty 3.



Obrázek 3.2: Přesná (vlevo) a přibližná (uprostřed) násobička 2x2 bity. Vpravo je uvedena tvorba větších násobiček z menších. Převzato z [11].

Vytvořená přibližná násobička je použita jako stavební blok pro větší násobičky. Na obrázku 3.2 je rovněž ukázáno sestrojení aproximační vícebitové násobičky ze čtyř přibližných menších násobiček a sčítaček. Počet chyb err_{rate} pak roste s velikostí násobičky, ale střední chyba $err_{avg\%}$ roste velice pomalu. Tento fakt je zaznamenán v tabulce 3.3.

3.2.5 SALSA

Jedním z algoritmů pro konvenční návrh aproximačních obvodů je SALSA (Systematic Logic Synthesis of Approximation Circuits). Jedná se o konvenční iterativní metodu, která podle plně funkčního obvodu a zadané chybovosti navrhuje aproximační obvod. Algoritmus pracuje následovně:

- Pro každý primární výstup obvodu se určí množina hradel, pro které je výstup funkce kvality neaktivní.
- Dle těchto hradel se obvod zjednoduší, aproximuje.
- Upraví se obvod podle aproximačního obvodu a celý tento proces se iteruje přes všechny primární výstupy.

SALSA rovněž obsahuje různé akcelerační a heuristické techniky, například zkoumá různé vstup-výstupní závisloti [23].

3.2.6 ABACUS

Pro syntézu aproximačních obvodů na behaviorální úrovni byla představena metoda ABACUS (A Technique for Automated Behavioral Synthesis of Approximate Computing Circuits) [15].

Jedná se o algoritmus, který využívá tzv. hladové prohledávání. Každý složitější kombinační aproximační obvod může být reprezentován abstraktním syntaktickým stromem, zkráceně AST z angl. Abstraction Syntax Tree. V každé iteraci algoritmu greedy search transformuje AST podle různých transformačních operátorů. Mezi transformační operátory patří zjednodušení datových typů, kdy se provádí operace pouze na nejvíce významných bitech, nebo nahrazení proměnné konstantou. Další možností je samotná transformace plně funkčních aritmetických obvodů na jednodušší verze např. sčítání se provede pomocí logické operace OR nebo násobení se zamění za bitový posuv. Na úrovni syntaktického stromu má

tento algoritmus transformaci pro podstromy obsahující proměnnou s podobnými hodnotami např. výraz $y_i * x_i + y_j * x_j$ se může zamění za $(y_i + y_j) * x_i$.

Podle stochastického algoritmu se pak vytvoří množina možných aproximačních obvodů. Pro každý nalezený obvod se musí vypočítat jeho fitness hodnota, která odpovídá vzorci

$$fitness = w_1 * accuracy + w_2 * power + w_3 * area, \tag{3.14}$$

kde pro váhy platí vztah $w_1 > w_2 > w_3$. Přesnost výsledného obvodu *accuracy* je simulována na trénovacích množinách. Prohledávání je ukončeno pokud byl nalezen obvod s vyšší chybovostí, nebo byl vyčerpán počet iterací algoritmu.

Kapitola 4

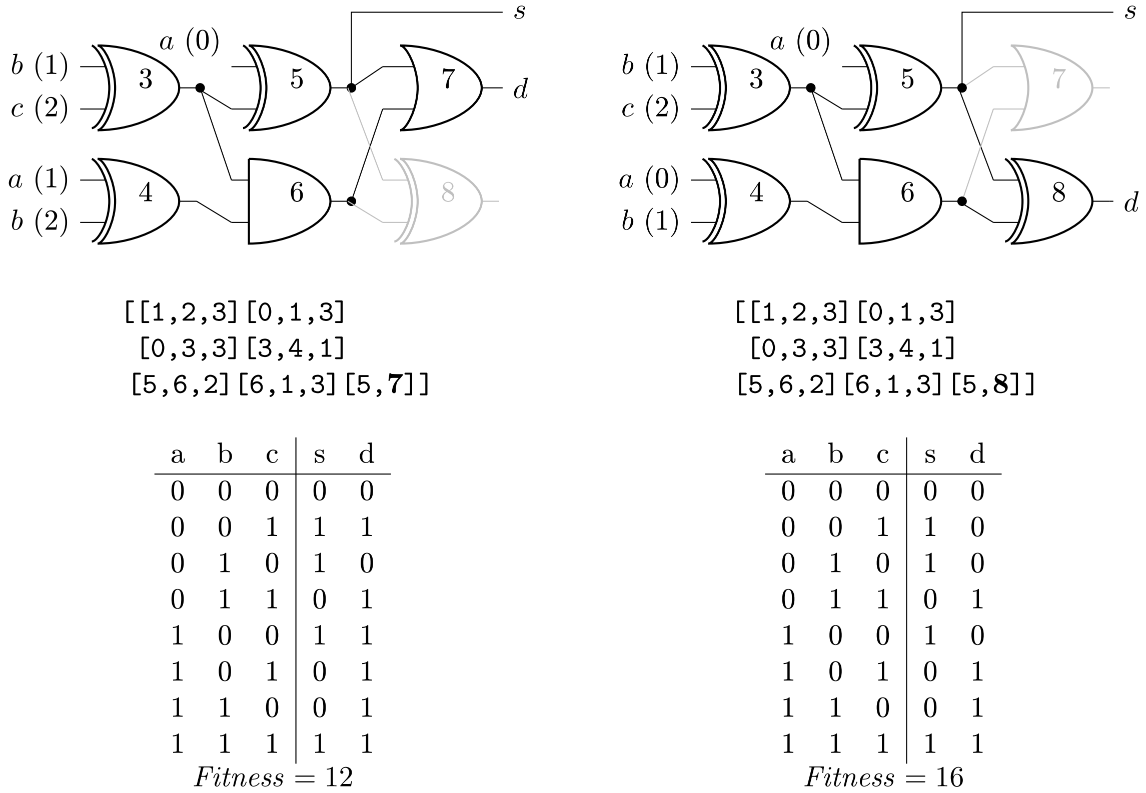
Evoluční návrh pomocí kartézského genetického programování

Kartézské genetické programování, zkráceně CGP (z angl. cartesian genetic programming), bylo představeno Julianem Millerem v roce 1999 pro návrh obvodů [13]. Jedná se o algoritmus spadající do třídy evolučních algoritmů, které vycházejí z Darwinovy teorie evoluce a různých teorií neodarwinismu. Hlavní hnací silou evoluce je přírodní výběr, jinými slovy selekce. Nadprůměrně kvalitní jedinci mají více potomků, a proto se jejich genetická informace objevuje v dalších generacích častěji než u méně kvalitních jedinců. Postupem času získáváme jedince, jež jsou dokonalejší než jedinci na začátku evoluce. Evoluční algoritmy pracují analogicky.

Jedinec v CGP je nejčastěji reprezentován acyklickým orientovaným grafem. Je možné použít i obecné orientované grafy, např. pro návrh sekvenčních obvodů. Uzly jsou uspořádány do matice o n_r řádcích a n_c sloupcích. Každý uzel představuje určitou operaci z konečné množiny operací Γ. V některých implementacích je nutno zjistit maximální aritu n_a operace f takové, že $f \in \Gamma$. Tato arita udává počet vstupů uzlu. V návrhu logických obvodů uzly představují dvouvstupá hradla s binární aritou, je však možno použít i vícevstupá hradla s vyšší aritou [17]. Dále je nutné definovat parametr l-back l, což je míra propojitelnosti uzlů mezi jednotlivými sloupci grafu. Například pro $l = 1$ je propojitelnost minimální, neboť se propojují uzly mezi sousedními sloupci. Pro $l = n_c$ je pak propojitelnost maximální, protože se mohou propojit libovolné sloupce. Dále musí být určen počet primárních vstupů n_i a počet primárních výstupů n_o grafu. Tyto hodnoty reprezentují počet vstupů a výstupů výsledného obvodu.

Zapojení obvodu je zakódováno chromozomem konstantní délky. Každý uzel grafu je kódován k-ticí $(i_1, i_2, \ldots, i_{n_a}, \alpha)$, kde α je kód funkce z Γ. Hodnoty i_1, i_2 až i_{n_a} jsou indexy libovolného uzlu předchozích sloupců nebo se jedná o zapojení na primární vstup výsledného obvodu. Chromozom dále obsahuje n_o genů, které označují uzly nebo primární vstupy připojené na primární výstupy. Příklad takového zakódovaní je znázorněn na obrázku 4.1. Rovněž můžeme vidět, že kódování je redundantní a zároveň některé primární vstupy nemusejí být použity.

Evoluce pak hledá takové propojení grafu, aby odpovídalo zadané specifikaci. Na počátku je vytvořena počáteční populace buďto náhodně, nebo s využitím nějaké heuristiky např. plně funkčního obvodu. V každé generaci je vybrán rodič, což je jedinec s nejlepší fitness hodnotou. Mutací rodiče se vytvoří potomci, kteří spolu s rodičem tvoří základ nové populace. Tato varianta generování populace je pak nazývána jako evoluční strategie $(1 + \lambda)$,



a	b	c	s	d
0	0	0	0	0
0	0	1	1	1
0	1	0	1	0
0	1	1	0	1
1	0	0	1	1
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$Fitness = 12$

a	b	c	s	d
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

$Fitness = 16$

Obrázek 4.1: Zakódovaní grafu v CGP, chromozóm a fitness před (zleva) a po mutaci.

kde λ udává počet potomků. Je-li v populaci více jedinců s nejvyšší fitness hodnotou, vybere se za rodiče ten, který nebyl v předchozí generaci rodičem. Evoluce končí nalezením dostatečně kvalitního jedince nebo vyčerpáním počtu generací n_e.

Mutací se náhodně změní propojení uzlů, či jejich operace. Parametr h udává počet genů v chromozomu, u kterých dojde ke změně. Mutace je neutrální tehdy, když nemá vliv na hodnotu fitness daného jedince. Tato situace nastává ve dvou případech. Buď aplikujeme mutaci na neaktivní uzel, který nemá vliv na výsledné řešení (fenotyp). Nebo aplikací mutace přejdeme k jinému fenotypu, jenž má fitness hodnotu shodnou s rodičovskou. Opakem neutrálních mutací jsou adaptivní mutace.

Kartézské genetické programování bylo úspěšně použito kromě návrhu kombinačních obvodů také pro návrh obrazových filtrů, klasifikaci, či tvorbu jednoduchých programů [17].

4.1 Vícekriteriální optimalizace

Návrh aproximačních obvodů může být formulován jako vícekriteriální optimalizační problém, ve kterém existuje několik vzájemně konfliktních kritérií (chybovost, plocha, zpoždění a příkon). U vícekriteriální optimalizace neexistuje jedno nejlepší řešení, které by pokrylo všechny potřeby. Představme si člověka, který navrhuje obvody. Bude jej navrhovat na základě zpoždění, plochy či chyby? Situace, kdy se člověk bude rozhoduje podle jednoho parametru, nastávají velice zřídka.

Sestrojíme tzv. Paretovu množinu, která bude zohledňovat všechny kritéria. Říkáme, že řešení x Pareto dominuje řešení y, když $\forall i \in N : f_i(x) \leq f_i(y) \land \exists j \in N : f_i(x) < f_i(y)$,

kde f_i odpovídá danému kritériu a $N = \{1 \dots m\}$ je množina kritérií $|N| = m$ a je cílem minimalizovat f_i. Nechť P je množina řešení, potom $x \in P$ nazýváme Pareto optimální právě tehdy, když neexistuje žádné řešení $y \in P$ takové, že y Pareto dominuje x a zároveň řešení x náleží do Paretovy množiny Φ $(x \in \Phi)$.

4.2 Fitness funkce pro aproximační obvody

Fitness funkce slouží k ohodnocení kandidátního řešení. Každému jedinci se přiřadí tzv. fitness hodnota, která udává kvalitu nalezeného řešení. V evoluci se pak vybírá jedinec s nejlepší hodnotou fitness, čili řešení buď s minimálním, nebo maximálním ohodnocením.

Pro tvorbu fitness funkce plně funkčních logických obvodů potřebujeme mít jasně definovanou specifikaci obvodu. Pro každý vstupní vektor, kterých je celkem 2^{n_i}, je přiřazen výstupní vektor o velikosti n_o. Pro každou vstupní kombinaci je definována hodnota na výstupu. Celkově se tedy specifikuje $n_o * 2^{n_i}$ výstupních kombinací (v bitech).

