VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

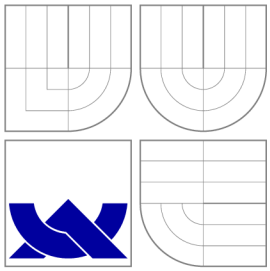
OPTIMALIZACE PROCESOROVÉHO JÁDRA PRO KNIHOVNU OPENCV

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

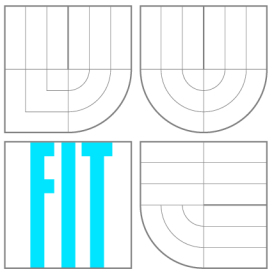
AUTOR PRÁCE
AUTHOR

FILIP BENNA

BRNO 2015



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV INFORMAČNÍCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF INFORMATION SYSTEMS

OPTIMALIZACE PROCESOROVÉHO JÁDRA PRO KNIHOVNU OPENCV

OPTIMIZATION OF A PROCESSOR CORE FOR THE OPENCV LIBRARY

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

FILIP BENNA

VEDOUCÍ PRÁCE
SUPERVISOR

prof. Ing. TOMÁŠ HRUŠKA, CSc.

BRNO 2015

Abstrakt

Tato bakalářská práce se zabývá překladem knihovny OpenCV pro procesorové jádro Codix RISC a následnou optimalizací vybraných aplikací implementovaných s použitím této knihovny. Optimalizace je založena na rozšiřitelnosti procesorového jádra, proto je v této práci teoreticky popsáno a na příkladu vysvětleno přidání vektorových instrukcí do instrukční sady procesoru. Nakonec jsou uvedeny dosažené výsledky a jejich porovnání s neoptimalizovanou verzí aplikace.

Abstract

This bachelor's thesis deals with compilation of OpenCV library with Codix RISC processor core as the target machine and following optimization of chosen applications based on this library. Process of optimization is based on extensibility of the processor core, therefore this thesis theoretically describes and also shows on example extending the processor's instruction set with vector instructions. The thesis also contains the results of the optimization and their comparison with the not optimized application.

Klíčová slova

OpenCV, Codix RISC, SIMD, optimalizace

Keywords

OpenCV, Codix RISC, SIMD, optimization

Citace

Filip Benna: Optimalizace procesorového jádra
pro knihovnu OpenCV, bakalářská práce, Brno, FIT VUT v Brně, 2015

Optimalizace procesorového jádra pro knihovnu OpenCV

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana prof. Ing. Tomáše Hrušky, CSc. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Filip Benna
30. července 2015

Poděkování

Rád bych poděkoval všem, kteří mi s vytvořením této práce pomohli, především panu Ing. Adamu Husárovi, Ph.D. za odbornou pomoc a cenné rady.

Obsah

Kapitola 1

Úvod

Velká část elektronických zařízení, se kterými se dnes setkáváme, se řadí do kategorie tzv. vestavěných systémů, případně k zařízením internetu věcí. Kritickou vlastností těchto zařízení je spotřeba energie, protože se často jedná o zařízení provozovaná na baterii. S tímto omezením jsou vyvíjeny i jednotlivé komponenty pro tato zařízení jako jsou procesory. Pokud je nutné vyvinout procesor, který bude prostorově malý s nízkou spotřebou energie, je nevyhnutelné obětovat výkon tohoto čipu, případně některé vlastnosti, kterými jiné čipy disponují. Zde nastává problém, jelikož většina již implementovaných knihoven a aplikací a to především těch, které pracují v reálném čase, nejsou na těchto procesorech použitelné. Mezi tyto standardně využívané knihovny patří i OpenCV, která obsahuje široké spektrum oblíbených algoritmů pro zpracování obrazu. Pro použití těchto algoritmů na čipech typu *ASIP* je tedy nutné provést jisté optimalizace kódu samotné knihovny a případně využít možnosti, které vývoj aplikačně specifických procesorů nabízí jako je úprava instrukční sady. Přesně tímto problémem se zabývá má práce, která je popsána v této zprávě.

Kapitola 2 slouží k uvedení do aktuálního stavu problematiky, tedy vysvětluje nezbytné pojmy a technologie, které jsou následně zmíněny v dalších kapitolách. Nejvíce se zabývá sadou nástrojů Codasip Framework, jelikož právě ty jsou využity při dále popsaných optimalizacích. Dále je v této kapitole popsán princip vektorových instrukcí, jakožto jedné z nejvýhodnějších optimalizačních technik aplikačně specifických procesorů.

Kapitola 3 obsahuje popis řešení, který jsem navrhl před samotnou implementací tohoto řešení. Nejedná se tak nutně o jeho finální podobu, ale spíše o vysvětlení toho, jaké aplikace jsem zvolil pro následnou optimalizaci a také optimalizační techniky, které jsou u těchto aplikací použitelné.

Kapitola 4 se již zabývá samotnou implementací, popisuje její jednotlivé kroky, mezi které patří překlad potřebných knihoven a následně optimalizace pomocí odstranění operací s plovoucí desetinnou čárkou a také rozšíření instrukční sady procesoru. Jsou zde také uvedeny výsledky procesu optimalizace a porovnány dva různé přístupy k rozšíření instrukční sady, včetně popisu rozšíření procesorového jádra Codix RISC při využití obou těchto přístupů.

Práce je ukončena závěrem, ve kterém zhodnocuji výsledky práce, a také jsou zde naznačena místa, kam by mohl vývoj založený na této práci pokračovat.

Kapitola 2

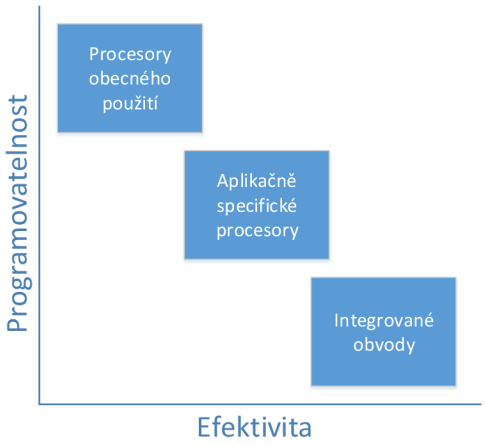
Aktuální stav problematiky

V této kapitole budou popsány některé nástroje a technologie, které jsem dále použil v rámci ostatních kapitol. Cílem kapitoly není vyčerpávající výpis pojmů, ale výběr těch nejdůležitějších, kterým by měl čtenář rozumět, aby pochopil zbytek textu.

2.1 Aplikačně specifické procesory

S procesory se v dnešní době setkáváme naprosto všude. Od procesorů které jsou použity v počítačích, přes procesory v tabletech a chytrých telefonech až po procesory, které ovládají činnost spotřebičů, jako jsou mikrovlnné trouby. Jedná se o komponentu, která je používána již několik desítek let a za tuto dobu prošla dramatickým vývojem. Z počátku se jednalo o jednoduché konečné automaty, které se později vyvinuly v paralelní procesory schopné vykonávat několik instrukcí ve stejný čas, až nakonec evoluce dospěla do stádia, kdy jsou procesory schopny střídat vykonávání několika úloh a tímto simulovat jejich současné provádění. Je tedy patrné, že současné procesory jsou velmi komplikovaná zařízení, která se mohou skládat až z několika miliard transistorů [1]. S tímto se však pojí také růst v ceně takovýchto procesorů a jejich spotřeba energie. Tato negativa nejsou přípustná pro řadu aplikací procesorů. Proto v dnešní době přestáváme procesory chápat jako pevnou entitu, kterou je možné zakoupit a integrovat do jakéhokoliv zařízení. Naopak se vývoj v tomto odvětví začal zabývat jednoduchými specifickými čipy, které jsou vytvořeny za účelem maximálního zefektivnění vykonání dané úlohy a tedy snížení spotřeby energie procesoru a mimo jiné také jeho výrobní ceny. Tyto procesory jsou nazývány aplikačně specifické [1], případně také anglickou zkratkou *ASIP*. Oproti aplikačně specifickým nabízí obecné procesory lepší znovupoužitelnost a případné chyby v návrhu těchto procesorů mohou být vyladěny softwarově. Naopak při návrhu aplikačně specifických procesorů by případné chyby v návrhu znamenaly vážnější problémy znamenající často ztrátu optimalizovaných vlastností procesoru. Stále se však oproti integrovaným obvodům jedná o programovatelné jednotky, takže je možné měnit jimi vykonávanou úlohu. Vztah mezi programovatelností a efektivitou těchto komponent je naznačen na obrázku 2.1.

[1] *Application-specific Instruction Set Processor*



Obrázek 2.1: Poměr mezi programovatelností a efektivitou různých typů procesorů

Celkově je možné rozlišit tři přístupy k vytvoření specializovaných procesorů [2]:

- *Parametrizovatelné procesory* - jedná se o přístup, kdy několik procesorů tvoří rodinu, kde všichni členové vychází z jednoho základu. U tohoto základního čipu je možné zapínat a vypínat jednotlivé funkce (např. *FPU*), případně je škálovat (např. počet registrů) a tímto vytvářet jednotlivé specializované procesory.
- *Rozšiřitelné procesory* - základní verzi čipu je možné rozšířit o aplikačně specifická rozšíření, ať už hardwarová (např. speciální registry) nebo rozšíření instrukční sady procesoru.
- *Specificky vyvinuté procesory* - jedná se o procesory, které jsou od začátku vyvíjeny s jasným cílem použití. Může se jednat buď o vývoj procesoru od úplného začátku nebo je jako výchozí bod použita nějaká šablona. S tímto procesorem je vyvíjen současně i jeho *toolchain*, řetězec obsahující nástroje jako je překladač a simulátor. Ideálně je postačující popis v jednom jazyce a z něj je následně vygenerován jak návrh procesoru, tak přidružené nástroje.

Je samozřejmě patrné, že není možné každý přístup k vývoji *ASIP* zařadit přesně do jedné z těchto kategorií, jelikož se mohou vzájemně překrývat. Předcházející popis je zde začleněn proto, aby bylo možné lépe zasadit procesor Codix RISC do kontextu aplikačně specifických procesorů. Tento procesor byl navrhnut se záměrem, který odpovídá přesně třetí popsané variantě, a sice že slouží jako šablona, která může být následně rozvíjena pro konkrétní účely bez nutnosti pokaždé implementovat základy procesoru. Prakticky celý přístup k vývoji *ASIP* pomocí nástrojů Codasip Frameworku (viz kapitola 2.3) je zaměřen tímto směrem.

2.1.1 Codix RISC

Codix RISC je jedno z procesorových jader nabízených společností Codasip jako šablona pro vývoj dalších procesorů. Jedná se o procesor s 32bitovou šířkou instrukcí a disponuje třiceti dvěma 32bitovými registry obecného použití.

Na základě filozofie návrhu instrukční sady procesoru je možné rozlišit dvě kategorie [3]:

- *CISC* - procesory s komplexní instrukční sadou[2]. Tato kategorie byla populární především v sedmdesátých letech. Ovšem to neznamená, že se dnes již nevyužívá, jelikož populární architektura Intel IA-32 patří do této kategorie. Instrukční sada je označována za komplexní proto, že nabízí instrukce složitější než jednoduché aritmetické a logické operace a umožňuje používat operandy přímo z paměti.
- *RISC* - procesory s redukovanou instrukční sadou[3]. Tyto procesory naopak nabývají na popularitě v dnešní době. Oproti *CISC* kategorii však nabízejí pouze základní instrukce, které je nutné kombinovat pro dosažení složitějších efektů. Navíc jsou často také limitovány v tom, že operandy těchto instrukcí musí být v registrech procesoru a ne v paměti. Toto všechno však redukuje složitost logiky procesoru.

Již z názvu procesoru je zjevné, že Codix RISC patří do druhé z těchto kategorií.

2.2 LLVM

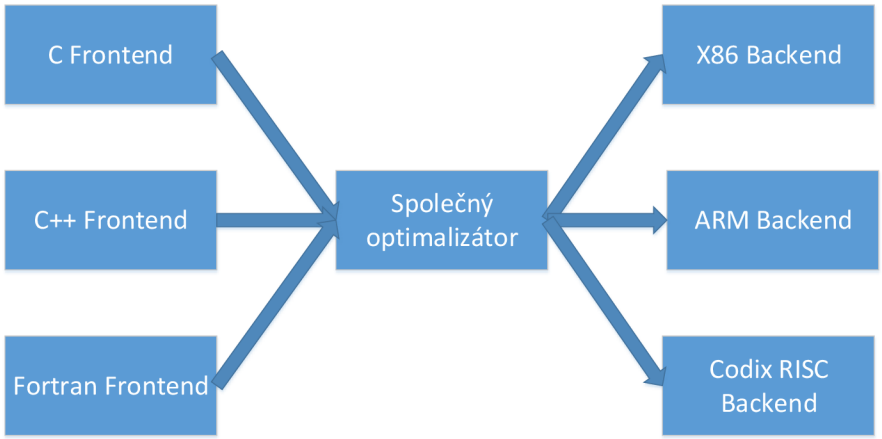
LLVM (původně odvozeno z *Low Level Virtual Machine*) je název projektu, který zastřešuje vývoj sady nástrojů jako je assembler, překladač a debugger. Vývoj započal v roce 2000, kdy již existovaly jiné alternativy pro překlad jazyků jako je C a C++ (jednou z nich je překladač GCC). Důvodem, proč se i přesto překladač LLVM prosadil je jeho vnitřní architektura. Většina tehdy existujících alternativ měla monolitickou strukturu. Z toho vyplývá, že bylo velmi složité využít jen část překladače pro jiné účely. Klasická struktura překladače se skládá ze tří částí, proto se jí také říká třífázová struktura [4] (naznačeno na obrázku 2.2):

- *Frontend*[4] - stará se o syntaktickou analýzu zdrojového kódu, kontrolu chyb a budování reprezentace zdrojového kódu, která se nazývá *abstraktní syntaktický strom*. Tento strom může být ještě převeden na speciální reprezentaci kódu určenou pro optimalizátor.
- *Optimalizátor* - provádí transformace kódu za účelem jeho zefektivnění.
- *Backend*[5] - často také zvaný generátor kódu, generuje strojový kód na základě instrukční sady cílové platformy.