Fitness hodnota určitého kandidátního obvodu C_{obt} v zásadě odpovídá počtu chybných odezev od referenčního obvodu C_{ref}. Pro každý výstupní vektor obt_i kandidátního řešení se zjistí Hammingova vzdálenost s referenčním vektorem ref_i. Hammingova vzdálenost dvou binárních řetězců shodné délky je počet pozic, ve kterých se tyto řetězce liší. Hodnota fitness *fit* odpovídá sumě Hammingových vzdáleností všech výstupních kombinací. Tato fitness funkce může být implementována podle vztahu

$$fit = \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} ref_{i,j} \oplus obt_{i,j}, \tag{4.1}$$

který je ve své podstatě ekvivalentní se vztahem 3.2.

Existují však i jiné možnosti, jak implementovat fitness funkci pro návrh aproximačních obvodů. V této podkapitole jsou uvedeny popisy fitness funkcí a jejich experimentální vyhodnocení.

4.2.1 Suma Hammingových vzdáleností

Jednou z primitivních možností tvorby aproximačních obvodů je postupné zmenšení vyhledávacího prostoru. Graf CGP je pak definován $n_c \in \{n_{g_AC} \mid 0 < n_{g_AC} < n_{g_PC}\}$, $n_r = 1$, $l = n_c$, kde n_{g_PC} je počet hradel plně funkčního obvodu [16]. Tato metoda však selhává, chceme-li přidat více řádků n_r.

Důmyslnější metodou je proto omezit fitness funkci na maximální počet použitých hradel aproximačního obvodu n_{g_AC}, takový, že $0 < n_{g_AC} < n_{g_PC}$ podle vztahu

$$fit_{SHD} = \begin{cases} \sum\limits_{i=1}^{2^{n_i}} \sum\limits_{j=1}^{n_o} ref_{i,j} \oplus obt_{i,j} & n_g \leq n_{g_AC} \\ \infty & \text{jinak} \end{cases}. \tag{4.2}$$

Je-li celkový počet uzlů n_g kandidátního řešení větší než požadovaný počet n_{g_AC}, potom nastavíme jeho fitness tak, aby nebyl vybrán rodičem.

4.2.2 Vážená suma Hammingových vzdáleností

Předchozí metoda není vhodná pro návrh obvodů realizujících aritmetické operace. Může se totiž nalézt obvod, který sice ve většině případů poskytne správný výsledek, ovšem v případě chyby je jeho funkcionalita podprůměrná. Jinými slovy obvod má malý počet chybně

určených bitů err_{bits} (vzorec 3.2), ovšem jeho celková chybovost err_{sad} (vzorec 3.3) je daleko větší. Předchozí metoda (podkapitola 4.2.1) by vybrala obvod s vyšší chybovostí err_{sad}.

Zavedeme takzvanou prioritu výstupů. Principem je, že se nastaví váhy pro jednotlivé hodnoty výstupů. V praxi to znamená, že nejvyšší váhu má MSB. Takže je větší pravděpodobnost, že se na LSB bude nacházet chyba častěji než na MSB. Aplikujeme vztah

$$fit_{weighted\ SHD} = \begin{cases} \sum_{i=1}^{2^{n_i}} \sum_{j=1}^{n_o} w_j * (ref_{i,j} \oplus obt_{i,j}) & n_g \leq n_{g_AC} \\ \infty & \text{jinak} \end{cases}, \tag{4.3}$$

kde obdržený výstup násobíme váhou w_j. V případě implementace je z časového hlediska vhodnější bitový posuv obdržené chyby než násobení konstantou.

4.2.3 Suma absolutních diferencí

Pro tvorbu aproximačních obvodů může být vhodnější využít fitness funkci, jež je používána při řešení problému symbolické regrese. Problém spočívá v nalezení matematické funkce, která nejlépe aproximuje referenční data s požadovanou přesností. Fitness hodnota v případě logických obvodů odpovídá sumě chyb kandidátního řešení. Tuto hodnotu můžeme získat podle vzorce

$$fit_{SAD} = \begin{cases} \sum_{i=1}^{2^{n_i}} |ref_i - obt_i| & n_g \leq n_{g_AC} \\ \infty & \text{jinak} \end{cases}, \tag{4.4}$$

kde referenční ref_i a kandidátní obt_i vektory chápeme jako celočíselné hodnoty. Tento princip byl uveden v článku [16].

4.3 Evoluční aproximace komplexnějších aritmetických obvodů

Tato podkapitola popisuje různé typy metod CGP pro návrh komplexnějších aritmetických obvodů, které jsou vhodné pro aproximační počítání. Problémem CGP je totiž ve škálovatelnosti jeho použití. Nejen doba ohodnocení kandidátního obvodu roste s počtem jeho vstupů, ale i počet generací nutných k nalezení řešení.

Existuje několik studií, které se zabývají právě tímto problémem a pokouší se jej řešit. Publikace [20] inicializuje CGP plně funkční násobičkou, kterou potom na základě různých fitness funkcí aproximují. Tento přístup je podrobněji popsán v podkapitole 4.3.1. Článek [21] vychází rovněž z inicializace plně funkčním obvodem. Postupně se dekrementuje počet hradel obvodu s cílem získat Paretovu množinu. Na tuto metodu se zaměřuje podkapitola 4.3.2.

4.3.1 Evoluční aproximace obvodů podle různých chybovostních metrik

CGP inicializujeme plně funkčním aritmetickým obvodem např. násobičkou vizte kapitolu 2.2.3. Designér určí velikost chyby K, podle které budeme referenční obvod aproximovat. Samotný běh CGP rozdělíme do dvou fází. V první fázi se snažíme pomocí evoluce vytvořit násobičku, jejíž fitness hodnota f_1 se blíží k hodnotě K. Fitness hodnotu f_1 implementujeme podle metrik pro stanovení chyby uvedených v kapitole 3.1. Vhodná je například

maximální chyba err_{max}, střední chyba err_{avg}, či relativní chyba $err_{relative}$. Druhá fáze začne tehdy, když získáme obvod s velikostí chyby K anebo hodnotě blízké tedy $K \pm \varepsilon$. V této části se snažíme o optimalizaci podle druhého kritéria, což může být třeba počet hradel, velikost zpoždění a nebo i jiná metrika pro stanovení chyby. Zahrnutí fitness hodnoty f_1 do výpočtu f_2 zaručuje, že se první optimalizační kritérium nezhorší. Nabízí se nám několik možností, jak přidělit fitness funkce pro f_1 a f_2. V původním článku [20] vytvořili tři možné scénáře.

V prvním scénáři aproximovali plně funkční násobičku na hodnotu K, která reprezentovala nejvyšší možnou chybu err_{max}, pod podmínkou, aby průměrná chyba err_{avg} byla co nejmenší. Fitness hodnota f_1 odpovídala výpočtu podle vztahu 3.6 a druhá fitness hodnota f_2 byla vypočtena podle vztahu 3.4.

V druhém scénáři se vytvářely obvody s co nejmenší plochou, kde hodnota K znamenala nejvyšší možnou chybu err_{max}. Takže fitness hodnota f_1 odpovídala výsledku vztahu 3.6 a druhá fitness hodnota f_2 znamenala počet použitých hradel.

Třetí scénář byl obdobný druhému. Hodnota K reprezentovala průměrnou chybu err_{avg}. Fitness hodnota f_1 byla vypočtena podle vztahu 3.4. Druhá fitness hodnota f_2 znamenala nejmenší počet hradel.

Bylo představeno, že pro osmibitovou násobičku s maximální chybou $err_{max\%} = 10$ lze získat až 96% snížení příkonu [20].

4.3.2 Evoluční návrh aproximačních obvodů pomocí heuristické inicializace

Článek [21] uvádí metodu heuristického inicializace pro aproximaci obvodů. Představený algoritmus CGP se mírně liší od algoritmu zmíněném v úvodu této kapitoly. Primární výstupy grafu CGP mohou být napojeny na logické konstanty 0 a 1. Tato drobná změna může být rozhodující pro aproximační obvody, zejména pro obvody s malým počtem hradel.

Heuristická inicializace probíhá následovně. Mějme plně specifikovaný obvod C_{ref}, který má n_g hradel a chceme navrhnout aproximační obvod s $n_g - 1$ hradly. Vytvoříme $2n_g$ obvodů takových, že každé hradlo v obvodu C_{ref} nahradíme propojem, který spojuje výstup hradla s prvním resp. druhým vstupem. Vznikne tak množina obvodů, které mají o jedno hradlo méně než původní obvod. Tyto obvody ohodnotíme fitness funkcí, která představuje hodnotu err_{sad}, vizte vztah 3.3. Běh CGP je pak inicializován nejvhodnějším obvodem, čili obvodem s nejmenší chybou.

Pro obvody s větším počtem hradel může být tato inicializace časově náročná. Z původního obvodu lze vytvořit aproximační obvody s $n_g - k$ hradly, kde k je počet náhodně odebraných hradel a vytvoříme N takových obvodů.

V původním článku autoři vytvořili dva přístupy. První z nich funguje na principu inkrementální evoluce. Z původního plně specifikovaného obvodu vytvoříme podle heuristiky aproximační obvod, jenž má celkem $n_g - 1$ hradel. Tento aproximační obvod optimalizujeme pomocí CGP, z jehož výsledku vytvoříme obvod s $n_g - 2$ hradly. Postupně získáme Pareto množinu aproximačních obvodů s nejlepší nalezenou funkcionalitou pro daný počet hradel. V druhém přístupu se CGP inicializuje s různě velkými obvody, které obsahovaly $n_g - 1, n_g - 2, \ldots, 2, 1$. Což znamená, že tvorba aproximačního obvodu o $n_g - 2$ hradlech nezáležela na obvodu s $n_g - 1$ hradly, ale závisela na původním C_{ref} obvodu.

V článku [21] bylo také ukázáno, že pomocí heuristické inicializace počáteční populace můžeme aproximovat 25 bitový medián o 221 komponentách. Na takto komplexních úlohách evoluční aproximace selhává, použijeme-li náhodnou inicializaci.

Kapitola 5

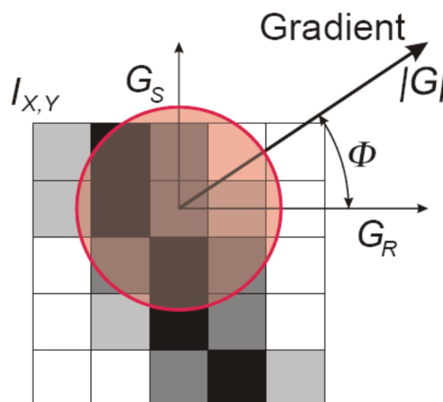
Detekce hran

Detekce hran v obraze je jednou z nejčastějších operací počítačového vidění, se kterou se můžeme setkat. Detekce hran se hojně využívá v algoritmech pro analýzu a segmentaci obrazu, rekonstrukci scény, sledování pohybu objektů, ale také například v medicínských aplikacích. Cílem detekce hran je nalezení pixelů v obraze, kde se výrazně mění jas. Tato změna má konstantní směr gradientu v okolí o určité velikosti, viz obrázek 5.1. Směr gradientu lze určit pomocí první derivace, která je nejvyšší v kolmém směru na hranu [27].

V dnešní době existuje velké množství hranových detektorů. Můžeme využít algoritmy postavené na poznatcích matematické analýzy, neuronových sítí, fuzzy logiky, ba i evolučních algoritmů [17]. Tato kapitola popisuje pouze základní detektory hran, ve kterých můžeme použít aproximační prvky. Podkapitola 5.1 se věnuje detekci pomocí Sobelových operátorů. Další podkapitola 5.2 prezentuje Cannyho hranový detektor.

5.1 Metody založené na první derivaci

Výpočet gradientu v diskrétním obrazu lze provést na základě konvoluce s vhodně zvolenými jádry – operátory. Nejpoužívanější jsou Sobelovy operátory, Robertsovy operátory a operátory Prewittové, viz tabulky 5.1 a 5.2. Každý zmíněný operátor se liší hodnotami konvoluční masky, které udávají jakou vahou se podílejí jednotlivé body na výpočtu gradientu. Dále se liší velikostí konvoluční masky, například Robertsův operátor používá konvoluční masku o velikosti 2x2, Sobelovo jádro a operátor Prewittové mají velikost 3x3.



Obrázek 5.1: Grafické znázornění hrany v obraze a jejího gradientu [27].

1	1	1	1	1	0	1	0	-1	0	-1	-1
0	0	0	1	0	-1	1	0	-1	1	0	-1
-1	-1	-1	0	-1	-1	1	0	-1	1	1	0

Tabulka 5.1: Prewittové operátory, zleva pro hrany pod úhlem 0°, 45°, 90° a 135°.

1	2	1	2	1	0	1	0	-1	0	-1	-2
0	0	0	1	0	-1	2	0	-2	1	0	-1
-1	-2	-1	0	-1	-2	1	0	-1	2	1	0

Tabulka 5.2: Sobelovy operátory, zleva pro hrany pod úhlem 0°, 45°, 90° a 135°.

Velikost gradientu $|G|$, tzv. magnitudu hrany, lze získat konvolucí dvou operátorů, které jsou k sobě kolmé. Vypočte se gradient např. v horizontálním (G_x) a vertikálním směru (G_y). Výsledná hodnota gradientu se dále počítá jako délka vektoru těchto dvou hodnot, viz vzorec 5.1. Jedná se o velikost gradientu v Eukleidovském prostoru, což může být náročné na výpočet. Pro výpočet v hardwaru můžeme použít zjednodušenou verzi založenou na Mannhatonské matrice, viz vzorec 5.2.

$$|G| = \sqrt{G_x^2 + G_y^2} \tag{5.1}$$

$$|G| = |G_x| + |G_y| \tag{5.2}$$

Proces detekce hran můžeme rozdělit na dvě fáze:

1. Zvýraznění oblastí hran, tedy oblastí s vysokou změnou intenzity.
2. Detekce hran pomocí prahování; magnitudy větší než zvolený práh T jsou zvýrazněny.

Příklad hranového detektoru můžeme vidět na obrázku 5.2. Pro výpočet gradientu byly zvoleny Sobelovy operátory pod úhly 0° a 90°. Můžeme vidět, že detekce po prahování neztenčuje hrany. Hlavním nedostatkem je, že hrany jsou nesouvislé, tedy přerušované – objevuje se diskontinuita hrany [27].

Obrázek 5.2: Vlevo původní obrázek. Uprostřed detekce hran pomocí Sobelových operátorů. Vpravo výsledek po prahování, kde $T = 42$.

5.2 Cannyho detektor hran

Cannyho detektor hran byl definován v roce 1986 Johnem Cannym na základě vlastností, které musí splnit detekce hran [4]. V práci Johna Cannyho se tvrdí, že optimální detekce hran musí splňovat určité požadavky na kvalitu, přesnost a jednoznačnost.

Kvalitou detekce se rozumí, že musí být nalezeny všechny hrany v obraze. Zároveň nesmí být detekovány hrany, které jimi nejsou. Přesnost detekce znamená, že nalezené hrany musí být tak blízko skutečné hraně v obraze, jak je to jen možné. Detekovaná hrana je jednoznačná, pokud nalezená hrana v obraze je označena jako hrana pouze jednou. Nesmí docházet ke vzniku vícenásobné odezvy na jednu hranu.

John Canny navrhl algoritmus, které se snaží splnit tyto požadavky. Pro dvourozměrný obraz se implementuje jako algoritmus složený ze čtyř částí:

1. Eliminace šumu
2. Výpočet magnitudy gradientu
3. Výpočet směru gradientu a ztenčení hran
4. Prahování s hysterezí

Výstupy jednotlivých kroků vidíme na obrázku 5.3.

První krok je zásadní, neboť potlačení šumu ovlivňuje výsledek dalších kroků algoritmu. K odstranění šumu se používá konvoluce Gaussovým filtrem, jehož kernel bývá běžně reprezentován maticí 5x5. Jádro lze vypočítat dle vztahu

$$Gauss(x, y) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}}. \tag{5.3}$$

Konvoluční matice však může být větší pro intenzivnější rozmazání zdrojového obrázku. V této práci budeme však používat běžnou velikost Gaussova jádra (tedy jádro o velikosti 5x5). Dále budeme používat celočíselnou metodu implementace, která je vhodnější pro hardware. Upravíme první část gaussovské funkce (část před e), která normalizuje křivku tak, aby její integrál byl roven jedné. Gaussovu funkci můžeme násobit celočíselnou konstantou. Tím nám vznikne jádro, které se hodí ke konvoluci na celočíselné úrovni. Následně můžeme provést normalizaci konvoluovaného obrázku pomocí bitových posuvů nebo podělením maximální hodnotou.

Druhý krok spočívá ve výpočtu magnitudy hrany. Pro tyto účely se používají nejčastěji Sobelovy operátory, viz předchozí podkapitola.

Obrázek 5.3: Jednotlivé kroky Cannyho detektoru hran. Zleva původní obrázek, rozmazaný obrázek, získaná magnituda, ztenčení hran a hysterezovaný obrázek.

Cílem dalšího kroku je ztenčit nalezené hrany podle lokálních maxim. Pro každou magnitudu musí být zařazen směr gradientu, a to do čtyř skupin na úhly 0°, 45°, 90° a 135°. Směr gradientu se obvykle počítá dle vztahu

$$\Theta = \tan^{-1}(G_x/G_y). \tag{5.4}$$

Podle normály tohoto přibližného úhlu se provede ztenčení. Pokud jsou obě sousední magnitudy menší než daná magnituda, jedná se o hranu.

Posledním krokem je dvojité prahování a hystereze. Body obrazu, které mají vyšší magnitudu než zvolený vyšší práh T_H, jsou automaticky řazeny jako hrana. Body obrazu, které mají magnitudu mezi nižším T_L a vyšším prahem T_H, jsou hrany právě tehdy, když jsou napojeny na bod, který byl označen za hranu. Tento krok tedy odstraňuje slabé hrany, které nejsou napojeny na silné.

Kapitola 6

Akcelerace ohodnocení aproximovaného kandidátního řešení

Kartézské genetické programování je časově velmi náročný algoritmus, proto se tato kapitola zaměřuje na jeho urychlení, a to především na akceleraci ohodnocení kandidátního řešení. Využitím profilovacího nástroje `GNU gprof` můžeme zjistit, v jaké funkci stráví program nejvíce času. Pro tento účel byly zadány shodné parametry CGP $n_r = 1$, $n_c = 80$, $l = 40$, $\Gamma = \{\text{BUF, NOT, AND, OR, XOR, NAND, NOR, XNOR}\}$, $\lambda = 4$, $h = 1$, $n_e = 5 \cdot 10^6$ pro tříbitové, čtyřbitové, pětibitové a šestibitové sčítačky. Inicializace populace byla provedena náhodně. Evaluace obvodu `cgp_eval()` využívala paralelní 64-bitovou simulaci, která zahrnovala i nepoužitá hradla. Výpočet fitness hodnoty `calc_fitness()` byl podle metody SHD, která využívala funkci `zeroscount()` implementovanou podle lookup tabulky. Zjištění počtu použitých hradel, `used_nodes()`, se provádělo po skončení výpočtu fitness.

Z výsledků experimentu, viz tabulka 6.1, vyplynulo, že nejvíce času strávil algoritmus simulací kandidátního řešení `cgp_eval()`. V jedné simulaci je nutné ohodnotit 2^{n_i} vstupních vektorů pro všechna hradla celého chromozomu. Takže doba simulace kandidátního obvodu roste exponenciálně s rostoucím počtem jeho vstupů. Tuto vlastnost potvrzují zachycené hodnoty v tabulce.

Sčítačka	3+3 bity 6 vstupů		4+4 bity 8 vstupů		5+5 bity 10 vstupů		6+6 bity 12 vstupů	
`cgp_eval()`	74.10 %	38.7 s	87.63 %	152.0 s	91.01 %	620.2 s	92.24 %	2440.1 s
`zeroscount()`	3.55 %	1.8 s	5.07 %	8.8 s	6.01 %	41.1 s	5.93 %	157.0 s
`calc_fitness()`	1.24 %	0.6 s	1.54 %	2.6 s	1.52 %	10.3 s	1.73 %	45.6 s
`used_nodes()`	18.30 %	9.5 s	4.79 %	8.3 s	1.15 %	7.8 s	0.00 %	0.0 s
`cgp_mutate()`	2.13 %	1.1 s	0.78 %	1.3 s	0.29 %	1.9 s	0.12 %	3.2 s
`main()`	0.78 %	0.4 s	0.30 %	0.5 s	0.10 %	0.6 s	0.08 %	2.1 s

Tabulka 6.1: Procento času strávené v různých funkcí algoritmu CGP pro různě velké sčítačky pomocí nástroje GNU `gprof`.

6.1 Paralelní simulace

Částečné řešení tohoto problému představuje paralelní simulace. Při ní se využije toho, že můžeme do proměnné typu integer uložit 64 bitů. Potom v jednom průchodu obvodem získáme výsledek pro 64 vstupních kombinací. Urychlení je 64 oproti naivní simulaci [17]. Rovněž se může využít SSE jednotka, která je součástí většiny procesorů. Tato jednotka má 128-bitové registry a obsahuje efektivní instrukci *popcnt*, která je vhodná pro zjištění počtu jedničkových bitů v daném registru.

6.2 Simulace fenotypu a přeskočení neutrálních mutací

Další praktikou je, že se nemusí simulovat celý obvod, stačí pouze simulace použitých hradel. Mohou se zanedbat hradla, které nejsou použita ve finálním obvodu. Vyhledání nepoužitých hradel sice přináší jistou režii, avšak v případě aproximace kandidátního řešení je nutno zjistit počet aktivních hradel u každého řešení.

V článku [8] je představen způsob akcelerace pomocí přeskočení neutrálních mutací. Dojde-li pouze k mutaci neaktivních genů, což jsou nepoužité geny ve fenotypu, potom kandidátní řešení nemusíme ohodnocovat. Známe totiž jeho fitness hodnotu a jeho použité uzly. Rovněž jsou v [8] prezentovány upravené operátory mutace, např. mutuje se do doby, než se zmutuje aktivní gen.

6.3 Předkompilace chromozomu a fitness funkce

Obecně je známo, že při zřetězeném zpracování instrukcí jsou problematické instrukce skoku. Při interpretaci funkcionality obvodu se tyto skokové instrukce mohou negativně projevit. Možností akcelerace je tedy využití předkompilace chromozomu [22]. Jedná se o tzv. Just-in-time (JIT) kompilaci. Kompilace chromozomu s instrukcemi skoku probíhá pouze jednou a to před simulací kandidátního obvodu. Aby tato metoda byla efektivní, musí se zkompilovaný chromozóm spustit vícekrát, protože samotná kompilace má také svou režii. Vhodnost JIT kompilace závisí nejen na počtu trénovacích vektorů, ale i na velikosti grafu obvodu.

Předchozí práce [22] však nezahrnuje zkompilování výpočtu fitness hodnoty, která může být rovněž časově náročná, např. výpočet SAD. V této práci byl použit online CGP generátor [19], který vygeneruje překladač chromozomu. Kompilace byla obohacena o výpočet fitness funkce. Aby překlad chromozomu do spustitelné struktury byl co nejvíce efektivní, je nutno přepsat fitness funkci do strojového kódu. Pro tyto účely byla použita reference instrukční sady Intelu x64 [12] spolu s reverzním inženýrstvím již zkompilovaných úseků kódu. Reverzní inženýrství bylo provedeno pomocí programu `objdump` a překladače `gcc` s vhodně nastavenými parametry.

6.3.1 Experimentální vyhodnocení JIT kompilací

Experimenty byly prováděny na grafu o velikosti $n_c = 80$, $n_r = 1$, $l = 40$ s množinou uzlů $\Gamma = \{$BUF, NOT, AND, OR, XOR, NAND, NOR, XNOR$\}$. Fitness funkce byla realizována SAD metodou, jejíž mezivýpočty se ukládaly do vyhrazené paměti. V kompilované verzi fitness funkce byl použit zásobník procesoru. Byly navrhovány tří, čtyř, pěti a šesti bitové sčítačky a čtyř, pěti a šesti bitové násobičky po dobu $n_e = 10^6$ generací. Byl zvolen shodné běhy algoritmu pro porovnání interpretace chromozomu s JIT kompilací chromozomu a s JIT kompilací chromozomu i fitness funkce. Nakonec byla vybrána průměrná

Úloha	TV	n_o	Interpretace	JIT kompilace	JIT kompilace s fitness funkcí
3+3 adder	1	4	**588 077**	327 128	259 358
4+4 adder	4	5	132 500	**151 369**	139 292
5+5 adder	16	6	30 939	86 055	**103 971**
6+6 adder	64	7	7 847	34 638	**71 532**
4x4 multiplier	4	8	108 528	**120 736**	107 512
5x5 multiplier	16	10	27 799	69 156	**88 449**
6x6 multiplier	64	12	7 061	23 917	**59 988**

Tabulka 6.2: Porovnání metod akcelerací pomocí JIT kompilace v závislosti na počtu trénovacích vektorů TV a řešeném problému. Naměřené hodnoty jsou uvedeny jako počet evaluací za sekundu, dle vztahu 6.1. Oranžovou barvou jsou znázorněny nejlepší výsledky pro danou úlohu.

hodnota počtu evaluací za sekundu z deseti různých běhů. Počet evaluací za sekundu byl vypočten dle vztahu

$$eps = \frac{\lambda * n_e}{t}, \tag{6.1}$$

kde t je doba trvání běhu programu. Měření probíhalo na 2.3 GHz procesoru Intel Core i3-2350M.