V případě architektury překladače LLVM jsou tyto tři části na sobě nezávislé. Je nutné zajistit, aby *frontend* produkoval validní kód pro optimalizátor a také aby byl *backend* schopen zpracovat výstup optimalizátoru. Pokud je toto zajištěno je možné tvořit různé kombinace *frontendů* a *backendů*, které sdílí stejný optimalizátor. Takto je možné přidat podporu nového programovacího jazyka jen implementací nového *frontendu* a zajištění podpory nové cílové platformy znamená implementaci nového *backendu*. Pokud by tyto části fungovaly jako nedělitelný celek, bylo by nutné vždy implementovat celý překladač.

Právě proto je LLVM velmi vhodný pro použití ve vestavěných systémech. Společnosti zabývající se těmito zařízeními často disponují širokým spektrem procesorů s různou architekturou a pro každý z nich potřebují generovat jiný strojový kód. Při použití LLVM jako

[2] *Complex instruction set computer*
[3] *Reduced instruction set computer*
[4] Přední část
[5] Zadní část



Obrázek 2.2: Třífázová struktura LLVM[6]

překladače je dostačující mít pro každou architekturu vlastní *backend* překladače a zbývající části mohou být společné pro všechny procesory. Stejným způsobem je modularita LLVM využita v případě rozšíření instrukční sady procesoru. Tato úprava taktéž vyústí pouze v generování nového *backendu*.

2.3 Codasip Framework

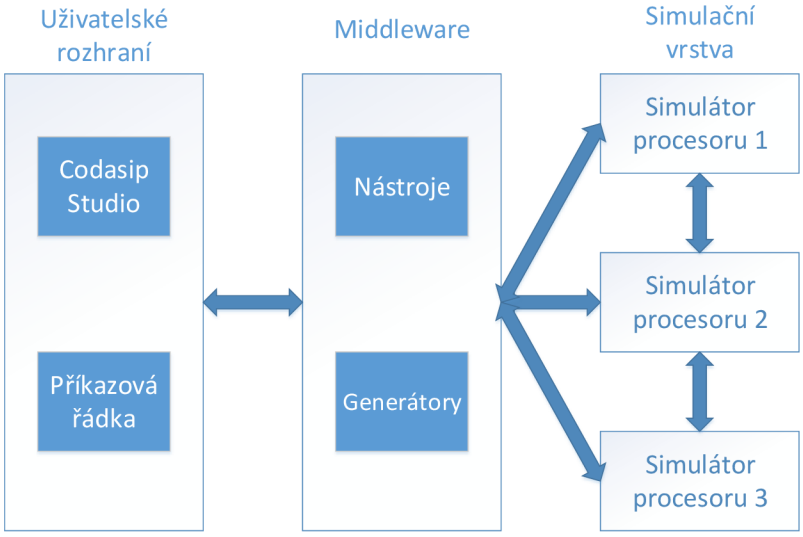
Veškerá práce popsaná v tomto dokumentu, byla vykonána za účelem ověření a rozšíření schopností nástrojů společnosti Codasip, které se sdružují do sady nazvané Codasip Framework. Tyto nástroje společně vytvářejí integrované vývojové prostředí, které se skládá ze tří vrstev [5] (obrázek 2.3):

- *Uživatelské rozhraní* - slouží k zprostředkování služeb *frameworku* uživateli. Od uživatele přijímá příkazy a následně zobrazuje informace o průběhu a výsledcích požadovaných operací. Je k dispozici ve dvou verzích: textové (příkazová řádka) a grafické (Codasip Studio).

- *Middleware* - server, který zpracovává příkazy od uživatelského rozhraní, se kterým komunikuje přes *TCP/IP* protokol. Může se jednat o služby, které je schopen vykonat sám, mezi které se řadí generování nástrojů. Dále také umožňuje komunikaci uživatele se simulátory procesorů a díky tomu například nastavení parametrů simulace. Komunikace však probíhá oboustranně a to z toho důvodu, aby mohl *middleware* poskytovat uživateli informace o výsledku požadovaných operací.

[6] Obrázek převzat a upraven z [4]

- *Simulační vrstva* - spravuje simulátory procesorů, přičemž umožňuje i multiprocesorovou simulaci.



Obrázek 2.3: Vrstvy Codasip Frameworku[7]

Codasip Studio je grafické rozhraní Codasip Frameworku, založené na velmi populárním vývojovém prostředí Eclipse. Kromě přístupu ke službám dalších nástrojů z *frameworku* obsahuje také editor s rozpoznáním syntaxe zdrojových kódů v jazyce CodAL (viz kapitola 2.3.1), který se pro návrh *ASIP* čipů Codasip používá [6].

2.3.1 CodAL

CodAL je jazyk spadající do rodiny *ADL*[8] jazyků [7]. Tyto jazyky slouží k popisu hardwarových komponent, především aplikačně specifických procesorů (ty jsou popsány v kapitole 2.1). Tyto procesory většinou tvoří základ tzv. systémů na čipu, kde kromě samotného procesoru nalezneme i další komponenty jako jsou paměti a sběrnice, které je také možné pomocí jazyka CodAL popsat. Jedná se o jazyk, který je svou syntaxí velmi podobný jazyku C. Jazyk CodAL spadá do kategorie jazyků, které jsou vhodné pro současný vývoj hardware a software, což znamená, že potřebné nástroje pro programování a simulaci procesoru jsou generovány přímo z jeho popisu. Návrháři tedy stačí vytvořit pouze jeden popis, aby získal všechny potřebné nástroje a zároveň popis procesoru v jazyce VHDL nebo Verilog.

[7] Obrázek vytvořen na základě [5]
[8] *Architecture description language*

Každý popis v jazyce CodAL se skládá z těchto částí [8]:

- *Hlavička modelu* - definuje, o jaký typ popisu se jedná. Existují dvě možnosti, model s přesností na instrukce[9] a model s přesností na cykly[10].
- *Popis zdrojů* - popis hardwarových dispozic procesoru, jako jsou registry, rozhraní paměti a jiné.
- *Popis instrukcí a událostí* - událostmi je chápáno chování procesoru ve specifických situacích jako je stav *halt* nebo *reset* procesoru. Dále je nutné, aby každý popis procesoru obsahoval alespoň jednu instrukci. Kromě standardních skalárních instrukcí je možné v jazyce CodAL zapsat také vektorové instrukce, což je kritická vlastnost v rámci této práce a bude podrobněji popsána v následujících kapitolách.

Každá instrukce definovaná v modelu procesoru obsahuje několik částí. Nejprve se jedná o zdroje, které bude instrukce využívat, jako jsou registrové operandy. Poté je to sekce *assembleru*. Ta obsahuje textovou reprezentaci instrukce, která je využita pro generování *assembleru* a *disassembleru*. Třetí částí je binární podoba instrukce, která reprezentuje instrukci ve strojovém kódu. Poslední částí je sémantika instrukce, kde je ve stylu velmi podobném jazyku C popsáno chování instrukce. Všechny tyto části jsou viditelné v ukázce kódu 2.1.

Vložený kód 2.1: Ukázka definování instrukce LUI v jazyce CodAL

```
element instruction_lui
{
        use gpreg as  reg_dst;
        use uimm16;

        assembler { "LUI" reg_dst "," uimm16 };
        binary { OPC_LUI:6 reg_dst 0b00000 uimm16 };

        semantics
        {
                regs[reg_dst] = uimm16 << 16;
        };
}
```

2.4 OpenCV

OpenCV je softwarová knihovna obsahující především algoritmy strojového učení a počítačového vidění. Je vydána pod licencí BSD a jedná se tedy o *open source* software [9]. Proto je možné nahlédnout do zdrojových kódů knihovny a případně je i upravovat.

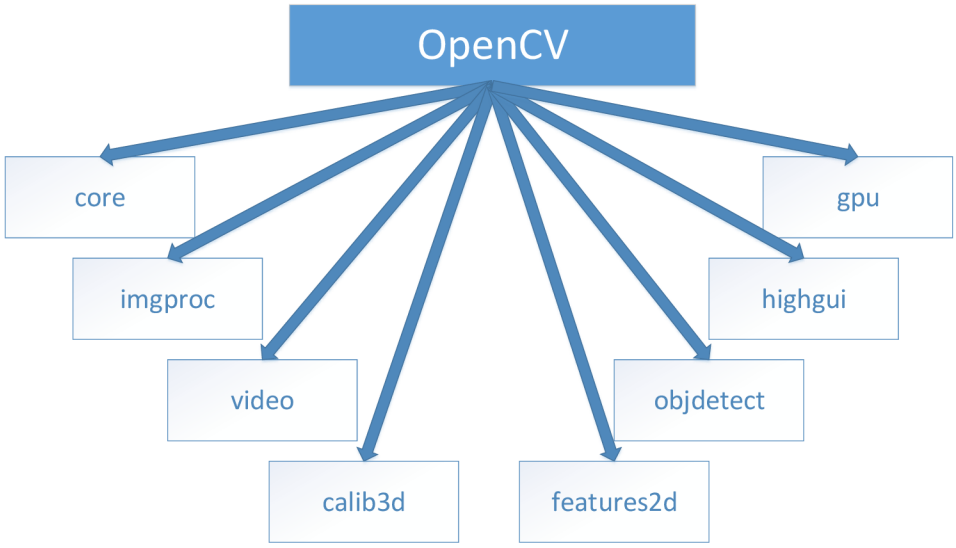
Kořeny knihovny sahají ke studentům elitních univerzit jako je MIT ve Spojených státech amerických, kde si studenti předávali hotové implementace algoritmů tak, aby noví studenti nemuseli vše implementovat od začátku a mohli své projekty budovat na základech předchozích studentů. Následně vývoj OpenCV přejala společnost Intel s několika cíli:

[9] *Instruction accurate*
[10] *Cycle accurate*

- Pokročit ve výzkumu počítačového vidění pomocí kódu, který bude nejen *open source*, ale také optimalizovaný.
- Popularizovat vývoj v této oblasti tím, že vývojáři budou mít v podobě této knihovny výchozí bod pro své aplikace a jejich kód se tímto stane lépe čitelným a přenositelným.
- Pomoci vývoji komerčních aplikací zaměřených na zpracování obrazu tím, že bude knihovna nabízena zdarma.

Poslední bod je pravděpodobně tím nejdůležitějším. Díky tomu, že je knihovna šířena zdarma a je možné pomocí ní vytvářet komerční produkty, které nemusí být vydány pod *open source* licencí, vzniká větší množství těchto poměrně náročných aplikací a tím se zvyšuje i poptávka po výkonných procesorech. Tímto Intel, v pozici významného producenta procesorů, nejspíše dosáhl větších zisků než kdyby OpenCV nabízel jako placený software [10].

V dnešní době již vývoj knihovny nespadá pod Intel, ale byl přebrán společností Itseez [9]. Uživatelská základna OpenCV čítá přibližně 20000 členů a samotná knihovna obsahuje přes 2500 optimalizovaných algoritmů a podporuje všechny tři největší počítačové operační systémy (Windows, MacOs, Linux) a také dva největší mobilní operační systémy (Android, iOS). Ačkoliv je knihovna nativně napsána v jazyce C++, disponuje kromě něj také rozhraním pro jazyky: C, Python, Java a MATLAB. Knihovna OpenCV je modulární. To znamená, že je rozdělena na osm modulů, které obsahují sady algoritmů zaměřených na konkrétní oblast uplatnění. Výjimku tvoří modul *core*, který obsahuje definice datových struktur, které jsou využívány všemi ostatními moduly. Jednotlivé moduly jsou vyjmenovány na obrázku 2.4.



Obrázek 2.4: Modulární struktura knihovny OpenCV

Prakticky všechny obsažené algoritmy lze rozčlenit do dvou skupin [10]:

- *Algoritmy vytvářející novou reprezentaci (transformační algoritmy)* - tyto algoritmy přijmou vstupní obraz a celý jej zpracují s cílem vytvořit nový upravený obraz. Jako příklad může posloužit převod barevného obrazu do obrazu v odstínech šedi.

- *Rozhodující algoritmy (klasifikační algoritmy)* - tyto algoritmy analyzují vstupní obraz s účelem rozhodnout, zdali obsahuje hledaný předmět. Příkladem může být detekce osob v obraze.

2.5 Standardní knihovna jazyka C++

Zhruba do roku 1997 probíhal proces standardizace jazyka C++. Součástí tohoto procesu byla i snaha o vytvoření standardní knihovny pro tento programovací jazyk. Hlavní náplní tohoto snažení bylo sesbírání často používaných algoritmů, které již byly implementovány a jejich následné spojení do jednoho celku knihovny. Je tedy patrné, že různé části knihovny byly implementovány jiným přístupem a nabízí jiná rozhraní a případně jiné úrovně kontroly logických chyb. V současné době standardní knihovna nabízí [11]:

- komponenty pro vstup/výstupní operace,
- podporu řetězců (string),
- datové struktury,
- efektivní implementaci řady algoritmů pro řazení, vyhledávání a jiné,
- podporu pro mezinárodní použití (různé znakové sady).