Výsledky experimentů jsou uvedeny v tabulce 6.2. Z výsledků vyplývá, že předkompilace chromozomu se vyplatí, máme-li alespoň čtyři 64-bitové trénovací vektory. Dále bylo zjištěno, že zahrnutí fitness funkce do JIT kompilace je vhodné již od šestnácti trénovacích vektorů.

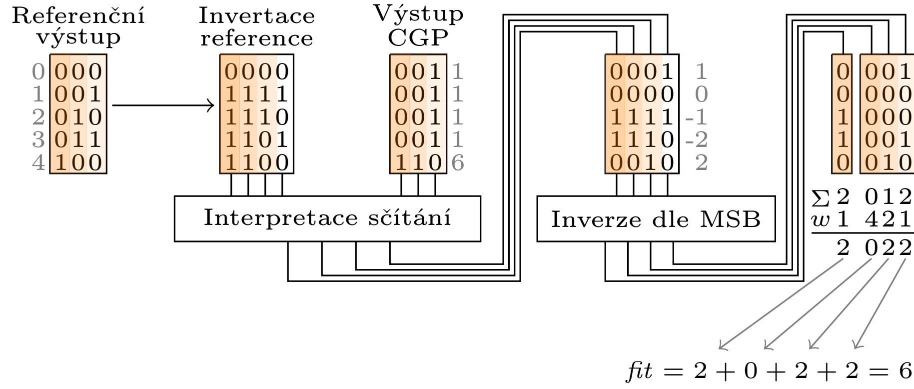
6.4 Paralelní simulace výpočtu fitness hodnoty pro aritmetické obvody

Použije-li se paralelní simulace, potom je vhodné implementovat i efektivní výpočet fitness hodnoty, který je v případě SAD metody poměrně složitý. Odezvou paralelní simulace obvodu v CGP je n_o 64-bitových vektorů, viz podkapitola 6.1. Tyto vektory jsou seřazeny od nejvíce významných bitů po nejméně významné. Pro výpočet fitness hodnoty fit_{SAD}, která je definovaná vztahem 4.4, můžeme provést tzv. transpozici bitů. Tím je myšlen převod n_o 64-bitových vektorů na 64 celočíselných hodnot, ze kterých snadno vypočteme fitness hodnotu fit_{SAD}. Tento postup byl proveden v práci [16], [20] a [21].

Problematickou částí dosavadního přístupu je právě v procesu transpozice vektorů, jelikož je potřeba mnoha bitových posuvů, aritmeticko-logických operací a skokových instrukcí, které zabraňují jejich zřetězenému zpracování v CPU. Dále je nutností mít alokovanou paměť pro výstupní odezvu a přístup do ní může značně zpomalit běh programu.

Akcelerace lze dosáhnout interpretací logických obvodů na vektorové úrovni registrů. K paralelnímu výpočtu SAD využijeme 64-bitové registry a booleovské operace podobně, jako v paralelní simulaci kandidátního řešení. Výpočet je tedy založen na interpretaci odčítačky a absolutní hodnoty pomocí booleovských funkcí.

Odčítání realizujeme následovně. Před zahájením evoluce CGP invertujeme referenční výstup specifikace obvodu C_r. Invertovaný referenční výstup C_r^- se v průběhu simulace sečte



Obrázek 6.1: Paralelní výpočet fitness hodnoty SAD na 5-ti bitových vektorech, kde Σ značí *popcnt* a w je váha vektoru. Šedou barvou označeny celočíselné hodnoty.

s výstupem o kandidátního obvodu C_o. Operace sčítání je implementována jako interpretace sčítačky s postupným přenosem na bitové úrovni. Nejdříve se vypočte hodnota výstupu s_0 a příznak přenosu c_0 podle Booleových vzorců 6.2 a 6.3, které znázorňují poloviční binární sčítačku. Potom se postupně získají jejich j-té hodnoty, jak je uvedeno ve vztazích 6.4 a 6.5.

$$s_0 = C_r^-(0) \oplus C_o(0) \tag{6.2}$$
$$c_0 = C_r^-(0) \cdot C_o(0) \tag{6.3}$$
$$s_j = C_r^-(j) \oplus C_o(j) \oplus c_{j-1} \tag{6.4}$$
$$c_j = C_r^-(j) \cdot C_o(j) + (C_r^-(j) \oplus C_o(j)) \cdot c_{j-1} \tag{6.5}$$

Problémové je také získání absolutní hodnoty. Po odečtení jsou některé hodnoty v bitových vektorech kladné a některé záporné. Na nejvíce významném vektoru s_{n_o} je uložen příznak záporného čísla, podle kterého se invertují zbylé vektory $s_j = s_j \oplus s_{n_o}, j \neq n_o$.

Nyní máme skoro vypočtenou sumu absolutních diferencí pro 64 hodnot. Každý vektor reprezentuje určitý binární základ w, např. 1, 2, 4, 8, 16 atd. V jednotlivých vektorech j se určí počet jedniček *popcnt*, který se následně vynásobí váhou vektoru $w = 2^j$. Nutno podotknout, že v nejvíce významném vektoru (MSB) je uložen příznak přenosu, který vznikl při odčítání vektorů. Tuto hodnotu musíme přičíst k výsledné fitness, ale přičteme ji bez váhy. Získání absolutní hodnoty můžeme zapsat vztahem 6.6.

Celý postup paralelní simulace je znázorněn na obrázku 6.1.

$$fit = popcnt(s_{n_o}) + \sum_{j=0}^{n_o-1} 2^j * popcnt(s_{n_o} \oplus s_j) \tag{6.6}$$

6.4.1 Experimentální vyhodnocení paralelního výpočtu

Budeme zkoumat tři možné implementace fitness funkce. První metoda (A1) počítá fitness SAD pomocí bitové transpozice vektorů. Druhá metoda (A2) využívá paralelního výpočtu SAD. Třetí urychlovací metoda (A3) předkompilovává paralelní výpočet SAD, jedná se tedy o JIT kompilaci metody A2. Akcelerační techniky porovnáme dle hodnoty *eps*, která je definovaná vztahem 6.1.

		Ohodnocení jedinců					
		S neutrálními mutacemi			Bez neutrálních mutací		
Násobička	$n_e * 10^6$	A1 [*eps*]	A2 [*eps*]	A3 [*eps*]	A1 [*eps*]	A2 [*eps*]	A3 [*eps*]
4x4	10	29 917	**230 149**	177 970	54 912	**277 923**	226 222
5x5	5	6 558	124 336	**142 373**	12 052	168 688	**186 781**
6x6	1	1 386	41 468	**85 696**	2 683	61 673	**129 541**
7x7	0,1	303	10 600	**38 142**	602	17 644	**53 930**
8x8	0,01	66	2 432	**11 325**	135	4 782	**23 190**

Tabulka 6.3: Porovnání rychlostí výpočtu funkce SAD. Metoda A1 počítá pomocí bitové transpozice vektorů. Metoda A2 je paralelní výpočet SAD. Metoda A3 předkompilovává paralelní výpočet SAD.

Pro každou metodu algoritmus využíval následující optimalizace: neohodnocování nepoužitých hradel a předkompilovanou 64-bitovou paralelní simulaci obvodu. Pro každou násobičku a každou akcelerační techniku bylo zvoleno celkem 10 nezávislých běhů, ze kterých byla spočtena průměrná hodnota *eps*. Nastavení parametrů CGP bylo zvoleno pro všechny běhy následovně: $\Gamma = \{\text{BUF, INV, AND, OR, XOR, NAND, NOR, XNOR}\}$, $n_r = 1$, $n_c = 40$, $l = 40$, $\lambda = 4$, $h = 2$. Ohodnocovaly se násobičky 4x4, 5x5, 6x6, 7x7 a 8x8 se specifickým počtem generací n_e. Tento experiment probíhal na 2.3 GHz procesoru Intel Core i3-2350M.

Výsledky experimentu jsou zobrazeny v tabulce 6.3. Můžeme vidět, že paralelní výpočet (A2) je efektivnější než původní metoda (A1). Pro větší násobičky (5x5 a víc) je však vhodnější použít kompilovanou fitness funkci (A3). Tento fakt vyplynul již z výsledků předchozího experimentu. Návrh 8-bitové násobičky byl tedy celkově urychlen víc než **170** krát oproti původní metodě (A1).

Dále můžeme sledovat zrychlení ohodnocení, přeskakujeme-li neutrální mutace. U této metody však záleží na velikosti prohledávacího prostoru a četností mutovaných genů.

Kapitola 7

Evoluce aproximačních aritmetických obvodů

Tato kapitola se věnuje návrhu a vyhodnocení aritmetických obvodů. Pro návrh aproximačních aritmetických obvodů pomocí CGP byl využit algoritmus prezentovaný v podkapitole 4.3.2. Ohodnocení kandidátních řešení bylo akcelerováno pomocí technik, které byly uvedeny v předchozí kapitole.

7.1 Návrh přibližných sčítaček a jejich vyhodnocení

Inicializace CGP proběhla za pomocí plně funkční sčítačky, která byla zkonstruována podle Kogge-Stonova algoritmu, který byl popsán v kapitole 2.2.1. Nastavení CGP bylo zvoleno následovně: $\Gamma = \{\text{BUF, INV, AND, OR, XOR, NAND, NOR, XNOR}\}$, $n_r = 13$, $n_c = 7$ což odpovídá zpoždění na úrovni hradel plně funkční Kogge-Stone sčítačky, $l = 7$, $\lambda = 4$, $h = 4$, $n_e = 500\ 000$. Počet hradel plně funkční sčítačky činil $n_g = 73$. Tato hodnota se po 50-ti bězích CGP dekrementovala. Pro namapování sčítačky o 73 hradlech do grafu o velikosti 7x13 bylo využito algoritmu ALAP (zkratka z angl. As Late As Possible).

Díky evolučnímu návrhu byla nalezena implementace plně funkční sčítačky s méně hradly ($n_g = 62$) a se shodným zpožděním. Vlastnosti aproximovaných sčítaček jsou zobrazeny v tabulce 7.1. Vidíme, že relativní zpoždění přibližných sčítaček může být horší než plně funkční řešení, avšak relativní plocha má klesavou tendenci.

7.2 Návrh přibližných násobiček a jejich vyhodnocení

Inicializace CGP proběhla za pomocí plně funkční násobičky, která byla zkonstruována pomocí Wallaceova stromu, který byl popsán v kapitole 2.2.3. Počet hradel plně funkční násobičky činil $n_g = 330$. Tato hodnota se po 50-ti bězích CGP dekrementovala. Nastavení CGP bylo zvoleno následovně: $\Gamma = \{\text{BUF, INV, AND, OR, XOR, NAND, NOR, XNOR}\}$, $n_r = 1$, $n_c = n_g + 3$, $l = n_c$, $\lambda = 4$, $h = 0.05 n_c$, $n_e = 500\ 000$. Pro dvoutýdenní běh byl využit školní server edesign. Vlastnosti jednotlivých násobiček lze porovnat v tabulce 7.2.