Existuje několik implementací standardní knihovny z toho důvodu, že vývoj standardní knihovny je často součástí vývoje samotného překladače jazyka. Následuje krátký popis několika z nich:

- *libc++* - knihovna vyvíjená pro překladač LLVM (viz kapitola 2.2). Knihovna je určena pro standard jazyka *C++11*. Do vytvoření této knihovny využíval překladač LLVM knihovnu libstdc++. Ta ovšem v nové verzi změnila licencování, což zabránilo vývojářům LLVM pokračovat v její modifikaci pro tento překladač [12].
- *libstdc++* - knihovna vyvíjená pro překladač GCC. Jedná se o *open source* projekt, do kterého mohou přispívat vývojáři z celého světa, přesto knihovna stále není kompletně dokončena [13].
- *stdcxx* - knihovna také známá jako *Apache C++ Standard Library*. Knihovna vznikla v roce 2005 na základě komerční knihovny společnosti *Rogue Wave Software*. Cílem této knihovny je podpora co nejširšího spektra hardware, operačních systémů a překladačů. Zároveň se snaží vytěžit co největší efektivitu ze specifických vlastností cílové platformy [14].

2.6 Optimalizační metody

Proces optimalizace je činnost, při které dochází k úpravě programu za účelem vytvoření ekvivalentního programu, který bude fungovat efektivněji nebo využívat méně zdrojů. Optimalizace je jedním z hlavních bodů této práce. Existuje spousta technik optimalizace software a také je možné je rozdělit podle několika kritérií. Cílem této kapitoly není kompletní výčet existujících nebo používaných optimalizačních technik, ale vysvětlení těch, které jsou dále zmiňovány v textu této práce.

Jelikož optimalizace knihovny OpenCV pro procesor, který může být pro toto použití upraven, spadá do metody zvané *Hardware/Software Codesign*[11] je možné rozdělit optimalizace následujícím způsobem:

- *Hardwarové* - tento způsob optimalizace je možné využít v případě, kdy má vývojář možnost ovlivnit hardware na kterém bude software spuštěn. V takovém případě může pro zefektivnění aplikace zvolit vhodnější hardware případně jej sám navrhnout pro jeho konkrétní účel použití.
- *Softwarové* - tyto optimalizace jsou nejčastěji používány. Ve většině případů se jedná o různé transformace zdrojového kódu programu. Velká část softwarových optimalizací je dnes automaticky prováděna při překladu programu, ale vývojář může některé z nich použít ve svém zdrojovém kódu i ručně.

Další velmi často používané rozdělení optimalizací je založeno na jejich přenositelnosti:

- *Platformě nezávislé* - tyto optimalizace jsou použitelné na většině, případně všech, platformách. To znamená, že je překladač může provádět bez ohledu na to, na jakém hardware bude výsledný program spuštěn. Tato obecnost optimalizací však může mít za následek jejich menší efektivitu.
- *Platformě závislé* - tato kategorie optimalizací je závislá na platformě na které bude spouštěn výsledný program, jelikož se snaží využít právě platformě specifických vlastností. Platformou může být například architektura použitého procesoru, případně dokonce konkrétní typ procesoru. Daná optimalizace tedy nemůže být v některých případech použita nebo musí být její výsledek uzpůsoben použité platformě.

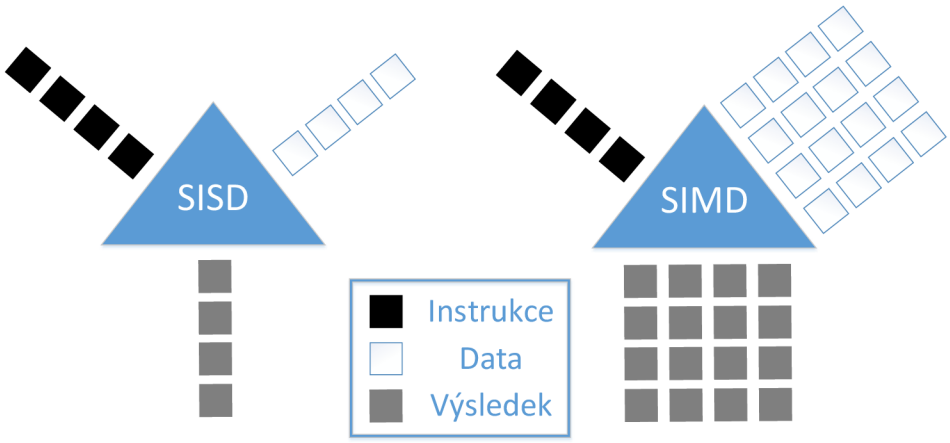
Mezi optimalizacemi, které jsou dále zmiňovány, je možné najít techniku zvanou *inlining*. Tato optimalizace spočívá v nahrazení volání funkce tělem volané funkce. Takto není nutné provádět rutiny obstarávající volání funkce a tím je ušetřen čas. Negativem použití techniky *inliningu* je to, že vkládání těla funkce do kódu zvětšuje velikost zdrojového kódu. Tato technika spadá do softwarových optimalizací a je také platformě nezávislá.

Naopak použití vektorových instrukcí je optimalizace, pro kterou je nutné mít hardwarovou podporu, která není přítomna na všech procesorech, a proto je tato technika platformě závislá.

2.6.1 SIMD

Zkratka *SIMD* je odvozena z anglického názvu *single instruction, multiple data*[12]. Jedná se o jednu z kategorií Flynnovy taxonomie, konkrétně specifikuje počítačovou architekturu, která je schopna vykonávat stejnou operaci na větším počtu vstupních operandů (nebo párech operandů) ve stejnou dobu [15]. Tento přístup je výhodný především pro multimediální aplikace, jako je zpracování obrazu. Zde je často nutné aplikovat stejnou operaci na celý vstupní obraz. Takto není nutné provádět zpracování postupně po jednom pixelu, ale je možné zpracovat několik pixelů ve stejný okamžik. Podmínkou pro využití této techniky je to, že šířka registrů použitého procesoru je větší než šířka operandů prováděné operace. Rozdíl mezi klasickým zpracování *Single instruction, Single data stream (SISD)*, kdy je daná operace prováděna nad jedním operandem (případně jedním párem operandů) a technikou *SIMD* je znázorněna na obrázku 2.5.

[11] Současný vývoj hardware a softwar
[12] Jedna instrukce, více dat



Obrázek 2.5: Rozdíl mezi zpracováním SISD a SIMD[13]

Původně byl přístup *SIMD* vynalezen pro použití na rozsáhlých paralelně pracujících přístrojích, kde každý operand byl zpracováván jiným procesorem, který přistupoval ke své vlastní paměti [16]. V dnešní době je tato technika zakomponována přímo do procesorů obecného použití a logika vykonávající tyto operace je tedy na stejném čipu jako zbytek procesoru. Proto se úzkým místem, které určuje úroveň paralelismu, stává sběrnice vedoucí od procesoru k paměti nebo případně šířka *cache* samotného procesoru. Pokud tedy procesor disponuje 64bitovou sběrnicí, může provádět zároveň dvě 32bitové operace nebo například osm 8bitových paměťových operací.

SIMD operace jsou na současných procesorech zpřístupněny pomocí vektorových instrukcí často označovaných jako *SIMD* instrukce. Tyto instrukce se od klasických skalárních instrukcí liší tím, že operandy nepovažují za nedělitelný prvek, ale jsou chápány jako vektor několika prvků, kdy je na každý z těchto prvků nutné aplikovat instrukcí specifikovanou operaci. Nejpoužívanější sady *SIMD* instrukcí byly navrhnuty pro procesory značky Intel. Nejprve se jednalo o řadu *MMX*[14] následně o řady *SSE*[15] [17]. Ze začátku bylo nutné, aby Intel dodával i překladač jazyka C, který obsahoval rozšíření pro využití těchto instrukcí. Programátor má možnost začlenit *SIMD* instrukce do svého kódu pomocí tzv. *intrinsics* funkcí. Tyto funkce jsou do kódu začleněny *inline* (tato technika byla vysvětlena dříve v této kapitole) a překladačem přeloženy jedna ku jedné na některou z vektorových instrukcí [18]. Podobného efektu by bylo možné dosáhnout i pomocí *inline assembleru*[16]. Ovšem jakékoliv vkládání assembleru do kódu jazyka vyšší úrovně často zabrání v dalších optimalizacích kódu. Efekty *intrinsics* funkcí jsou překladači známy a proto může s tímto kódem provádět další optimalizace.

Bohužel začlenění speciálních vektorových instrukcí do programu snižuje jeho přenositelnost. Je nutné aby použité *intrinsics* byly známé použitému překladači a také aby architektura procesoru, na kterém program poběží, podporovala použité vektorové instrukce. Intel nabízí popis v pseudokódu všech *SIMD* instrukcí, které používá na svých procesorech. Díky tomu je případně možné nahradit použité *intrinsics* funkce a tedy následně vektorové

[13] Obrázek přejat a upraven z https://www.numscale.com/boost-simd/
[14] *Multimedia Extension*
[15] *Streaming SIMD extension*
[16] Vkládaného assembleru

instrukce za ekvivalentní implementaci v použitém programovacím jazyce. Tuto implementaci poté překladač přeloží do méně efektivní posloupnosti jednoduchých instrukcí. Kód 2.2 ukazuje dostupný popis jedné z těchto instrukcí. Popis chování vektorových instrukcí je také možné využít při vývoji procesoru, který bude s určitou sadou *SIMD* instrukcí kompatibilní.

Vložený kód 2.2: Popis SIMD instrukce pro pravý shift 32bitových celých čísel[17]

```
FOR j := 0 to 3
        i := j*32
        IF count[63:0] > 31
                dst[i+31:i] := SignBit
        ELSE
                dst[i+31:i] :=
                        SignExtend(a[i+31:i] >> count[63:0])
        FI
ENDFOR
```

2.6.2 Profilace

Profilace není sama optimalizační technikou, ale s optimalizacemi úzce souvisí. Pomocí profilace je totiž možné zjistit, kterou optimalizaci má smysl na daný program použít.

Profilace je technika, která na základě vykonání programu nebo jen na základě jeho syntaktické analýzy[18] určí cenu vykonání jednotlivých algoritmů, procedur a funkcí ve smyslu času stráveného jejich výpočtem. Dále je pomocí profilace možné zjistit počet volání jednotlivých funkcí a také hardwarové nároky programu, nejčastěji množství využité paměti [20]. Takto je možné najít v kódu místa, která jsou vykonávána nejčastěji, případně jejich výpočet zabírá nejvíce času a následně soustředit optimalizace na tyto části kódu.

Existují dva základní přístupy k profilaci [19]:

- *Statická profilace* - při statické profilaci dochází pouze k analýze zdrojového kódu a není nutné spuštění programu. Při této analýze dochází k *parsování* kódu a budování struktury zvané *Control Flow Graph*. V této struktuře jsou následně propočteny ceny jednotlivých operací ve zdrojovém kódu a díky tomu je možné vypočítat i cenu cesty procházející vytvořeným grafem. Tato profilace tedy hledá nejhorší možné hodnoty času výpočtu a využití prostředků, které mohou při vykonání programu nastat.

- *Dynamická profilace* - dynamická profilace analyzuje program při samotném výpočtu. Spousta potenciálně zkoumaných programů však pro svou činnost vyžaduje nějaké vstupy. Rozdílné vstupy však mohou způsobit rozdílné chování programu. Je tedy nutné vybrat testovací vstupní hodnoty pečlivě, aby bylo chování programu reprezentativní i pro další potenciální vstupní hodnoty. Často je dynamická profilace prováděna *vzorkováním*. Kontrolovat, jaká část kódu je vykonávána nepřetržitě by bylo velmi náročné a profilace by takto zabrala velké množství času. Proto kontrola probíhá jen v pravidelných intervalech.