7.3 Graf rozptylu chyb pro aritmetické obvody

V tabulkách 7.1 a 7.2 jsou zachyceny pouze statistické údaje o chybě. Pro uživatele přibližných aritmetických obvodů je vhodnější chybu znázornit tak, aby věděl, kdy chyba nastane,

Počet Hradel	Relativní plocha	Relativní zpoždění	Chyba MIN	Q1	Q2	Q3	MAX
1	24	1,00	1	16	33	55	128
6	320	8,85	1	8	17	28	64
12	624	10,31	1	5	10	17	64
18	848	9,49	1	4	9	17	64
24	1 120	9,80	1	3	7	13	64
30	1 312	15,07	1	3	7	15	64
36	1 648	12,16	1	3	7	15	64
42	1 024	8,74	1	4	10	23	64
43	2 416	12,05	1	1	1	2	3
48	2 520	13,69	1	1	1	1	1
54	2 784	13,63	1	1	1	1	1
60	2 888	14,62	1	1	1	1	1
62	3 272	14,05	0	0	0	0	0

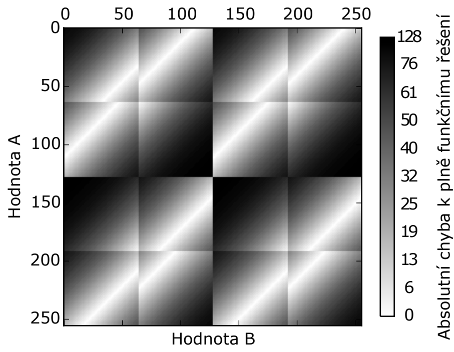
Tabulka 7.1: Vlastnosti evolučně navržených sčítaček. Relativní plocha a zpoždění odpovídá VLSI hodnotám z tabulky 2.2. Sloupce Q1-Q3 označují kvartily chyb.

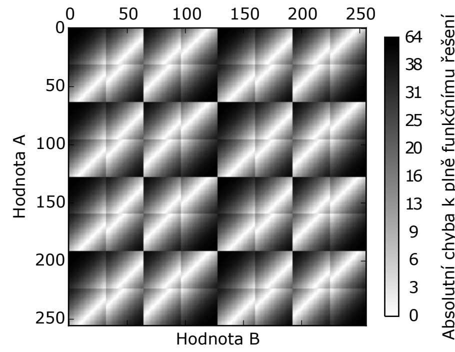
Počet Hradel	Relativní plocha	Relativní zpoždění	Chyba MIN	Q1	Q2	Q3	MAX
1	40	1,76	1	2 437	4 781	7 373	16 600
30	1 464	18,60	1	467	1 078	1 874	5 888
60	3 136	36,83	1	237	495	838	3 714
90	4 712	41,43	1	135	285	487	5 257
120	6 040	34,99	1	84	178	305	924
150	7 672	42,80	1	60	127	214	739
180	9 232	44,89	1	36	76	129	2 390
210	10 872	51,15	1	21	41	70	244
240	12 712	48,81	1	10	21	35	106
270	14 048	54,11	1	6	11	17	48
300	15 672	52,15	1	2	3	7	18
319	16 760	51,03	2	2	2	2	2
330	16 752	56,66	0	0	0	0	0

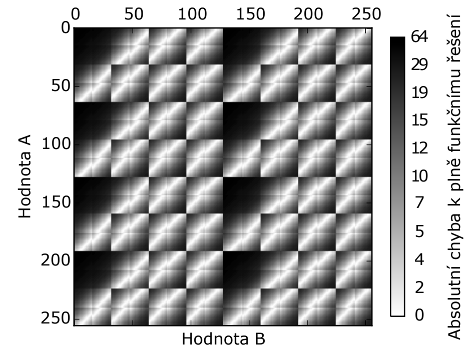
Tabulka 7.2: Vlastnosti evolučně navržených násobiček. Relativní plocha a zpoždění odpovídá VLSI hodnotám z tabulky 2.2. Sloupce Q1-Q3 označují kvartily chyb.

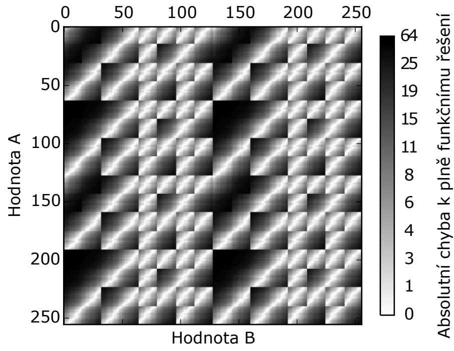
a aby věděl, jaká je její velikost. Budeme vycházet z grafu rozptylu chyb, který byl popsán v kapitole 3. Graf rozptylu chyb znázorňuje pouze případy, kdy nastává chyba. Pro aritmetické obvody je nutno tento graf upravit tak, abychom získali informaci o velikosti chyby. Pro tyto účely zavedeme třetí rozměr definovaný barvou.

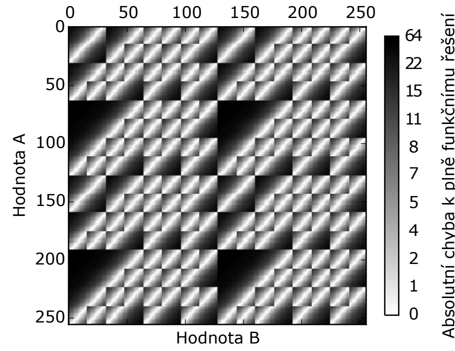
Graf rozptylu chyb pro aritmetické obvody můžeme vytvořit následovně. Víme, že barvu šedi dokážeme zakódovat do 256 bitů. Jinými slovy máme celkem 256 barev, které lze seřadit podle jejich jasu. Z přibližného obvodu získáme pro všechny kombinace absolutní velikost chyby. Chyby následně seřadíme a určíme 256 kvantilů, které odpovídají hodnotě jasu. Ve výsledku jasné barvy značí malou chybu a tmavé znázorňují velkou chybu. Pro každou vstupní kombinaci přiřadíme příslušný kvantil a zobrazíme do grafu. Výsledné grafy si lze prohlédnout na obrázcích 7.1 a 7.2 , kde můžeme pěkně pozorovat vývoj chyby.

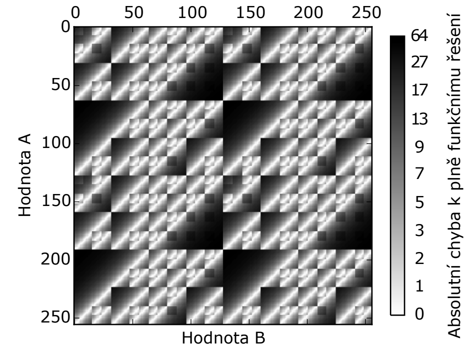


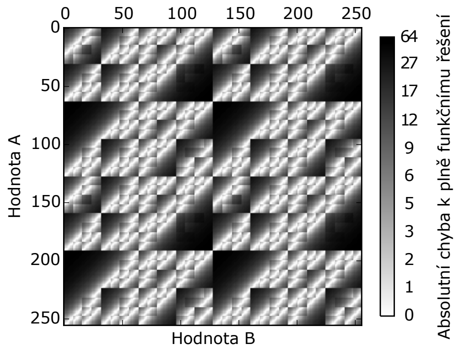


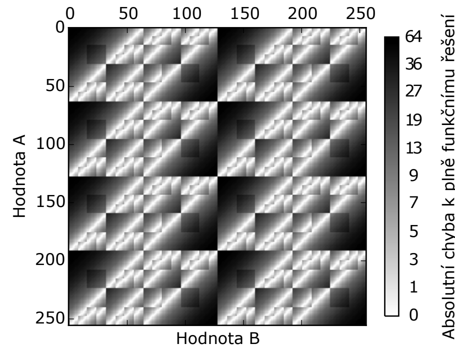


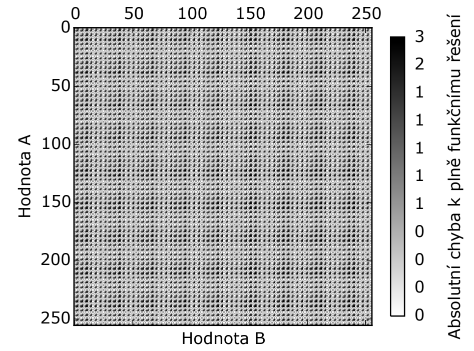


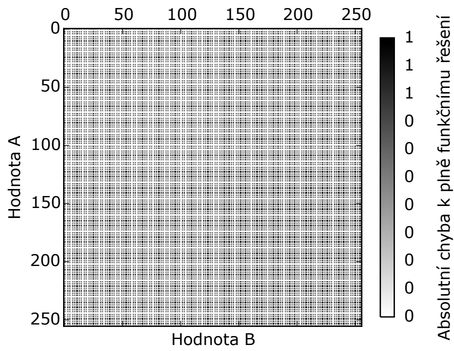


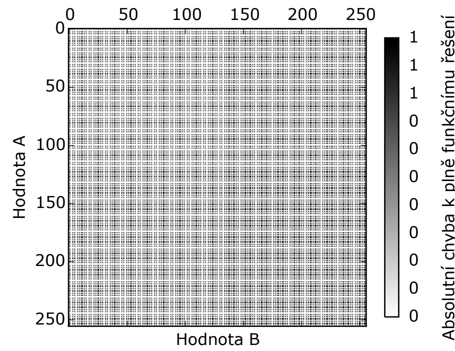


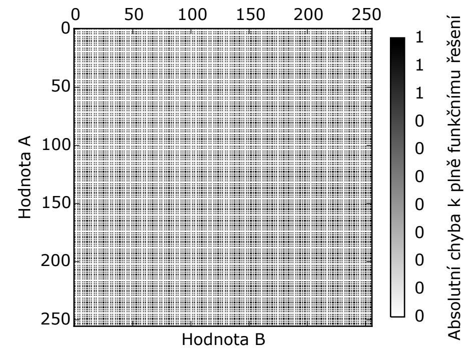




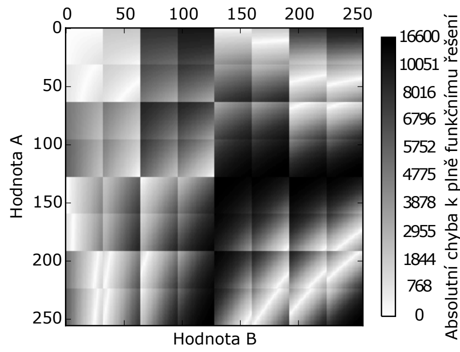


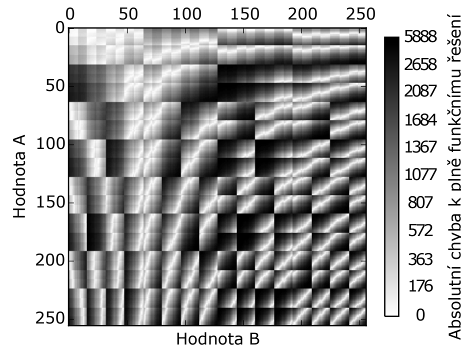


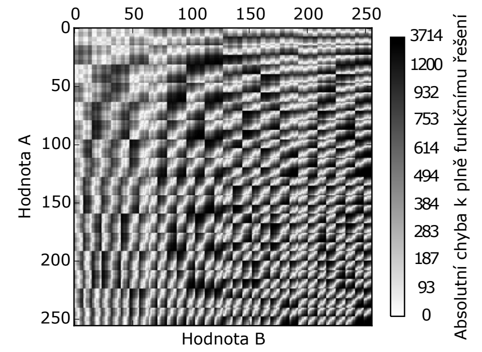


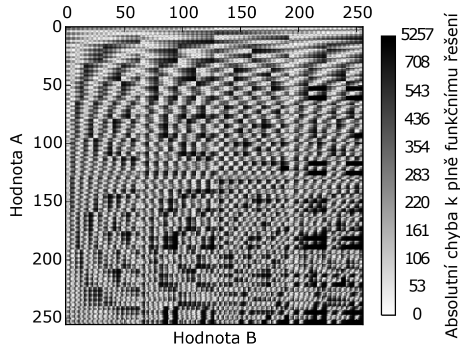


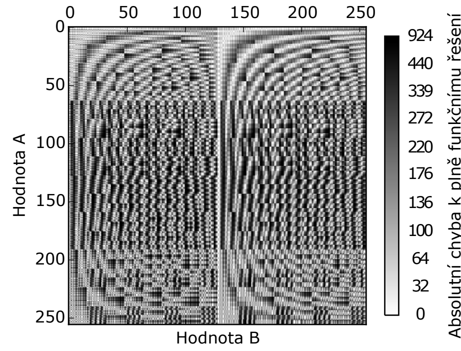
Obrázek 7.1: Grafy chyb vyvinutých přibližných sčítaček

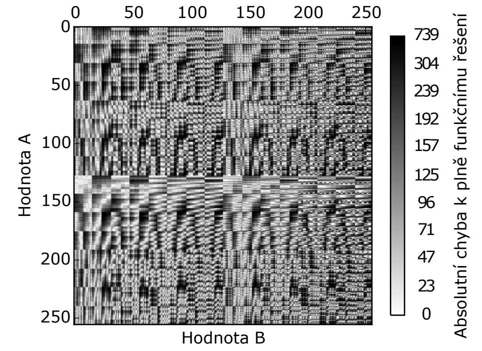


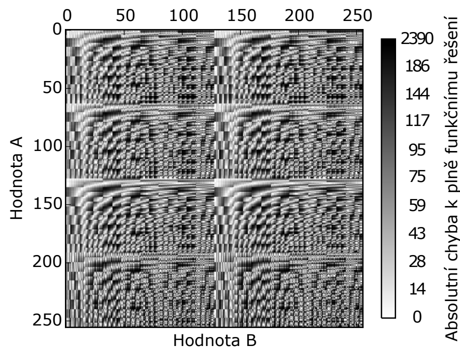


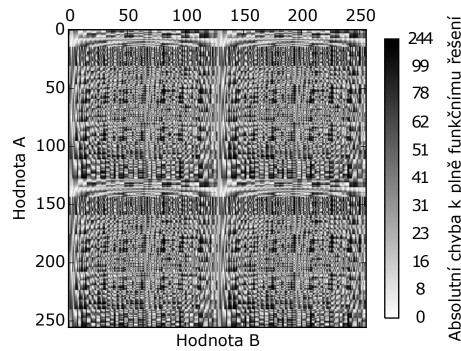


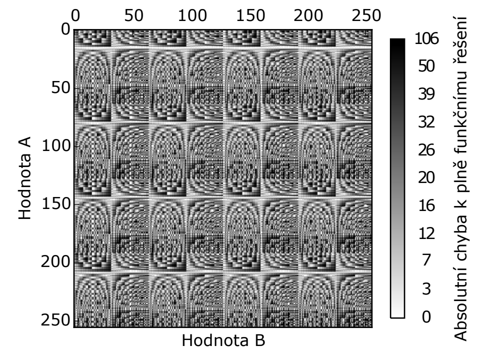


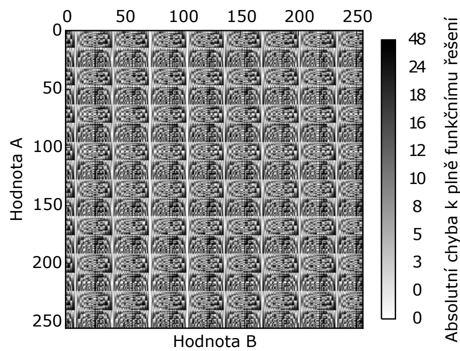


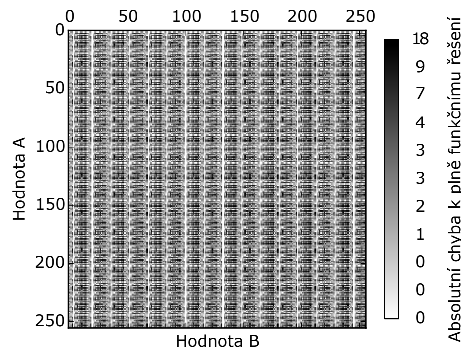


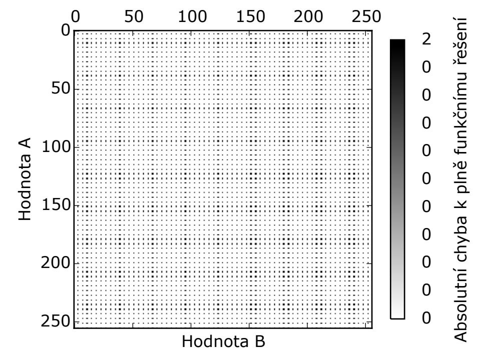












Obrázek 7.2: Grafy chyb vyvinutých přibližných násobiček

Kapitola 8

Použití navržených přibližných obvodů v detekci hran

Tato kapitola popisuje použití navržených přibližných obvodů v detekci hran. První část kapitoly se zabývá použitím aproximačních sčítaček v Sobelově detekci hran. Druhá část se věnuje aproximaci v Cannyho detektoru hran.

Každá část této kapitoly se nejdříve věnuje hardwarové implementaci. Následně se určí prvky, které budou aproximovány. Nakonec se provede experimentální vyhodnocení aproximovaných detektorů s plně funkčními řešeními.

8.1 Přibližná Sobelova detekce hran

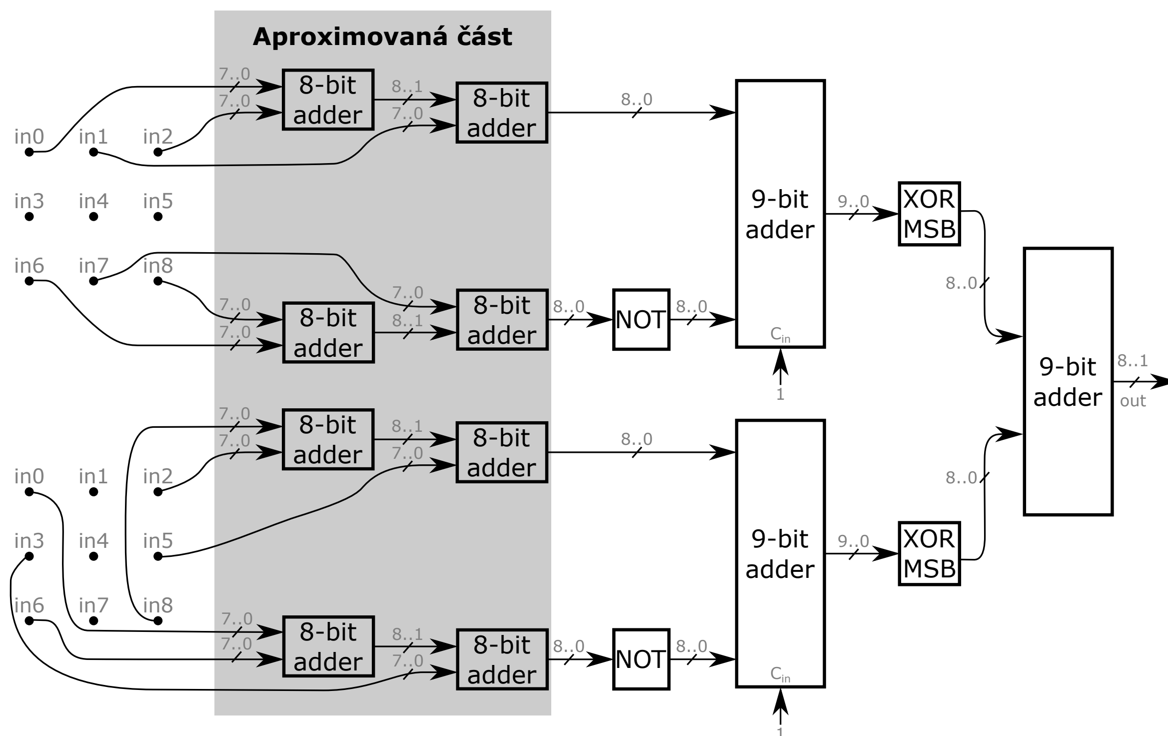
V kapitole 5 bylo sděleno, že proces Sobelovy detekce hran lze rozdělit na dvě fáze: zvýraznění oblastí s vysokou změnou intenzity a následné prahování. První část nám poskytuje více informací o magnitudě detekovaných hran než druhá část, ve které se informace ztrácí prahováním. Proto budeme pouze navrhovat hardwarovou implementaci první části.

8.1.1 Hardwarová implementace a návrh aproximace

Konvoluci s použitím Sobelových jader lze realizovat pouze za pomocí osmibitových sčítaček a odčítaček. Je nelogické aplikovat násobičky, neboť Sobelova konvoluční maska obsahuje pouze hodnoty 0, 1, 2 a jejich zápory. Součin realizovaný konstantou 0 se ve výsledné sumě nikterak neprojeví, neboť nula je absorbující prvek. Konstanta 1 je neutrální prvek pro násobení, tudíž jej nemusíme provádět. Násobení konstantou 2 lze vykonat pomocí bitového posuvu. Záporné hodnoty můžeme realizovat odčítačkou.

Pro hardware je vhodnější magnitudu vypočítat podle vztahu $|G| = |G_x| + |G_y|$. Absolutní hodnoty $|G_x|$ a $|G_y|$ lze získat inverzí bitů podle MSB. Přičtení příznaku přenosu můžeme zanedbat. Výsledný obvod je znázorněn na obrázku 8.1.

V navrženém obvodu aproximujeme rychlé Kogge-Stone sčítačky, které vykonávají konvoluci. Odčítání s výpočtem gradientu zůstane vždy přesné.



Obrázek 8.1: Návrh hardwarové implementace Sobelova detektoru hran

8.1.2 Experimentální vyhodnocení

Výsledky aproximovaného řešení lze spatřit na obrázku 8.2. Vidíme, že můžeme aproximovat sčítačky až na 43 hradlech, kde lidské oko nepostřehne chybu. U sčítaček s méně hradly lze snadněji pozorovat chybu, např. na rameni Lenny vidíme chybně detekované hrany.

Dle tabulky 8.1 bylo zjištěno, že plně funkční implementace detektoru hran se skládá z 897 hradel, jejichž relativní plocha je 41 716. Relativní zpoždění obvodu činilo 65,16. Nahradíme-li přesné osmibitové sčítačky aproximovanými o 43 hradlech, snížíme celkovou plochu na 34 868. Relativní zpoždění se redukuje na hodnotu 61,13. Docílilo se zmenšení plochy o 16,4% a zrychlení o 9%, přičemž chyba detekce je pro lidské oko nepatrná.

Obvod	Počet použití	Počet hradel	Relativní plocha	Relativní zpoždění
8-bit sčítačka	8	62	3 272	14,05
9-bit NOT	9	18	432	1,00
9-bit sčítačka (se vstupem Cin)	3	79	3 708	17,24
9-bit XOR	2	18	648	1,55
Celkem	Σ	897	41 716	*65,13

Tabulka 8.1: Parametry hardwarové implementace přesného Sobelova detektoru hran. Relativní plocha a zpoždění odpovídá VLSI hodnotám z tabulky 2.2.
* Jedná se o kritickou cestu.

(a) 1 hradlo

(b) 6 hradel

(c) 12 hradel

(d) 18 hradel

(e) 24 hradel

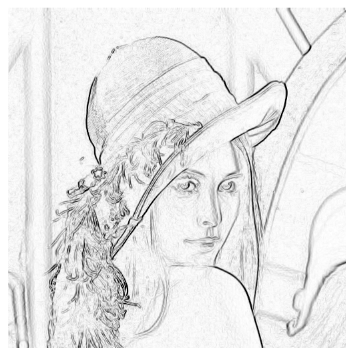
(f) 30 hradel

(g) 36 hradel

(h) 42 hradel

(i) 43 hradel

(j) 48 hradel



(k) 54 hradel

(l) 62 hradel (plně funkční)

Obrázek 8.2: Použití různě aproximovaných sčítaček v Sobelově detekci hran

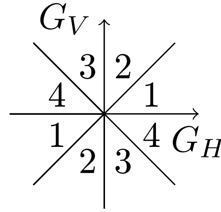
8.2 Přibližná Cannyho detekce hran

V kapitole 5 bylo vysvětleno, že Cannyho hranový detektor je algoritmus, který pracuje následovně. Nejdříve se eliminuje šum, potom se vypočtou magnitudy. Pak probíhá detekce lokálních maxim a nakonec se provede dvojité prahování s hysterezí.

8.2.1 Návrh aproximace

Musíme rozhodnout, kterou část algoritmu budeme aproximovat. Byl představen článek [9] o hardwarové aproximaci Gaussova filtru nevyžadující násobení. Aproximaci konvoluce pomocí Sobelových operátorů jsme si ukázali v předchozí podkapitole. V hysterezi nemůžeme aplikovat navržené sčítačky nebo násobičky. Vylučovací metodou nám zůstala detekce lokálních maxim, jinými slovy potlačení nemaxim.

Operace tan^{-1} se v hardwaru implementuje dosti náročně. Určení úhlů lze vypočítat pomocí přesných úhlů podle vertikální (G_V) a horizontální složky gradientu (G_H). Proto vypočteme úhly podle vertikálních a horizontálních složek Sobelových filtrů, jak je znázorněno na grafu 8.3.



Obrázek 8.3: Získání čtyř složek přibližných orientací úhlů podle velikosti Sobelových složek.