[17] Kód přejat z https://software.intel.com/sites/landingpage/IntrinsicsGuide/
[18] Často označované původně anglickým termínem *parsování*

2.7 Existující řešení

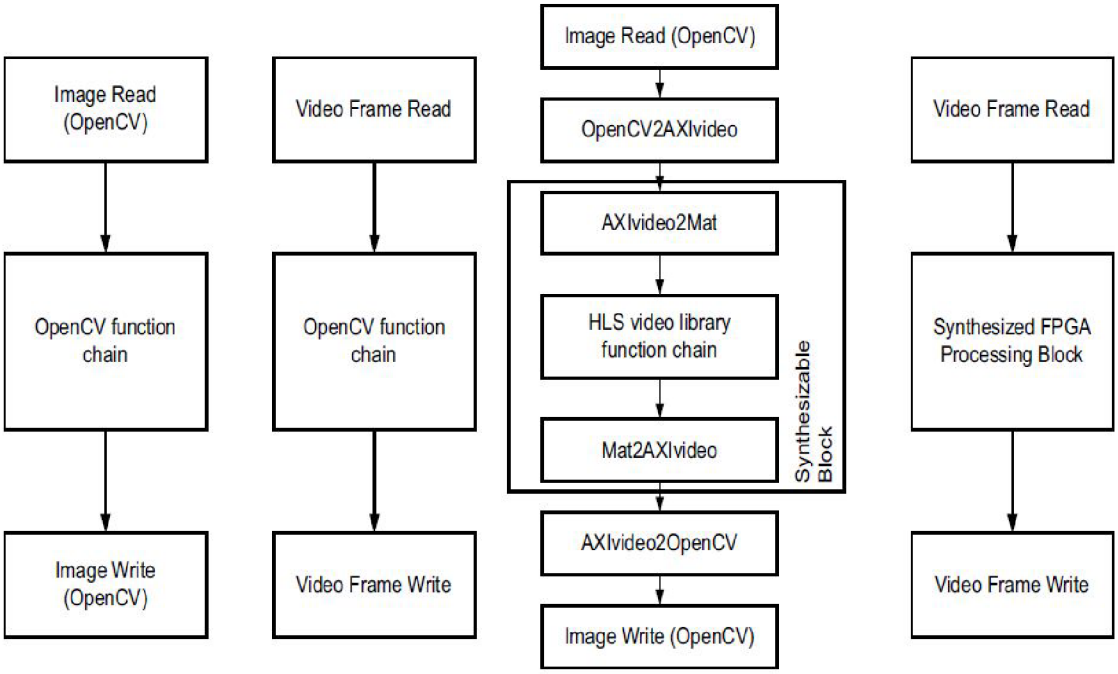
Codasip není jedinou společností na trhu, která se zajímá o návrh elektronických komponent a automatizaci této činnosti. Stejně jako pro ni se tak i pro konkurenční společnosti stalo nutností disponovat implementací knihovny OpenCV přeložitelnou a optimalizovanou pro jejich aplikačně specifické procesory. Následuje výčet již existujících řešení této problematiky.

Cadence Tensilica Cadence je společnost zabývající se automatizací návrhu elektronických komponent. Nabízí služby pro společnosti vyvíjející tzv. systémy na čipu (*SoC*). V portfoliu Cadence je možné najít procesorová jádra Tensilica, která se řadí do kategorie *ASIP* (více v kapitole 2.1). Díky softwarovým nástrojům, které tato společnost také vyvíjí, je tedy možné tato jádra rozšířit a vytvořit tak procesor uzpůsobený pro konkrétní aplikaci. Na tomto jádře jsou založeny i procesory IVP a IVP-PE, které díky specializované instrukční sadě obsahující i *SIMD* instrukce (viz kapitola 2.6.1), výrazně zrychlují aplikace zaměřené na zpracování pixelů. Oba tyto procesory jsou optimalizovány pro operace nad osmi, šestnácti a třiceti dvěma pixely [21]. Tyto procesory jsou svou architekturou vhodné pro aplikace založené na OpenCV a proto byla tato knihovna právě pro procesor IVP portována [22].

CEVA-MM3101 Ceva je další ze společností zabývajících se návrhem elektronických komponent se specializací na digitální signálové procesory (*DSP*). V portfoliu této společnosti je možné najít procesory pro zpracování zvuku, procesory zaměřené na komunikační technologie (3G, LTE, Bluetooth), ale také procesory pro zpracování obrazu. Právě do poslední kategorie se řadí procesor MM3101. Tento procesor je určen především pro zařízení osazená kamerou, u kterých je kritická spotřeba energie. Disponuje technologiemi jako je vektorová jednotka umožňující zpracování *SIMD* instrukcí a také je pro tento procesor vytvořena knihovna CEVA-CV, která obsahuje více než 750 pro tento procesor optimalizovaných algoritmů z knihovny OpenCV. Kromě toho Ceva nabízí také již vytvořené aplikace pro stabilizaci obrazu, rozpoznávání gest a obličejů a jiné [23].

Xilinx Zynq Xilinx je největším světovým výrobcem programovatelných hradlových polí[19] (*FPGA*). Jednou z řad systémů na čipu, které tato společnost produkuje je řada Zynq [24]. Jedná se o systém obsahující jak obecný procesor architektury ARM tak také programovatelné hradlové pole. Právě pro tento systém z portfolia Xilinxu je možné optimalizovat algoritmy z knihovny OpenCV. Samotnou aplikaci napsanou s použitím knihovny OpenCV je možné přeložit pro architekturu ARM. Ovšem tím není docíleno zrychlení oproti standardním procesorům. Důležité je využití *FPGA*. K tomu existuje Xilinxem vyvíjená knihovna Vivado HLS[20]. Ta obsahuje podmnožinu algoritmů OpenCV zachovávající stejné rozhraní, ovšem samotný algoritmus je možné syntetizovat právě do hradlového pole na čipu Zynq. Tím, že takto vznikne čistě hardwarová implementace je zvýšena efektivita algoritmu a urychleno jeho vykonávaní. Je zjevné, že se toto neobejde bez změny zdrojových kódů původní OpenCV aplikace. Kromě nahrazení cílových funkcí za jejich HLS ekvivalenty je také nutné zajistit platformě závislé přístupy ke vstupnímu a výstupnímu obrazu. Tento proces je graficky naznačen na obrázku 2.6 [25].

[19] *Field Programmable Gate Array*
[20] *High-level synthesis*



Obrázek 2.6: Postupná optimalizace OpenCV algoritmu pomocé Vivado HLS[21]

[21] Obrázek přejat z [25]

Kapitola 3

Návrh řešení

Následná kapitola se zabývá návrhem řešení, tedy mou představou toho, jakým směrem se bude ubírat samotná implementace před tím, než jsem jí započal. Také jsou zde zdůvodněna některá rozhodnutí, která byla v rámci specifikace zadání ponechána na mě.

3.1 Knihovna Apache C++

Mít k dispozici přeloženou standardní knihovnu jazyka C++ bylo pro dokončení celé práce nezbytné a to nejen proto, že je to jedním z bodů zadání práce. Naprostá většina knihoven a nástrojů implementovaných ve standardních programovacích jazycích vyžaduje pro svůj překlad nejen překladač příslušného jazyka, ale právě také standardní knihovnu a OpenCV v tomto není žádnou výjimkou. Nástroje Codasip Frameworku umožňují vygenerování samotného překladače, ale vygenerování standardní knihovny samozřejmě není možné. Standardní knihovnu je naopak nutno dodat, aby mohla být přidána do exportovaného toolchainu a následně využívána při vývoji C++ aplikací pro danou platformu.

Základní podmínkou při výběru vhodné implementace standardní knihovny bylo její vydání pod *open source* licencí. Vybral jsem Apache C++ Standard Library (kódové označení *stdcxx*, další alternativy jsou zmíněny v kapitole 2.5). Tato knihovna, je vydána pod *Apache License*, která je *open source* licencí [26]. Kromě toho však tato knihovna disponuje dalšími vlastnostmi, které ji činí lépe použitelnou pro tento účel oproti jiným implementacím. Cílem vývojářů bylo totiž vytvořit knihovnu, která splňuje mezinárodní standard ISO/IEC 14882, ale zároveň je maximálně efektivní na jakékoliv platformě, pro kterou je možné knihovnu přeložit. Toho docílí využitím platformě specifických vlastností, které se pojí jednak se samotným hardwarem, ale také s operačním systémem, případně překladačem. Zároveň je knihovna schopna fungovat i v případě, že překladač C++ pro danou platformu nepodporuje všechny standardní funkce jazyka C++. To je velmi užitečné v případě využití knihovny pro vestavěné systémy, kde překladače nemusí podporovat některá specifika jazyka jednoduše z toho důvodu, že nejsou při programování pro tato zařízení potřeba [14].

Tato situace nastala i při překladu knihovny pro platformu Codix RISC, kdy překladač nepodporuje výjimky a typovou identifikaci za běhu[1]. Z tohoto důvodu musí být některé části zdrojových kódů knihovny implementovány v několika verzích lišících se vyžadovanými schopnostmi platformy. Při překladu knihovny je poté vybrána správná verze. Před samotným překladem je proto provedena sada testů, která zjistí specifika dané platformy. Testy

[1] *Run-Time Type Identification*

jsou vytvořeny jednoduchým způsobem, a sice že se pokusí vždy jednu vlastnost překladače použít a následně se ověří, zdali překlad proběhl v pořádku. Pokud ano, jsou některé testy následně i spuštěny a zkontrolovány výsledky jejich běhu. Souhrnným výstupem testů je configurační soubor, v podobě hlavičkového souboru, obsahující množství *#define* direktiv preprocesoru, které poté zastíní, případně zviditelní, kritické části zdrojových souborů. Tento přístup je velmi praktický, jelikož vyžaduje minimální úpravy zdrojových souborů a případně umožňuje ruční úpravou configuračního souboru zastínit některé schopnosti překladače pokud nejsou například ještě dostatečně otestovány.

Vložený kód 3.1: Část konfiguračního souboru Apache C++ standardní knihovny

```
#define _RWSTD_CLOCK_T          unsigned long
#define _RWSTD_PTRDIFF_T        int
#define _RWSTD_SIZE_T           unsigned
#define _RWSTD_SIZE_MAX         _RWSTD_UINT_MAX
#define _RWSTD_RAND_MAX         2147483647
#define _RWSTD_EOF              -1
#define _RWSTD_WEOF             -1
```

3.2 Vybrané aplikace OpenCV

Jedním z kroků, které jsem musel učinit, byl výběr aplikací založených na knihovně OpenCV, které poslouží jako prezentace funkčnosti knihovny OpenCV na platformě Codix RISC. Nemohl jsem se však rozhodnout pro jakékoliv ukázky, jelikož samotná prezentace celého systému byla přesně zadána. Systémem je myšlen samotný čip, který vykonává ukázkovou aplikaci a dále kamera a monitor. Tento systém může sloužit při různých prezentacích firmy Codasip. Vstup aplikace tvoří obraz snímaný kamerou, který mohou ovlivnit samotní návštěvníci svým pohybem v záběru kamery. Je tedy nezbytně nutné, aby ukázková aplikace pracovala v reálném čase. Takto nebude vznikat velká prodleva mezi skutečnou událostí a obrazem na monitoru. Existuje velké množství demo aplikací pro OpenCV, některé z nich jsou dokonce popsány v oficiální dokumentaci. Ne všechny jsou však určeny k tomu, aby byly spouštěny cyklicky pro transformaci videa. I toto jsem tedy musel vzít v úvahu. Z předešlého textu je také patrné, že bude nakonec v praxi využita jen jedna z vybraných aplikací. Proto jsem se rozhodl vybrat jednu méně náročnou, u které je vysoký předpoklad toho, že se ji podaří zoptimalizovat tak, aby fungovala v reálném čase, a tedy bude zajištěno funkční *demo*. Druhá aplikace je tedy náročnější a snaží se narazit na limity platformy. To může být užitečné pro pozdější návrhy vylepšení platformy Codix RISC v případě, že se ukáže, že bez nich není možné dosáhnout zpracování obrazu v reálném čase.

3.2.1 Rozpoznání hran

Jako první a jednodušší aplikaci OpenCV jsem zvolil detekci hran[2]. Jedná se o algoritmus z řady transformačních algoritmů (vysvětleno v kapitole 2.4). Pro výběr toho algoritmu existují dva důvody:

- Aplikace pro detekci hran již byla jednou využita jako prezentace firmy Codasip. Ovšem jednalo se o prezentaci platformy Codix Stream, která je určena pro aplikace podobného typu. Codix RISC je procesor původně obecného zaměření, který je možný

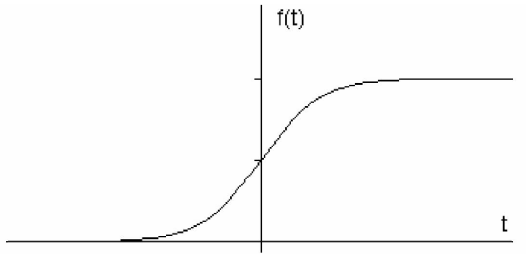
[2] *Edge detection*

uživatelsky rozšiřovat, takže by použití tohoto algoritmu umožnilo porovnání dvou různých platforem.

- Detekce hran je naprosto základní algoritmus zpracování obrazu. Je součástí složitějších aplikací, především detektorů objektů. Pokud se tedy někdo rozhodne OpenCV využít, je velmi pravděpodobné, že detekce hran bude minimálně jedním z kroků jeho algoritmu. Existence optimalizované verze tohoto algoritmu pro danou platformu je tedy samozřejmě výhodou.

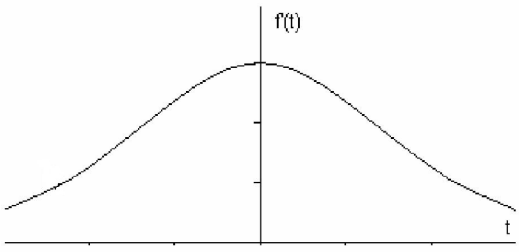
Existuje množství různých algoritmů pro detekci hran a několik z nich je implementováno i v knihovně OpenCV. Já jsem zvolil algoritmus známý pod jménem *Sobel*. Jedná se o algoritmus z řady algoritmů pro detekci hran založených na první derivaci. Oproti algoritmům pracujících s druhou derivací tyto algoritmy více chybují při práci s obrazem, který obsahuje šum. Ovšem jedná se o jednodušší algoritmy z hlediska implementace. To znamená, že nejsou tak výpočetně náročné a také je zde větší prostor pro optimalizace. Dále krátce vysvětlím princip algoritmu.