Proces ztenčení je založen na získání magnitudy sousedních pixelů, které se nachází na normále od daného směru hrany. Pokud jsou sousední magnitudy menší než magnituda daného pixelu, potom se jedná o lokální maximum. Výpočet normály můžeme realizovat váhováním sousedních magnitud. První část detekce lokálního extrému pro první směr je znázorněna vztahy 8.1 a 8.2, kde horní indexy odpovídají souřadnicím obrázku. Pokud obě nerovnice platí, jedná se o lokální extrém. Úpravou této nerovnice získáme vztah 8.3 a 8.4 s násobením, které můžeme aproximovat.

$$|G^{x-1,y-1}|\frac{G_V^{x,y}}{G_H^{x,y}} + |G^{x,y-1}|(1 - \frac{G_V^{x,y}}{G_H^{x,y}}) \leq |G^{x,y}| \tag{8.1}$$

$$|G^{x-1,y-1}|\frac{G_V^{x,y}}{G_H^{x,y}} + |G^{x,y+1}|(1 - \frac{G_V^{x,y}}{G_H^{x,y}}) \leq |G^{x,y}| \tag{8.2}$$

$$|G^{x-1,y-1}|G_V^{x,y} - |G^{x,y-1}|(G_V^{x,y} - G_H^{x,y}) \leq |G^{x,y}|G_H^{x,y} \tag{8.3}$$

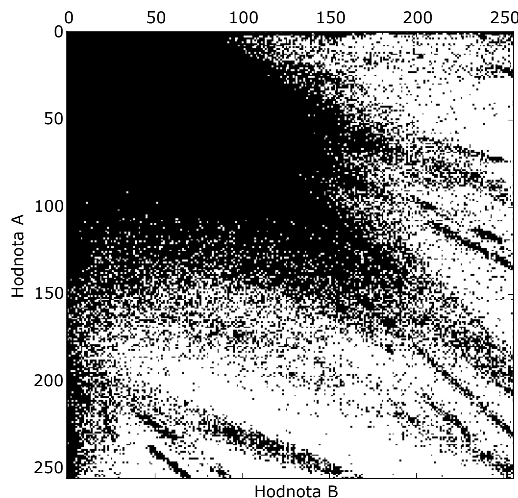
$$|G^{x-1,y-1}|G_V^{x,y} - |G^{x,y+1}|(G_V^{x,y} - G_H^{x,y}) \leq |G^{x,y}|G_H^{x,y} \tag{8.4}$$

Náročnou částí je však hardwarová realizace dvojitého prahování s hysterezí, kde je potřeba speciální paměti o velikosti obrázku. Pro zjednodušení problému budeme předpokládat, že Cannyho detekce hran bude implementována softwarově pro procesor se zabudovaným přibližným násobením.

8.2.2 Tvorba přibližných násobiček podle částečné specifikace

V předchozí kapitole jsme si ukázali vývoj násobičky podle úplné specifikace (S1), kde se definovalo všech 2^{16} možných vstup-výstupních kombinací. V této kapitole se budou aproximovat osmibitové násobičky podle částečné specifikace (S2). Takže definujeme pouze některé vstup-výstupní kombinace násobičky.

Je zřejmé, že násobička v procesu ztenčení hran nevyužívá všechny možné kombinace operandů, ale pouze část, jak je znázorněno na obrázku 8.4. Tento graf znázorňuje využití operandů násobení a podle tohoto využití definujeme částečnou specifikaci.



Obrázek 8.4: Využití operandů násobení v Cannyho detekci hran. Černou barvou znázorněno, že došlo k násobení daných prvků. Bíla barva značí, že daná kombinace nebyla využita. Hodnoty brány z několika obrázků.

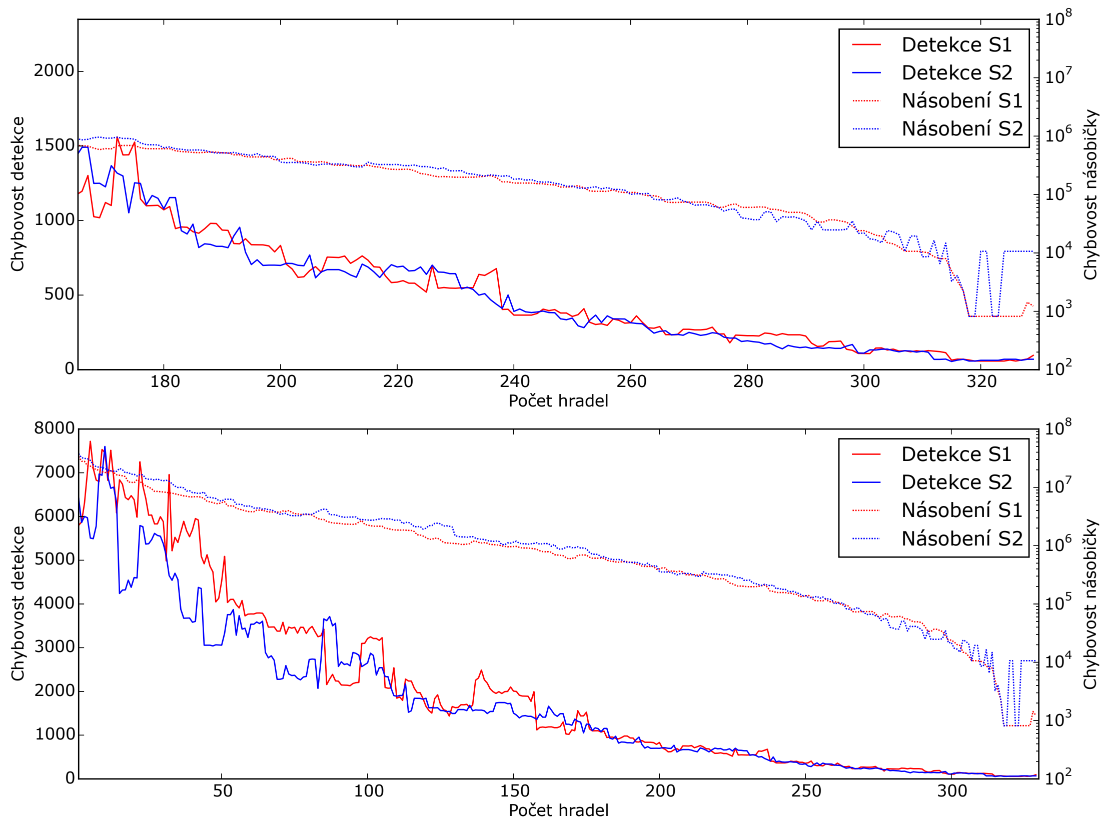
Aby bylo porovnání částečné specifikace (S2) férové, nastavíme běh CGP shodnými parametry, kterými byly vytvořeny násobičky podle úplné specifikace (S1). Takže inicializace algoritmu CGP proběhla za pomocí plně funkční násobičky, která byla zkonstruována pomocí Wallaceova stromu [25]. Počet hradel plně funkční násobičky činil $n_g = 330$. Tato hodnota se po 50-ti bězích CGP dekrementovala. Nastavení CGP bylo zvoleno následovně: $\Gamma = \{$BUF, INV, AND, OR, XOR, NAND, NOR, XNOR$\}$, $n_r = 1$, $n_c = n_n + 3$, $l = n_c$, $\lambda = 4$, $h = 0.05n_c$, $n_g = 500\,000$. Navržené obvody byly porovnány s násobičkami, které byly navrhovány podle úplné specifikace.

8.2.3 Experimentální vyhodnocení

V této podkapitole porovnáme návrh násobiček podle úplné specifikace (S1) a částečné specifikace (S2). Budeme sledovat relativní chybovost násobičky, počet chyb v detekovaném obrázku a počet evaluací kandidátní násobičky.

Výsledky detekce pomocí přibližných násobiček vyvinutých podle metody S1 si můžeme prohlédnout na obrázku 8.6. Vidíme, že jsou hrany ztenčeny správně u násobení sestaveného z 210 hradel. U násobení o 180 hradlech lze pozorovat chybně ztenčenou oblast pod komínem.

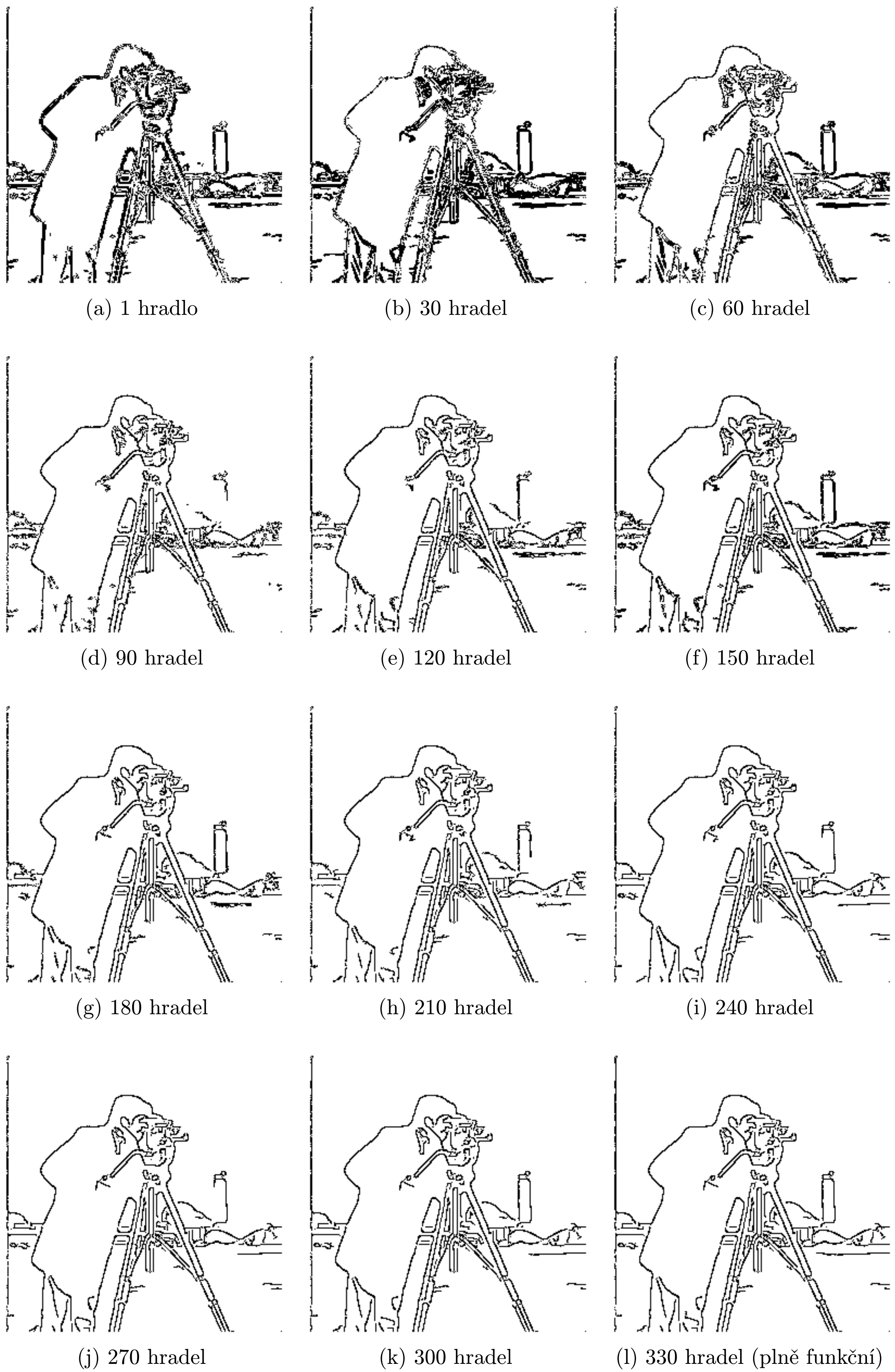
Při podrobnější analýze zjistíme, že obě metody návrhu násobiček (S1 a S2) se chovají různě v detekci hran. Tento fakt je zachycen grafem na obrázku 8.5. Například detekce hran pomocí přibližných násobiček, kde počet hradel $n_g = 217 \ldots 230$, dopadla pro metodu S2



Obrázek 8.5: Porovnání chybovosti aproximačních násobiček a použití v detekci hran. Chybovost násobičky byla počítána dle vztahu 4.4. Chybovost detekce byla stanovena dle chybně určených pixelů vůči plně funkčnímu řešení.

hůře. Avšak detekce hran, kde $n_g = 15 \dots 85$, dopadla pro metodu S2 podstatně lépe, i když měla hůře aproximované násobičky.