Jak je z názvu kapitoly jasné, algoritmus *Sobel* vyhledává v obraze hrany. Hranu je možné definovat jako skokovou změnu intenzity v obraze (viz obrázek 3.1).



Obrázek 3.1: Hrana jako skoková změna intenzity[3]

Pokud provedeme první derivaci podle t, získáme *gradient* (viz obrázek 3.2). Nejvyšší místo gradientu označuje centrum hrany. Algoritmus funguje tak, že označí daný pixel jako hranu, pokud hodnota gradientu přesáhne určitý práh.



Obrázek 3.2: Gradient[3]

K detekování gradientu dochází pomocí aplikování konvolučních filtrů na obrázek. Filtry jsou dva, přičemž druhý z nich je pouze o 90 stupňů otočený první a to proto, že jeden slouží k detekování gradientu v horizontální ose a druhý ve vertikální. Příklad takovýchto filtrů je k vidění na obrázku 3.3.

[3]Obrázek přejat z [27]

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

Gx

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

Gy

Obrázek 3.3: Konvoluční filtry pro horizontální a vertikální směr[3]

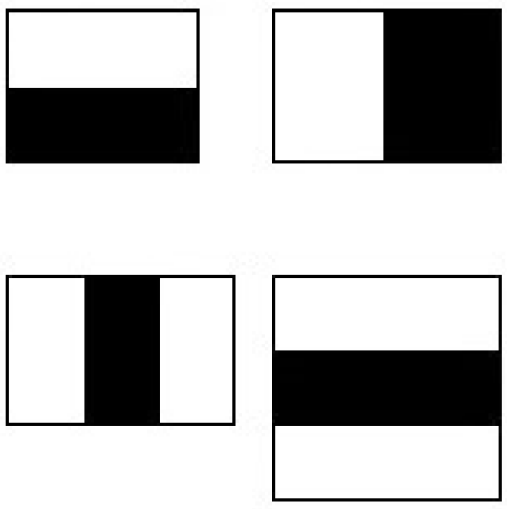
Výsledky obou filtrů jsou pak složeny a tak je nalezena absolutní velikost gradientu a jeho směr v každém bodě obrázku [27].

3.2.2 Rozpoznání obličejů

Druhý a komplikovanější ukázkový algoritmus je detekce obličejů[4]. Patří především do druhé kategorie algoritmů pro zpracování obrazu a tedy k rozhodovacím algoritmům, jelikož je jeho smyslem rozhodnout, zdali je na obrázku člověk a následně nalézt jeho obličej. Částečně je možné jej však považovat i za transformaci obrazu, jelikož vstupní obraz rozšíří o rámeček okolo případného obličeje.

Pro detekci objektů v obraze bylo vynalezeno několik algoritmů. Ovšem přelomem se stal algoritmus z roku 2001, jehož autory jsou jsou Paul Viola a Michael Jones. Podle nich se algoritmus často označuje jako *Viola & Jones* algoritmus nebo také algoritmus *Haar cascade*, což je odvozeno z principu fungování algoritmu.

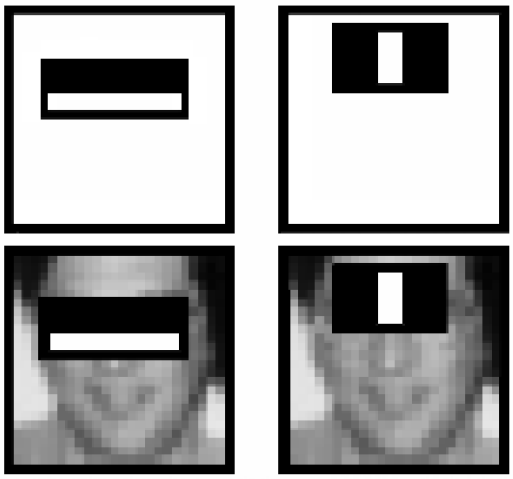
Jedná se o algoritmus založený na strojovém učení. To znamená, že je nejprve nutné natrénovat program na sedě vstupů, která obsahuje jak pozitivní příklady, tedy obrázky, na kterých se vyskytuje hledaný objekt, tak negativní příklady, kam patří obrázky, na kterých hledaný objekt není. Na obrázky je následně aplikována sada konvolučních filtrů, které jsou posouvány tak, aby byly zjištěny hodnoty konvoluce ve všech místech vstupního obrázku. Kromě toho jsou jednotlivé filtry aplikovány v různých velikostech. Příklady těchto filtrů jsou zobrazeny na obrázku 3.4.



Obrázek 3.4: Ukázka filtrů pro detekci hran a čar[5]

[4] *Face detection*

Při klasifikaci obrázku je následně možné z výsledku aplikování konkrétního filtru na konkrétní pozici v obrázku určit, že se jedná například o ústa na lidském obličeji. Ovšem většina kombinací *velikost filtru/pozice v obrázku* nevrací výsledky, které by prokazatelně pomáhaly určit, zdali je na obrázku hledaný objekt nebo není. Proto je následně pomocí algoritmu *Adaboost* vybrána jen sada těchto filtrů, které jsou použity při klasifikaci jako tzv. *slabé klasifikátory*. Konkrétně je nutné, aby každý z klasifikátorů měl větší úspěšnost než při náhodném hádání, tedy aby správně klasifikoval více než polovinu vstupů z dané množiny obrázků. Na obrázku 3.5 jsou ukázány právě dobře umístěné filtry v klasifikovaném obrázku. Pokud tedy budeme klasifikovat okno vstupního obrázku o velikosti $24x24$ pixelů, stačí po redukci pomocí *Adaboost* aplikovat jen 6000 klasifikátorů a ne původních více než 160000. Při klasifikaci se výsledky těchto slabých klasifikátorů skládají a tím se zvyšuje pravděpodobnost, že bylo správně rozhodnuto o přítomnosti nebo nepřítomnosti hledaného objektu ve vstupním obraze.



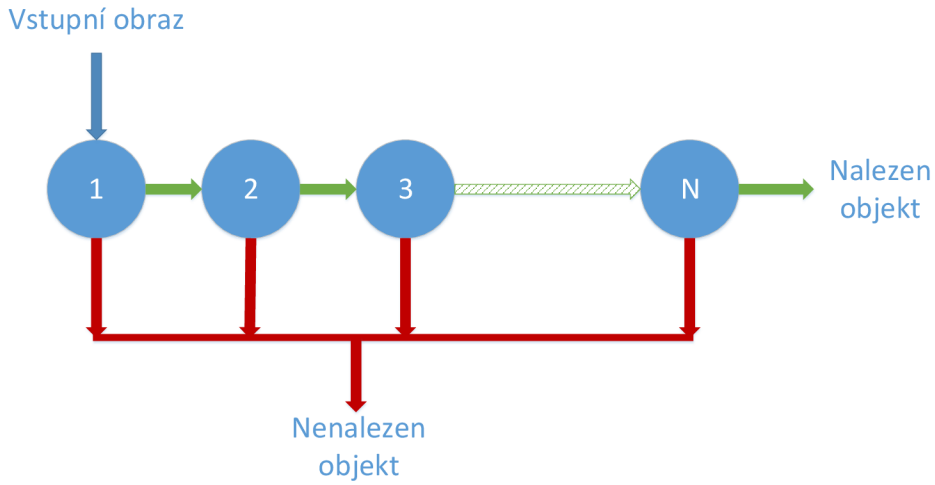
Obrázek 3.5: Ukázka smysluplně umístěných filtrů do obrázku lidského obličeje[5]

Ovšem i takto by bylo nutné provézt příliš mnoho výpočtů pro každý vstup. Větší část vstupního obrazu totiž často žádný obličej neobsahuje. Proto byl zvolen sofistikovanější přístup než aplikování celé sady klasifikátorů na každé okno vstupu. Klasifikátory jsou rozčleněny do skupin, které jsou následně seřazeny do kaskády (názorně ukázané obrázkem 3.6). Na vybrané okno obrázku je nejprve aplikována první skupina klasifikátorů z této kaskády, a pokud se již tehdy rozhodne, že se nejedná o obličej, nejsou další skupiny klasifikátorů vůbec použity [28].

3.3 Instrukční rozšíření

Hlavní náplní této práce je optimalizace vybraných algoritmů. Existuje spousta technik, které toto umožní, ovšem jednu z nich nabízí přímo implementace OpenCV. Výpočetně náročné části některých algoritmů, které jsou k tomu vhodné, byly přepsány pomocí *intrinsics* funkcí, aby byly následně překladačem přeloženy na *SIMD* instrukce ze sady SSE2, jak je tomu například i v ukázce kódu 3.2. Zdrojové kódy takto obsahují jak optimalizovanou

[5]Obrázek přejat z [28]



Obrázek 3.6: Systém kaskády skupin klasifikátorů

verzi, tak původní verzi a na základě přítomnosti instrukčního rozšíření na procesoru, pro který je OpenCV přeloženo, je vybrána vhodná verze. Procesor Codix RISC sadou *SIMD* instrukcí v základní verzi nedisponuje. Ale jak již bylo zmíněno, základní verze tohoto procesoru je pouze šablonou pro další vývoj a tedy je možné chybějící instrukce do modelu procesoru doimplementovat.

Vložený kód 3.2: Ukázka použití intrinsics funkcí v OpenCV

```
for( ; i <= width - 4; i += 4, src += 4 )
{
        __m128i f, s0 = z, x0, x1;

        for( k~= j = 0; k~< _ksize; k++, j += cn )
        {
                f = _mm_cvtsi32_si128(kx[k]);
                f = _mm_shuffle_epi32(f, 0);
                f = _mm_packs_epi32(f, f);

                x0 = _mm_cvtsi32_si128(*(const int*)(src + j));
                x0 = _mm_unpacklo_epi8(x0, z);
                x1 = _mm_mulhi_epi16(x0, f);
                x0 = _mm_mullo_epi16(x0, f);
                s0 = _mm_add_epi32(
                        s0, _mm_unpacklo_epi16(x0, x1));
        }
        _mm_store_si128((__m128i*)(dst + i), s0);
}
```

Kapitola 4

Implementace a výsledky

4.1 Překlad a integrace knihovny Apache C++

4.1.1 Překlad

Při překladu knihovny dochází ke generování konfiguračního souboru, kterým se následně řídí překlad knihovny. Tento soubor obsahuje direktivy preprocesoru, jejichž hodnota je nastavena na základě výsledků sady testů. Některé z těchto testů však vyžadují spuštění. To je nepraktické, jelikož uživatel překládající knihovnu nemusí mít k dispozici simulátor cílové platformy. Proto jsem vytvořil skript, který řídí překlad knihovny tak, aby simulace nebyla potřeba. Překlad se nyní skládá z těchto kroků:

1. Spuštění překladového systému s parametry pro pouhé generování konfiguračního souboru. Vygenerovaný soubor není správný. Jelikož není k dispozici simulátor, budou všechny testy simulaci vyžadující označeny za nefunkční a vlastnosti těmito testy kontrolované za nepodporované překladačem. Dojde však k vybudování adresářové struktury pro přeloženou knihovnu, která je nutná pro další kroky.

2. V nově vytvořené struktuře adresářů je nahrazen konfigurační soubor se špatnými hodnotami za předem připravený soubor s hodnotami správnými. Tento soubor je uchováván společně s tímto skriptem a je nutné jej vygenerovat znovu jen v případě, kdyby došlo k zásadním změnám v překladači.

3. Spuštění překladového systému knihovny s parametry pro překlad knihovny s použitím již existujícího konfiguračního souboru.

4. Instalace knihovny do specifikovaného adresáře.

Skript přijímá několik parametrů, kterými se specifikují cesty k potřebným nástrojům a také adresář pro instalaci knihovny. Skript jsem takto vytvořil proto, aby jej bylo následně možné použít při integraci překladu knihovny do generování *toolchainu*.

4.1.2 Integrace

Zprovoznění knihovny vyžadovalo jisté úsilí, a tudíž nedává smysl, aby byla použita jen pro účel překladu knihovny OpenCV. Součástí práce na zprovoznění standardní knihovny se tedy stala i snaha o její začlenění do Codasip Frameworku tak, aby byla generována společně s ostatními nástroji, stejně jako je tomu v případě standardní knihovny jazyka C.

Generování této knihovny probíhá při generování překladače v případě, že tuto možnost uživatel vybere v Codasip Studiu. Pro generování standardní C++ knihovny je využit stejný postup, při kterém dojde k zavolání skriptu popsaného v sekci 4.1.1 s potřebnými parametry.

Před integrací knihovny do frameworku jsem provedl testování. To bylo provedeno na testovací sadě vytvořené pro překladač LLVM. Tato testovací sada obsahuje testy zaměřené na jazyk C++, jelikož C++ je jedním z jazyků, který je možné překladačem LLVM překládat. Některé z testů však předpokládají, že překladač disponuje všemi standardními vlastnostmi. Vzhledem k tomu, že překladač pro Codix RISC některé vlastnosti jazyka C++ nepodporuje, bylo nutné vytřídit z testovací sady testy, které z tohoto důvodu selžou.

4.2 Floating point aritmetika

Pro obě ukázkové aplikace jsem vytvořil profil aplikace. Nejdůležitější informace, kterou je možné v tomto profilu nalézt je pořadí funkcí, které zabraly největší úseky času výpočtu. Na nejnáročnější funkce je následně soustředěno nejvíce úsilí při optimalizaci programu. V profilu těchto dvou aplikací jsem však nalezl jinou důležitou informaci. Nejvytíženějšími funkcemi jsou emulace *floating point* operací[1]. Je typické, že *ASIP* čipy nedisponují *FPU* jednotkou, což je jednotka, která hardwarově realizuje *floating point* operace. To ovšem neznamená, že programy spouštěné na těchto procesorech nesmí tyto operace provádět. *Run time* knihovna překladače pro tento účel obsahuje sadu funkcí, které realizují výpočty s pohyblivou desetinnou čárkou pomocí instrukcí pro výpočty s pevnou desetinnou čárkou, kterými procesor samozřejmě disponuje.

Pokud programátor použije operaci, kterou procesor umí vykonat, je tato operace přeložena na instrukci daného procesoru. Pokaždé když je ve zdrojovém kódu programu použita operace nepodporovaná hardwarově, překladač nahradí použití této operace za volání příslušné funkce z *run time* knihovny. Jako příklad uvádím operaci sčítání nad datovým typem *int* a *float*. Ač se na úrovni jazyka C jedná o stejný kód (viz úseky kódu 4.1 a 4.2, je v případě datového typu *int* přeložen na instrukci sčítání a v případě datového typu *float* na volání emulační funkce (viz úseky kódu 4.3 a 4.4).

Vložený kód 4.1: Sčítání s pevnou desetinnou čárkou v jazyce C

```
int main(void)
{
        int a = 1;
        int b = 3;
        int c;
        c = a + b;
        return c;
}
```

Vložený kód 4.2: Sčítání s pohyblivou desetinnou čárkou v jazyce C

```
int main(void)
{
        float a = 1.5;
        float b = 3.6;
        float c;
        c = a + b;
        return c;
}
```

[1] Operací s pohyblivou desetinnou čárkou

Vložený kód 4.3: Sčítání s pevnou desetinnou čárkou v assembleru

```
st zero, [ fp + -4 ]
r3 = or zero, +1&0xffff
st r3, [ fp + -8 ]
r3 = or zero, +3&0xffff
st r3, [ fp + -12 ]
r3 = ld [ fp + -8 ]
nop
nop
r3 = add r3, +3
st r3, [ fp + -16 ]
sp = or fp, zero
fp = ld [ sp + 0 ]
sp = add sp, +8
jump ra
```

Vložený kód 4.4: Sčítání s pohyblivou desetinnou čárkou v assembleru

```
st zero, [ fp + -4 ]
r4 = lui +1069547>>16 &0xffff
st r4, [ fp + -8 ]
r4 = lui +1080452>>16 &0xffff
r4 = or r4, +1080452&0xffff
st r4, [ fp + -12 ]
r5 = ld [ fp + -8 ]
nop
nop
r3 = or r5, zero
call $__addsf3
st r3, [ fp + -16 ]
call $__fixsfsi
sp = or fp, zero
ra = ld [ sp + +4 ]
fp = ld [ sp + 0 ]
sp = add sp, +8
jump ra
```

Toto řešení se netýká pouze operací s desetinnou čárkou. RISC procesory často nedisponují i jinými instrukcemi, které je možné nahradit sledem jednodušších instrukcí. Mezi tyto instrukce patří u Codix RISC i dělení. To, že se emulační funkce výrazně projevily v profilu celé aplikace, neznamená nutně, že všechny výpočty v této aplikaci jsou *floating point*, ale je z toho patrné, že emulace těchto operací je poměrně náročná. Zbavením se těchto emulovaných výpočtů by došlo k výraznému zrychlení běhu programu.

Převod výpočtů z *floating* na *fixed point* je běžná praxe. Tento problém je možné řešit změnou algoritmu ve zdrojovém kódu tak, aby bylo dosaženo co nejpřesnějšího výsledku, ale s použitím pouze *fixed point* čísel. Ovšem tento přístup je velmi podobný tomu co dělá překladač při emulaci těchto operací. Dojde prakticky k nahrazení operace, která je v případě přítomnosti *FPU* reprezentována jednou instrukcí za sled *fixed point* instrukcí. Právě proto je následně většina času výpočtu aplikace strávena těmito operacemi. Tento přístup tedy ke zrychlení aplikace nevede. Pouze zachová přibližně stejnou přesnost výsledku bez potřeby *FPU* na použitém procesoru.

Existuje však ještě další, principiálně jednodušší řešení: všechna čísla, která jsou definována jako *floating point* jednoduše předefinovat na *fixed point*. Tímto i operace prováděné nad těmito čísly přestanou být operacemi s pohyblivou desetinnou čárkou. Tento zásah samozřejmě způsobí razantní ztrátu přesnosti výpočtu. Proto je otázkou zdali je možné si toto dovolit. U matematických nebo fyzikálních výpočtů, kde je přesnost velmi důležitá, by tento přístup nebyl možný. Ovšem u grafických výpočtů, zvláště takových kde je výsledek hodnocen lidským okem a celková přesnost je limitována dalšími faktory, jako je rozlišovací schopnost obrazovky, nemusí být ztráta přesnosti kritická.

Dobrou ukázkou algoritmů, kde ztráta přesnosti neznamená velký problém, jsou transformační algoritmy z OpenCV (viz kapitola 2.4). Do této kategorie spadá první ukázková aplikace pro rozpoznání hran. Je pravděpodobné, že se výsledný obraz po přepsání algo-

ritmu do *fixed point* aritmetiky liší od výsledného obrazu v případě původní *floating point* verze. Ovšem tyto změny nejsou okem rozpoznatelné, jak je možné porovnat na obrázcích 4.1. Pro to, aby případný pozorovatel mohl určit, že výsledek není naprosto správný, by potřeboval výstup původního algoritmu pro porovnání a tato situace pravděpodobně nenastane. Chyby, které vznikly vlivem nepřesných výpočtů se projeví nejčastěji jako změna odstínu barvy v některých místech obrazu, což není nápadné. Proto je použití těchto úprav pro tuto kategorii algoritmů s využitím pro pouhou prezentaci bezproblémové.

(a) Floating point verze

(b) Fixed point verze

Obrázek 4.1: Porovnání původního výsledku algoritmu s verzí bez floating point operací

Problém však nastane u klasifikačních algoritmů, kam spadá druhá ukázková aplikace pro rozpoznání obličejů. Při klasifikaci dochází ke komplikovanému výpočtu, jehož výsledek je následně porovnán s určitou prahovou hodnotou, která stanovuje mez mezi situací, kdy hledaný objekt na vstupním obrázku je a kdy není. Pokud chyby ve výpočtu, způsobené ztrátou přesnosti po odstranění *floating point* aritmetiky, dosáhnou takové úrovně, že se výsledná hodnota octne na opačné straně prahu než hodnota vypočtená původní verzí algoritmu, znamená to, že dojde k chybné klasifikaci. Tento problém se vyskytl i u této ukázkové aplikace, kde chybná klasifikace způsobí nerozpoznání obličeje tam, kde je, případně označení obličeje v místě vstupního obrazu, kde žádný obličej není.

4.3 Implementace sady SSE2

Tato kapitola se zabývá rozšířením instrukční sady procesoru Codix RISC o instrukce ze sady SSE2 a navzuje na kapitolu 3.3. Úseky kódu v OpenCV, které jsou optimalizovány pomocí SSE2 instrukcí jsou ohraničeny direktivami preprocesoru. Na základě hodnoty příslušného makra je poté optimalizovaný kód preprocesorem zachován nebo odstraněn před překladem. O tom jakou hodnotu potřebné makro nabude, je rozhodnuto při překladu knihovny, kdy probíhá kontrola těchto vlastností na cílové platformě. Codix RISC sadou instrukcí SSE2 nedisponuje a proto je při překladu pro tento procesor makro nastaveno tak, aby byl optimalizovaný kód odstraněn. Ukázka kódu B.1 obsahuje takto optimalizovaný algoritmus OpenCV pro lepší pochopení. Optimalizovaný kód nemá podobu *inline assembleru*, ale jedná se o volání *intrinsics* funkcí. Pokud bych tedy makro definoval sám s hodnotou zachovávající optimalizace nebo odstranil zmíněnou ochranu kódu, nepodařilo by se přeložit program. Překladač by jednoduše nebyl schopen najít implementaci použitých intrinsics funkcí.

Ve chvíli, kdy jsem přeložil některou z testovacích aplikací bez jakýchkoliv úprav, byla přeložena bez použití SSE2 optimalizací. Takto bylo možné ověřit, že je aplikace na platformě s Codix RISC spustitelná, ovšem její provádění zabralo příliš mnoho času. Mít tuto neoptimalizovanou implementaci ovšem bylo důležité, jelikož její výsledky jsem použil pro porovnání s následně vytvořenými optimalizovanými verzemi.

Před přidáním potřebných rozšíření do instrukční sady procesoru Codix RISC jsem vytvořil knihovnu obsahující implementaci těchto funkcí v jazyce C++. Jednotlivé funkce jsem implementoval na základě popisu jejich chování, které poskytuje Intel (ukázka takového popisu je kód 2.2). Nyní jsem mohl povolit překlad optimalizovaného kódu, jelikož má nyní překladač k dispozici implementované *intrinsics* funkce. Ve skutečnosti nebyly využity žádné SSE2 instrukce, protože volání *intrinsics* funkcí vedlo do mnou vytvořené knihovny, která tyto instrukce pouze emuluje stejně jako se to v případě procesoru Codix RISC dělá například s instrukcemi pro operace s plovoucí desetinnou čárkou (touto problematikou se zabývá kapitola 4.2). Tato knihovna tak může zastávat podobnou roli jako *compiler RT* knihovna[2] a to umožňovat spouštění programů, které ve svém zdrojovém kódu obsahují volání SSE2 *intrinsics* funkcí bez toho aby daná architektura obsahovala příslušné instrukce. Knihovna však v tuto chvíli obsahuje pouze podmnožinu SSE2 sady (instrukce, které jsou využity ve vybraných aplikacích OpenCV) a pro obecné využití by bylo potřeba doimplementovat emulace zbývajících instrukcí. Po ověření, že program stále produkuje stejný výstup, jako při použití bez implementované knihovny jsem přistoupil k začlenění samotných instrukcí do instrukční sady Codix RISC. Popis chování instrukce je velmi podobný jazyku C a tudíž jsem mohl vycházet již z implementace použité pro emulaci těchto instrukcí (pro názornost viz kód B.2).

Samotnou emulační knihovnu jsem využil i v tomto kroku. Implementace *intrinsics* funkcí jsem podmínil makrem preprocesoru tak, aby bylo možné se rozhodnout mezi již vytvořenou implementací v jazyce C++ a nyní umožněnou implementací pomocí nových instrukcí v instrukční sadě Codix RISC (viz ukázka kódu B.3).

Takto není nutné přepisovat kód samotné knihovny OpenCV z důvodu optimalizace pomocí instrukčních rozšíření. V případě potřeby další *intrinsics* funkce při optimalizaci jiného algoritmu stačí příslušnou funkci definovat uvnitř emulační knihovny a v jejím těle následně použít potřebné vektorové instrukce. Podobným způsobem je možné přesunout celou aplikaci na jinou platformu, která disponuje jinou instrukční sadou nebo stejnými instrukcemi s jiným jménem nebo syntaxí assembleru. Všechny potřebné změny je nutné udělat pouze v emulační knihovně, kód OpenCV zůstává nezměněn.

4.4 Vlastní SIMD instrukce

I po implementaci instrukcí ze sady SSE2 a tedy optimalizování knihovny OpenCV bylo vykonávání ukázkového programu příliš náročné, aby jej procesor Codix RISC vykonával v reálném čase. To v praxi znamená, že dosažená snímkovací frekvence byla příliš nízká a pozorovateli by se tedy výstup zpracovávaného videa nejevil jako plynulý. Za příčinu této situace je možné považovat nízkou taktovací frekvenci procesoru, která činí 50 MHz. Procesor s dostatečně vysokou taktovací frekvencí by obraz v reálném čase zpracovat dokázal. Ovšem změnit taktovací frekvenci procesoru Codix RISC jsem nemohl. Musel jsem tedy najít takovou optimalizační metodu, která urychlí výpočet ukázkové aplikace více než zavedení SSE2 instrukcí. Ačkoliv řešení nespočívá v implementaci již používané sady vektorových

[2] *Run time* knihovna překladače

instrukcí, rozšíření instrukční sady se pořád jevilo jako vhodný způsob.

Je možné implementovat instrukce, které nahradí různé úseky kódu. Smysl to má však jen tehdy, pokud se jedná o kód, který je vykonáván často, a tudíž velmi zatěžuje procesor. K tomu abych odhalil taková místa, jsem použil znovu profilaci. Takto jsem tedy nalezl nejnáročnější funkce. Ovšem nahradit celou funkci jednou instrukcí není možné. Konkrétní úsek funkce, jehož výpočet zabírá nejvíce času, jsem nalezl pomocí umístění speciální instrukce pravidelně do kódu zkoumané funkce. Tato instrukce nedělá nic jiného, než tisk počtu uběhlých procesorových cyklů od spuštění aplikace. Díky množství cyklů, které uplynuly mezi jednotlivými výpisy, je možné odhalit nejnáročnější úseky jednotlivých funkcí. Je to možné učinit i odhadem, jelikož se nejčastěji jedná o cykly, ale tento způsob je prokazatelnější. Právě cykly jsou programové konstrukce, které je možné optimalizovat vektorovými instrukcemi. Následuje přibližný postup této činnosti:

1. *Rozbalení smyček*[3] - Velmi často zpracovává každá iterace cyklu jedno číslo, které může být na příklad prvkem pole. Jako příklad uvedu 32bitové číslo a v případě Codix RISC jsem vytvářel 128bitové vektorové instrukce. V této situaci je nutné smyčku přepsat tak, aby v každé iteraci zpracovávala hned čtyři prvky a pro další iteraci se o čtyři prvky zpracování posunulo. Už tato změna může přinést zrychlení běhu aplikace. Odstraní se totiž přebytečná logika skoku, která se provádí mezi každými dvěma iteracemi, protože proběhne oproti původní implementaci jen čtvrtina iterací. Proto je tato technika sama o sobě optimalizační technikou. Ovšem v tomto případě pouze umožňuje použití efektivnější techniky - zavedení vektorových instrukcí.