Naprostá výhoda metody S2 je v rychlosti ohodnocení kandidátního řešení, jelikož se neohodnocují všechny případy násobení (kterých je 2^{16}), ale pouze část ($31594 \simeq 2^{15}$ případů fitness). Urychlení metody S2 oproti metodě S1 je tedy dvojnásobné.



(a) 1 hradlo (b) 30 hradel (c) 60 hradel

(d) 90 hradel (e) 120 hradel (f) 150 hradel

(g) 180 hradel (h) 210 hradel (i) 240 hradel

(j) 270 hradel (k) 300 hradel (l) 330 hradel (plně funkční)

Obrázek 8.6: Použití různě aproximovaných násobiček v Cannyho detektoru hran v procesu ztenčení hran, kde $\sigma = 1,5$, $T_L = 15$ a $T_H = 30$.

Kapitola 9

Závěr

V této práci byly vysvětleny principy činnosti číslicových obvodů, jejich parametry a konvenční návrhové metody. Rovněž bylo ukázáno konvenční sestrojení aproximačních obvodů a jejich vyhodnocení. Byly popsány možnosti evoluční aproximace pomocí kartézského genetického programování, na které tato práce navázala.

Byl prezentován možný výpočet fitness funkce pomocí vážených Hammingových vzdáleností. Tuto funkci můžeme využít v nearitmetických kombinačních obvodech, kde každý výstup má určitou prioritu, například kodéry nebo multiplexory.

Dále byl akcelerován výpočet fitness funkce sumy absolutních diferencí. Díky paralelnímu výpočtu a jeho zahrnutí do JIT kompilace byla evaluace osmibitové násobičky významně urychlena víc než 170 krát oproti původnímu přístupu.

Akcelerované CGP bylo použito pro vývoj osmibitových aproximačních prvků – sčítačky a násobičky. Přibližné sčítačky nahradily přesně pracující v konvoluci v Sobelovém detektoru hran. V obvodu, který realizoval detekci hran, se docílilo zmenšení jeho plochy o 16% a zrychlení o 9%, přičemž chybovost detekce byla vzhledem k lidským smyslům nepatrná.

Aproximované násobičky byly dále použity v Cannyho detekci hran pro výpočet směru gradientu. Byly ukázány dva možné přístupy k ohodnocení násobiček určených k detekci hran. První z nich hodnotil násobičku podle úplné specifikace a druhý podle částečné specifikace. Druhá varianta nám umožňuje aproximovat obvody ještě rychleji – záleží na počtu odebraných trénovacích vektorů. Z experimentálního porovnání těchto přístupů vzešlo, že násobičky navržené dle částečné specifikace mohou lépe detekovat hrany, i když jsou hůře aproximované.

Dále byl prezentován nový typ grafu chyb pro přibližné aritmetické obvody. Tento graf dobře zachycuje výskyt a velikost chyb.

V budoucí práci se můžeme zaměřit na eliminaci problému škálovatelnosti CGP pomocí dekompozice, či dynamické fitness funkce. Dále můžeme hledat další aplikace, které jsou tolerantní k chybám. Další možností pokračovaní by mohla být realizace přibližného detektoru v hardwaru a posoudit komplexně dopad aproximace.

Tato práce byla přednášena na studentské konferenci Excel@FIT 2015, kde ze 72 prací obdržela cenu sponzora konference TESCAN, páté místo v kategorii vědecký článek a třetí místo v kategorii inovační potenciál.

Literatura

[1] IEEE Standard for Floating-Point Arithmetic. *IEEE Std 754-2008*, Aug 2008: s. 1–70.

[2] Vsclib Standard Cell Library. 2013.
URL
http://www.vlsitechnology.org/html/cells/vsclib013/lib_gif_index.html

[3] Blinn, J.: Floating-point tricks. *IEEE Computer Graphics and Applications*, ročník 17, č. 4, Jul 1997: s. 80–84, ISSN 0272-1716.

[4] Canny, J.: A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, Nov 1986: s. 679–698, ISSN 0162-8828.

[5] Chippa, V.; Chakradhar, S.; Roy, K.; aj.: Analysis and characterization of inherent application resilience for approximate computing. In *50th ACM / EDAC / IEEE Design Automation Conference*, May 2013, ISSN 0738-100X, s. 1–9.

[6] Choudhory, M.: *Approximate logic circuits: Theory and application*. Dizertační práce, Rice University, 2011.

[7] Emre, Y.; Chakrabarti, C.: Quality-Aware Techniques for Reducing Power of JPEG Codecs. *J. Signal Process. Syst.*, Dec 2012: s. 227–237, ISSN 1939-8018.

[8] Goldman, B.; Punch, W.: Reducing Wasted Evaluations in Cartesian Genetic Programming. In *Genetic Programming*, Lecture Notes in Computer Science, Springer Berlin Heidelberg, 2013, ISBN 978-3-642-37206-3, s. 61–72.

[9] Hsiao, P.-Y.; Chen, C.-H.; Chou, S.-S.; aj.: A parameterizable digital-approximated 2D Gaussian smoothing filter for edge detection in noisy image. In *IEEE International Symposium on Circuits and Systems*, 2006, s. 1–4.

[10] Kogge, P. M.; Stone, H. S.: A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations. *IEEE Transactions on Computers*, Aug 1973: s. 786–793, ISSN 0018-9340.

[11] Kulkarni, P.; Gupta, P.; Ercegovac, M.: Trading Accuracy for Power with an Underdesigned Multiplier Architecture. In *24th International Conference on VLSI Design (VLSI Design)*, Jan 2011, ISSN 1063-9667, s. 346–351.

[12] Ludloff, C.; Mocko, M.; Lopes, A.; aj.: X86 Opcode and Instruction Reference tables.
URL http://ref.x86asm.net/geek64.html

[13] Miller, J. F.: *Cartesian Genetic Programming*. Natural Computing Series, Springer Berlin Heidelberg, 2011, ISBN 978-3-642-17309-7.

[14] Monajati, M.; Fakhraie, S.; Kabir, E.: Approximate Arithmetic for Low-Power Image Median Filtering. *Circuits, Systems, and Signal Processing*, 2015: s. 1–29, ISSN 0278-081X.

[15] Nepal, K.; Li, Y.; Bahar, R.; aj.: ABACUS: A technique for automated behavioral synthesis of approximate computing circuits. In *Design, Automation and Test in Europe Conference and Exhibition (DATE), 2014*, March 2014, s. 1–6.

[16] Sekanina, L.; Vašíček, Z.: Approximate Circuits by Means of Evolvable Hardware. In *2013 IEEE International Conference on Evolvable Systems (ICES)*, Proceedings of the 2013 IEEE Symposium Series on Computational Intelligence (SSCI), IEEE Computer Society, 2013, ISBN 978-1-4673-5847-7, s. 21–28.

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

[18] Shin, D.: *Techniques For Design and Synthesis of Approximate Digital Circuits for Error-tolerant Applications*. Dizertační práce, University of Southern California, 2012.

[19] Vašíček, Z.: Online CGP generator. 2012.
URL http://www.fit.vutbr.cz/~vasicek/cgp/

[20] Vašíček, Z.; Sekanina, L.: Evolutionary Design of Approximate Multipliers Under Different Error Metrics. In *17th IEEE Symposium on Design and Diagnostics of Electronic Circuits and Systems*, IEEE Computer Society, 2014, ISBN 978-1-4799-4558-0, s. 135–140.

[21] Vašíček, Z.; Sekanina, L.: Evolutionary Approach to Approximate Digital Circuits Design. *IEEE Transactions on Evolutionary Computation*, in press, ISSN 1089-778X.

[22] Vašíček, Z.; Slaný, K.: Efficient Phenotype Evaluation in Cartesian Genetic Programming. In *Proc. of the 15th European Conference on Genetic Programming*, Springer Verlag, 2012, ISBN 978-3-642-29138-8, s. 266–278.

[23] Venkataramani, S.; Sabne, A.; Kozhikkottu, V.; aj.: SALSA: Systematic logic synthesis of approximate circuits. In *Design Automation Conference (DAC), 2012 49th ACM/EDAC/IEEE*, IEEE Computer Society, June 2012, ISSN 0738-100X.

[24] Wakerly, J. F.: *Digital Design: Principles and Practices*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., třetí vydání, 2000, ISBN 0130555207.

[25] Wallace, C.: A Suggestion for a Fast Multiplier. *IEEE Transactions on Electronic Computers*, Feb 1964: s. 14–17, ISSN 0367-7508.

[26] Yazdanbakhsh, A.; Thwaites, B.; Park, J.; aj.: Methodical Approximate Hardware Design and Reuse. In *Workshop on Approximate Computing Across the System Stack (WACAS)*, 2014, s. 1–7.

[27] Zemčík, P.; Španěl, M.; Beran, V.; aj.: *Zpracování obrazu – skripta*. Vysoké učení technické v Brně, 2011.

Příloha A

Obsah CD

Přiložené CD obsahuje veškeré použité zdrojové kódy včetně pomocné literatury a výsledků v následující architektuře:

- `bib` použitá literatura
- `results` výsledky práce
  - `canny_line_surpression` výsledky aproximace ztenčení hran pomocí Cannyho detektoru
  - `dumb_adders` aproximované sčítačky
  - `dumb_mults` aproximované násobičky podle úplné specifikace (S1)
  - `dumber_mults` aproximované násobičky podle částečné specifikace (S2)
  - `sobel_detection` výsledky detekce hran pomocí aproximanovaného Sobelova detektoru
- `src` zdrojové kódy
  - `canny_detector` softwarová realizace aproximovaného Cannyho detektoru
  - `cgp_sad` CGP s výpočtem fitness podle SAD
  - `cgp_shd` CGP s výpočtem fitness podle SHD
  - `cgp_wshd` CGP s výpočtem fitness podle vážené SHD
  - `data` testovací sada dat
  - `sobel_detector` softwarová realizace aproximovaného Sobelova detektoru
  - `sobel_detector_thold` softwarová realizace aproximovaného Sobelova detektoru s využitím prahování
- `tex` technická zpráva ve formátu TEX

Příloha B

Použití implementovaných programů

Evoluční návrh pomocí cgp

Složky

- `/src/cgp_shd`
- `/src/cgp_wshd`
- `/src/cgp_sad`
- `/src/cgp_wsad`

Kompilace kódu

- `make` interpretace chromozomu
- `make jit` předkompilace chromozomu
- `make jitfit` předkompilace chromozomu i s fitness funkcí

Spuštění

`./cgp specifikace.txt [-r počet_řádků] [-c počet_sloucpů] [-l l_back] [-n maximální_počet_hradel] [-m počet_mutovaných_genů] [-p velikost_populace] [-g počet_generací] [-x název_souboru_s_incializaci_populace] [-s seed]`

Spuštění vyžaduje jeden povinný parametr s názvem souboru obsahující specifikaci obvodu `specifikace.txt`. Implicitní hodnoty jsou definovány následovně: `-r 1 -c 40 -l 40 -n 40 -m 1 -p 5 -g 5000000 -s 0` Hodnota seed je přičítána k aktuálnímu času. Soubor s inicializací populace vyžaduje ke korektnímu spuštění soubor obsahující chromozom. Příklady těchto souborů se nachází ve složce `/src/chromozomy`.

Aproximovaná Sobelova detekce hran

Složky

- /src/sobel_detector
- /src/sobel_detector_thold

Kompilace kódu

make

Spuštění

./sobel vstupni_obrazek.png scitacka.txt velikost_prahu

Soubor s aproximovanou sčítačkou vyžaduje specifickou syntaxi. Příklady těchto souborů lze nalézt v adresáři /results/dumb_adders. Pro korektní fungování programu je vyžadováno spuštění se všemi parametry. Prahování v případě použití /src/sobel_detector může být zanedbáno. Příklad použití:

./sobel /src/data/lena.png /results/dumb_adders/20.log 42

Aproximovaná Cannyho detekce hran

Složka

/src/canny_detector

Kompilace kódu

make

Spuštění

./canny vstupni_obrazek.png sigma nižší_práh vyšší_práh nasobicka.txt

Soubor s aproximovanou násobičkou vyžaduje specifickou syntaxi. Příklady těchto souborů lze nalézt v adresáři /results/dumb_mults nebo /results/dumber_mults. Pro korektní fungování programu je vyžadováno spuštění se všemi parametry. Příklad použití:

./canny /src/data/lena.png 2.0 10 20 /results/dumb_mults/42.log