2. *Rozšíření instrukční sady* - Následně je nutné implementovat instrukci, jejíž chování je ekvivalentní jedné iteraci optimalizované smyčky. Instrukce provede stejnou operaci nad všemi operandy. V případě 32bitového operandu zpracovala instrukce čtyři operandy najednou. Samotné načtení sady operandů z paměti a jejich následné uložení do paměti obstarávají další dvě instrukce, které musí být také přidány do instrukční sady.

3. *Úprava kódu* - Oproti použití SSE2 sady je nutné provézt úpravy přímo ve zdrojovém kódu knihovny OpenCV. Překladač má sice nyní k dispozici instrukce ekvivalentní iteraci smyčky, ale sám je v tomto místě nepoužije, jelikož jsou příliš složité. Proto je nutné tělo smyčky přepsat v *inline assembleru* s použitím vektorových instrukcí.

Následující ukázka kódu 4.5 porovnává částečně rozmotanou smyčku, ale stále implementovanou v jazyce C++ a stejný úsek kódu přepsaný do *inline assembleru* s využitím *SIMD* instrukcí. Vždy je přeložena jen jedna z těchto částí a to je podmíněno makrem preprocesoru *CODASIP_OCVEXT*.

Vložený kód 4.5: Úsek kódu optimalizovaný vlastními SIMD instrukcemi

```
        for( ; i <= width - 8; i += 8 )
        {
#ifndef CODASIP_OCVEXT
            D[i] = S2[i] - S0[i];
            D[i+1] = S2[i+1] - S0[i+1];

            D[i+2] = S2[i+2] - S0[i+2];
```

[3] *Loop unrolling*

```
                D[i+3] = S2[i+3] - S0[i+3];
                D[i+4] = S2[i+4] - S0[i+4];
                D[i+5] = S2[i+5] - S0[i+5];
                D[i+6] = S2[i+6] - S0[i+6];
                D[i+7] = S2[i+7] - S0[i+7];
#else
                asm (
                "load_128 v0, %[psrc0], 0 \n\t"
                "load_128 v1, %[psrc1], 0 \n\t"
                "nop \n\t"
                "nop \n\t"
                "colfilter_asym_lo v3, v0, v1\n\t"
                "load_128 v0, %[psrc0], 16 \n\t"
                "load_128 v1, %[psrc1], 16 \n\t"
                "nop \n\t"
                "nop \n\t"
                "colfilter_asym_hi v3, v0, v1\n\t"
                "store_128 v3, %[pdst], 0 \n\t"
                :
                : [psrc0]"r"(&S0[i]), [psrc1]"r"(&S2[i]),
                                      [pdst]"r"(&D[i])
                : "memory"
                );
#endif
        }
```

V případě, že dojde k dobré identifikaci náročných míst v kódu je možné implementováním instrukcí na míru pro tyto úseky kódu dosáhnout lepších výsledků než v případě použití existujících instrukčních rozšíření. Tak tomu je i v případě použití pro OpenCV aplikace. Ovšem toto řešení trpí ztrátou obecnosti. Procesor takto navržený je jednoúčelový. V případě, že hlavním kritériem je efektivita, je toto řešení přijatelné, stejně tak tomu je v případě této práce. V ostatních případech je rozumnější rozšířit instrukční sadu o standardní instrukce, které jsou využívány v širokém spektru již napsaných zdrojových kódů.

4.5 Výsledky

4.5.1 Rozpoznání hran

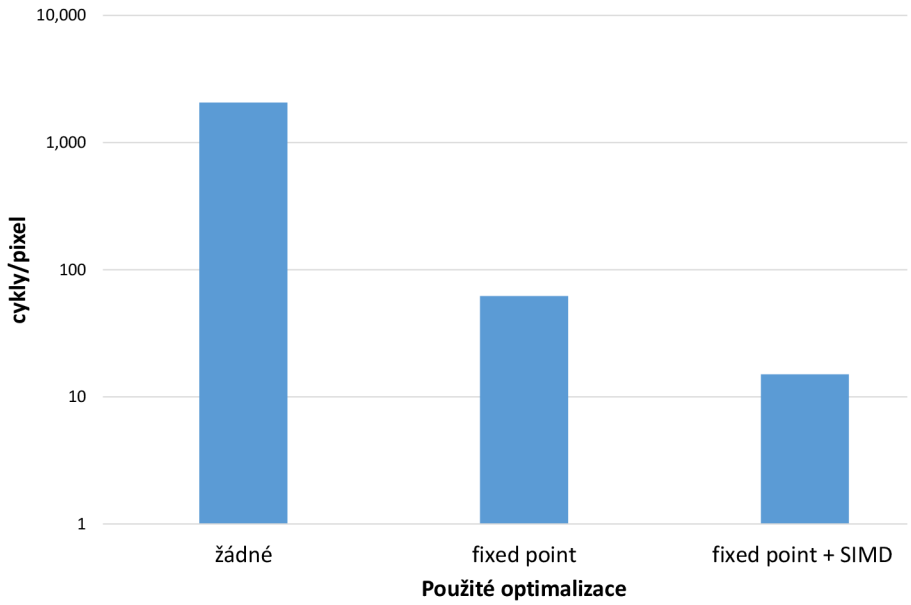
Tato ukázková aplikace se nakonec projevila jako velmi vhodná z pohledu optimalizace. Byly na ní postupně vyzkoušeny všechny zmíněné metody, výsledná verze je však optimalizovaná jen pomocí převedení *floating point* aritmetiky na *fixed point* a zavedením *SIMD* instrukcí.

Graf na obrázku 4.2 znázorňuje výsledky po postupném aplikování zmíněných optimalizací. Výsledky jsou zaznačeny v logaritmickém měřítku. Tyto stejné výsledky jsou pro přehlednost přepsány do tabulky 4.1. Výsledky jsou uvedeny v počtu procesorových cyklů nutných pro zpracování jednoho pixelu vstupního obrazu (c/p). Jako testovací vstup byl použit obrázek o rozlišení $512x512$ pixelů. Všechny výsledky byly naměřeny na procesoru Codix RISC.

Použité optimalizace	cykly/pixel
žádné	2063,504
fixed point	62,532
fixed point + SIMD	15,358

Tabulka 4.1: Výsledky optimalizace algoritmu pro rozpoznání hran

První hodnota se vztahuje k neupravené podobě algoritmu, tak jak je implementován v knihovně OpenCV. Druhá hodnota odpovídá verzi algoritmu, kde byly odstraněny *floating point* operace a nahrazeny za *fixed point*. Poslední výsledek jsem naměřil u aplikace s použitými, na míru pro tuto aplikaci vytvořenými, *SIMD* instrukcemi, které nahrazují náročné úseky kódu.



Obrázek 4.2: Porovnání rychlosti algoritmu s různými optimalizacemi (výsledky zaznačeny v logaritmickém měřítku)

Cílem bylo optimalizovat aplikaci tak, aby byla detekce hran prováděna s co nejvyšší snímkovací frekvencí. Limitem je samozřejmě frekvence snímání obrazu kamerou. Procesor Codix RISC je taktován na frekvenci 50 MHz, tedy vykoná padesát milionů cyklů za jednu sekundu. V případě zachování rozlišení vstupního obrazu na hodnotě $512x512$ pixelů a použití nejlépe optimalizované verze algoritmu získáme tuto hodnotu pro počet cyklů nutných pro zpracování jednoho vstupního obrazu:

$$15,358 * 512 * 512 = 4026007.552 \text{ cyklů} \tag{4.1}$$

Následným vydělením frekvence procesoru počtem cyklů nutným pro zpracování jednoho vstupu získáme snímkovací frekvenci:

$$50000000/4026007.552 = 12,419 fps \tag{4.2}$$

Pro dosáhnutí vyšší snímkovací frekvence je možné snížit rozlišení vstupního obrazu. Ten tak bude obsahovat menší počet pixelů a tudíž bude jeho zpracování trvat kratší čas a za jednu sekundu se tak stihne zpracovat větší množství vstupů.

4.5.2 Rozpoznání obličejů

Tato aplikace se ukázala jako problematická. Algoritmus využívá velké množství operací s plovoucí desetinnou čárkou (jak je patrné z ukázky profilu 4.6). Ty jsem se pokusil odstranit stejným způsobem jako u algoritmu pro rozpoznání hran. To však v tomto případě vedlo k nepřijatelným výsledkům se špatně rozpoznanými obličeji.

Vložený kód 4.6: Část profilu aplikace pro rozpoznání obličejů

```
<FUNCTION>
        <ID>__muldi3</ID>
        <PERCENT>19.6</PERCENT>
        <COUNT>461029</COUNT>
        <ADDITION>461029</ADDITION>
</FUNCTION>
<FUNCTION>
        <ID>__lshrdi3</ID>
        <PERCENT>16</PERCENT>
        <COUNT>375722</COUNT>
        <ADDITION>836751</ADDITION>
</FUNCTION>
<FUNCTION>
        <ID>__ashldi3</ID>
        <PERCENT>14.5</PERCENT>
        <COUNT>339707</COUNT>
        <ADDITION>1176458</ADDITION>
</FUNCTION>
```

Implementace tohoto algoritmu v OpenCV navíc nedisponuje verzí optimalizovanou pomocí sady instrukcí SSE2. Bohužel však nebylo ani možné najít vhodné místo pro zavedení vlastních vektorových instrukcí. To se stalo z toho důvodu, že algoritmus neobsahuje mnoho míst, jako jsou smyčky pracující nad poli hodnot. Práce s poli, která v algoritmu použita je, často přistupuje k prvkům pole nesekvenčně a tento přístup není možné implementovat pomocí *SIMD* instrukcí, jelikož ty potřebují načíst množinu operandů souvisle uloženou v paměti. Snaha o použití vlastních *SIMD* instrukcí také nevedla k viditelnému zrychlení aplikace.

4.6 Rozšíření Codix RISC

V této práci jsem se zabýval optimalizací vybraných aplikacích založených na knihovně OpenCV. Má však smysl uvažovat jakým způsobem je možné rozšířit procesor Codix RISC tak, aby na něm bylo možné spustit větší množství optimalizovaných aplikací z této knihovny. Níže uvádím dva různé přístupy k tomuto problému.

- Méně invazivním a jednodušším řešením je použití existujícího rozšíření instrukční sady. Řada algoritmů v OpenCV je již optimalizována instrukcemi ze sad SSE a tudíž

dává smysl přidat do instrukční sady procesoru právě instrukce z těchto sad. Obecnost tohoto řešení navíc přesahuje použití v OpenCV, protože se jedná o standardní instrukční rozšíření použité i v jiných projektech. Další výhodou je, že není nutné upravovat kód samotné knihovny (tento princip je vysvětlen v kapitole 4.3).

V případě optimalizace aplikace pro rozpoznání hran je potřeba pouze *14 instrukcí* ze sady SSE2. Samotná sada SSE2 však obsahuje *144 instrukcí*, kterými rozšiřuje starší instrukční sadu SSE, která obsahuje *instrukcí 70* [17]. Aby tedy optimalizace pomocí těchto sad mohly být opravdu obecně použitelné, je nutné přidat instrukce všechny, což by markantně rozšířilo instrukční sadu tohoto procesoru. Reálně je možné přidat do instrukční sady jen několik potřebných instrukcí, ovšem nebude poté možné počítat s optimalizacemi v celém OpenCV.

- Při finální optimalizaci aplikace pro rozpoznání hran jsem do instrukční sady procesoru Codix RISC přidal osm instrukcí. Šest z nich jsou instrukce specificky vytvořené pro optimalizaci konkrétních částí zdrojových kódů OpenCV, které implementují výpočet absolutní hodnoty, průměru hodnot a filtrů nutných při detekci hran. Tyto instrukce jsou oproti standardním instrukčním rozšířením použitelné pouze pro tento konkrétní úsek kódu. Zbývající dvě instrukce realizují 128bitový *load* (načtení hodnoty z paměti do registru) a *store* (uložení hodnoty z registru do paměti). Tyto dvě instrukce jsou nutné pro samotné použití dalších vektorových instrukcí, jelikož načtení hodnoty z paměti a po provedení operace její opětovné uložení do paměti probíhá vždy.

  Takto je možné dosáhnout efektivnějšího kódu, jelikož je celý optimalizovaný kód reprezentován jednou instrukcí (nepočítaje instrukce pro *load* a *store*), což se nestává při optimalizaci standardní sadou instrukcí, kdy je nutné použít sekvenci instrukcí. Je však nepravděpodobné, že instrukce implementující určitý úsek kódu knihovny bude použitelná ještě v jiném místě zdrojového kódu. V případě optimalizace celé knihovny by tedy bylo nutné přidat množství instrukcí velmi pravděpodobně větší než je počet instrukcí v sadách SSE. Navíc je nutné určit náročná místa kódu, která je praktické optimalizovat a následně je přepsat do *inline assembleru*.

Obě tato řešení nabízí výhody a nevýhody. V případě použití sady SSE je nutné pouze implementovat instrukce, jejichž chování je zdokumentováno Intelem a následně povolit použití těchto rozšíření při překladu knihovny. Implementace instrukcí na míru knihovně OpenCV je pracnější proces, ale je možné dosáhnout lepších výsledků.

Kromě rozšíření procesoru o vektorové instrukce je možné jej také rozšířit o *FPU* jednotku. Tato jednotka by umožnila výrazně rychlejší vykonávání aplikací, které jsou náchylné na přesnost výpočtu, jako je tomu v případě aplikace na rozpoznání obličeje.

Kapitola 5

Závěr

Primárním cílem této práce bylo zprovoznění aplikace implementované nad OpenCV pro prezentaci této knihovny na platformě založené na procesoru Codix RISC. Tento cíl se mi podařilo splnit. Vybral jsem dvě aplikace, tak abych zvýšil pravděpodobnost, že se alespoň jedna z nich projeví jako vhodná pro tento účel. Ve výsledku každá z těchto aplikací posloužila jinému účelu. Program pro detekci hran slouží právě jako ukázka toho, jak je možné pomocí nástrojů Codasip Frameworku specializovat procesorové jádro pro použití s konkrétní aplikací. Proces optimalizace aplikace pro rozpoznání hran naopak poukazuje na omezené schopnosti procesorového jádra Codix RISC a naznačuje, že pro aplikace podobné této by mohlo být nutné vytvořit jádro nativně podporující operace s pohyblivou desetinnou čárkou.

V případě optimalizace aplikace pro detekci hran se může jevit jako zásadní zrychlení způsobené odstraněním operací s plovoucí desetinnou čárkou. Tuto optimalizaci bylo nutné provést pro dosažení dobrých hodnot snímkovací frekvence při běhu algoritmu na Codix RISC. Neprezentuje však vlastnosti Codasip Frameworku. Ty jsou velmi dobře prezentovány zrychlením dosaženým po rozšíření instrukční sady o vektorové instrukce. Tato úprava totiž zrychlila algoritmus více, než čtyřikrát což je velmi dobrý výsledek. Implementace těchto rozšíření pro ukázkovou aplikaci slouží jako důkaz, že má smysl podobně optimalizovat i další algoritmy s cílem dosáhnout lepších výsledků než jakých dosahují konvenční optimalizační metody.

Mezi výstupy této práce, které nemusí být na první pohled patrné, patří obecně použitelná standardní knihovna C++, která je nyní přeložitelná pro procesor Codix RISC a umožňuje tak vytváření komplexnějších aplikací implementovaných v jazyce C++ pro tento procesor. Ačkoliv byla v rámci této práce přeložena z důvodu následného překladu knihovny OpenCV, je možné ji použít pro implementaci jakýchkoliv dalších aplikací. Stejným způsobem mohou další vývojáři aplikací pro platformu Codix RISC využít knihovnu OpenCV.

Samozřejmě je však na základě výsledků možné uvažovat i o dalším rozvoji této práce. Především je to optimalizace, která postihne co nejširší použití knihovny OpenCV. Optimalizace ukázkových aplikací měla posloužit pouze jako zkouška, zdali je možné dosáhnout efektivního zpracování OpenCV algoritmů na aplikačně specifických procesorech vytvořených pomocí Codasip Frameworku. Techniky vedoucí k optimalizaci celé knihovny jsou v této práci diskutovány.

Literatura

[1] MUJTABA, Hassan. Intel Xeon E5-2600 V3 "Haswell-EP"Workstation and Server Processors Unleashed For High-Performance Computing. [online]. 2014 [cit. 2015-07-29]. Dostupné z: http://wccftech.com/intel-xeon-e52600-v3-haswellep-workstation-server-processors-unleashed-highperformance-computing/

[2] PAOLO IENNE, Rainer Leupers. *Customizable embedded processors design technologies and applications*. San Francisco, Calif: Elsevier/Morgan Kaufmann, 2006. ISBN 0080490980.

[3] DANDAMUDI, Sivarama P. *Guide to RISC processors: for programmers and engineers*. New York: Springer, c2005, xv, 387 p. ISBN 03-872-1017-2.

[4] LATTNER, Chris. The Architecture of Open Source Applications: LLVM. [online]. [cit. 2015-07-25]. Dostupné z: http://www.aosabook.org/en/llvm.html

[5] CODASIP. *Codasip Framework Manual*. 6.5. Brno, 2014.

[6] Codasip: Products. [online]. [cit. 2015-07-25]. Dostupné z: https://www.codasip.com/products/

[7] SCHLIEBUSCH, Oliver, Heinrich MEYR a Rainer LEUPERS. *Optimized ASIP synthesis from architecture description language models*. Dordrecht: Springer, c2007, xiv, 193 p. ISBN 14-020-5686-9.

[8] CODASIP. *CodAL Manual: reference guide*. 5.0. Brno, 2014.

[9] OpenCV: About. [online]. [cit. 2015-07-25]. Dostupné z: http://opencv.org/about.html

[10] KAEHLER, By Gary Bradski and Adrian. *Learning OpenCV Computer Vision with the OpenCV Library*. Sebastopol: O'Reilly Media, Inc, 2008. ISBN 05-965-5404-4.

[11] JOSSUTIS, Nicolai M. *C standard library: a tutorial and reference*. Boston: Addison-Wesley, c1999, xx, 799 s. ISBN 02-013-7926-0.

[12] LLVM: libc++. [online]. [cit. 2015-07-25]. Dostupné z: http://libcxx.llvm.org

[13] GCC: libstdc++ FAQ. [online]. [cit. 2015-07-25]. Dostupné z: https://gcc.gnu.org/onlinedocs/libstdc++/faq.html

[14] Apache: stdcxx. [online]. [cit. 2015-07-25]. Dostupné z: http://stdcxx.apache.org

[15] XAVIER, C a S IYENGAR. *Introduction to parallel algorithms*. New York: Wiley, c1998, xvi, 365 p. ISBN 04-712-5182-8.

[16] BY PAUL COCKSHOTT, Kenneth Renfrew. *SIMD Programming Manual for Linux and Windows*. London: Springer London, 2004. ISBN 14-471-3862-7.

[17] Intel: Processors. [online]. [cit. 2015-07-25]. Dostupné z: http://www.intel.com/support/processors/sb/CS-030123.htm

[18] MSDN: Compiler Intrinsics. [online]. [cit. 2015-07-25]. Dostupné z: https://msdn.microsoft.com/en-us/library/26td21ds.aspx

[19] LIU, Dake. *Embedded Dsp Processor Design Application Specific Instruction Set Processors*. Burlington: Elsevier, 2008. ISBN 00-805-6987-0.

[20] MSDN: Profiling Overview. [online]. [cit. 2015-07-25]. Dostupné z: https://msdn.microsoft.com/cs-cz/library/bb384493(v=vs.110).aspx

[21] Cadence: Image/Video Processing. [online]. [cit. 2015-07-25]. Dostupné z: http://ip.cadence.com/ipportfolio/tensilica-ip/image-video-processing

[22] PR Newswire: Itseez OpenCV Library Now Available on Cadence Tensilica Image/Vision DSPs. [online]. [cit. 2015-07-25]. Dostupné z: http://www.prnewswire.com/news-releases/itseez-opencv-library-now-available-on-cadence-tensilica-imagevision-dsps-300111771.html

[23] CEVA: CEVA-MM3101. [online]. [cit. 2015-07-25]. Dostupné z: http://www.ceva-dsp.com/CEVA-MM3101

[24] Xilinx: Company Overview. [online]. [cit. 2015-07-25]. Dostupné z: http://www.xilinx.com/about/company-overview.html

[25] NEUENDORFFER, Stephen, Thomas LI a Devin WANG. XILINX. *Accelerating OpenCV Applications with Zynq-7000 All Programmable SoC using Vivado HLS Video Libraries*. 2015. Dostupné z: http://www.xilinx.com/support/documentation/application_notes/xapp1167.pdf

[26] The Apache Software Foundation: Apache Licence. [online]. [cit. 2015-07-29]. Dostupné z: http://www.apache.org/licenses/LICENSE-2.0

[27] MAINI, Raman a Dr. Himanshu AGGARWAL. Study and Comparison of Various Image Edge Detection Techniques. 2009. Dostupné z: http://people.ucalgary.ca/~dasaid/CPSC535/lab10/IJIP-15.pdf

[28] OpenCV Documentation: Face Detection using Haar Cascades. [online]. [cit. 2015-07-26]. Dostupné z: http://docs.opencv.org/master/d7/d8b/tutorial_py_face_detection.html

Příloha A

Obsah CD

Přiložené CD obsahuje následující adresáře:

- */doc* - Obsahuje technickou zprávu ve formátu pdf.
- */tex* - Obsahuje zdrojové kódy technické zprávy a obrázky v ní použité.
- */src* - Obsahuje zdrojové kódy vztahující se k optimalizaci knihovny OpenCV pomocí rozšíření instrukční sady. Struktura adresáře je detailněji popsána v souboru README obsaženém na CD.

Příloha B

Příklad implementace SSE2 instrukce

Vložený kód B.1: Část algoritmu implementovaná jak s použitím SSE2, tak bez této optimalizace

```
//část implementovaná pomocí SSE2 intrinsics funkcí je chráněna
        direktivou #if
#if CV_SSE2
        if( USE_SSE2 )
        {
            __m128 a4 = _mm_set1_ps(alpha),
                b4 = _mm_set1_ps(beta),
                g4 = _mm_set1_ps(gamma);
            __m128i z = _mm_setzero_si128();

            for( ; x <= size.width - 8; x += 8 )
            {
                __m128i u = _mm_unpacklo_epi8(
                        _mm_loadl_epi64(
                                (const __m128i*)(src1 + x)), z);
                __m128i v = _mm_unpacklo_epi8(
                        _mm_loadl_epi64(
                                (const __m128i*)(src2 + x)), z);

                __m128 u0 = _mm_cvtepi32_ps(
                        _mm_unpacklo_epi16(u, z));
                __m128 u1 = _mm_cvtepi32_ps(
                        _mm_unpackhi_epi16(u, z));
                __m128 v0 = _mm_cvtepi32_ps(
                        _mm_unpacklo_epi16(v, z));
                __m128 v1 = _mm_cvtepi32_ps(
                        _mm_unpackhi_epi16(v, z));

                u0 = _mm_add_ps(_mm_mul_ps(u0, a4),
                                _mm_mul_ps(v0, b4));
                u1 = _mm_add_ps(_mm_mul_ps(u1, a4),
                                _mm_mul_ps(v1, b4));
                u0 = _mm_add_ps(u0, g4); u1 = _mm_add_ps(u1, g4);
```

```
                u = _mm_packs_epi32(_mm_cvtps_epi32(u0),
                                    _mm_cvtps_epi32(u1));
                u = _mm_packus_epi16(u, u);

                _mm_storel_epi64((__m128i*)(dst + x), u);
            }
        }

#endif

//ekvivalentní algoritmus bez použití SSE2 rozšíření
        for( ; x < size.width; x++ )
        {
            float t0 = CV_8TO32F(src1[x])*alpha
                + CV_8TO32F(src2[x])*beta
                + gamma;
            dst[x] = saturate_cast<uchar>(t0);
        }
```

Vložený kód B.2: Implementace instrukce unpackhi_epi16 v jazyce CodAL

```
element instr_codasip_unpackhi_epi16
{
        use vreg as src1, src2, dst;
        use opc_codasip_unpackhi_epi16;

        assembler { opc_codasip_unpackhi_epi16 dst ","
                src1 "," src2 };
        binary { OPC6_VECTOR3:6 dst src1 src2 0:8
                opc_codasip_unpackhi_epi16 };

        semantics
        {
                v8ui16 s1, s2;
                v8ui16 res;

                s1 = (v8ui16)vregs[src1];
                s2 = (v8ui16)vregs[src2];

                res[0] = s1[4];
                res[1] = s2[4];
                res[2] = s1[5];
                res[3] = s2[5];
                res[4] = s1[6];
                res[5] = s2[6];
                res[6] = s1[7];
                res[7] = s2[7];

                vregs[dst] = (uint128)res;
        };
};
```

Vložený kód B.3: Intrinsics funkce pro instrukci unpackhi_epi16 v emulační knihovně

```
__m128i _mm_unpackhi_epi16 (__m128i a, __m128i b)
{
#ifndef CODIX_SSE
        v8ui16 dst, src1, src2;
        src1 = (v8ui16)a;
        src2 = (v8ui16)b;

        dst[0] = src1[4];
        dst[1] = src2[4];
        dst[2] = src1[5];
        dst[3] = src2[5];
        dst[4] = src1[6];
        dst[5] = src2[6];
        dst[6] = src1[7];
        dst[7] = src2[7];

        return (__m128i)dst;
#else
        __m128i res;
        asm (
                "load_128 v0, %[psrc1], 0 \n\t"
                "load_128 v1, %[psrc2], 0 \n\t"
                "nop \n\t"
                "nop \n\t"
                "codasip_unpackhi_epi16 v2, v0, v1\n\t"
                "store_128 v2, %[pdst], 0 \n\t"
                :
                : [psrc1]"r"(&a), [psrc2]"r"(&b), [pdst]"r"(&res)
                : "memory"
        );
        return res;
#endif
}
```