VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÝCH SYSTÉMŮ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER SYSTEMS

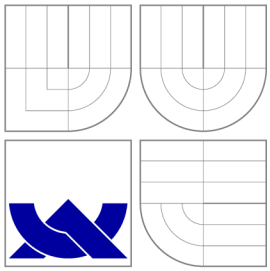
AKCELERACE MULTIMEDIÁLNÍCH APLIKACÍ POMOCÍ KOPROCESORU NEON

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

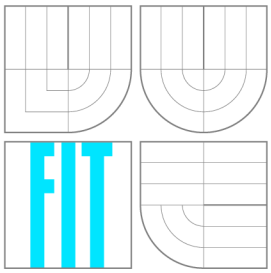
AUTOR PRÁCE
AUTHOR

RADIM KRATOCHVÍL

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

AKCELERACE MULTIMEDIÁLNÍCH APLIKACÍ POMOCÍ KOPROCESORU NEON

ACCELERATION OF GRAPHICS ALGORITHMS BY NEON COPROCESSOR

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

RADIM KRATOCHVÍL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. JAN VIKTORIN

BRNO 2015

Abstrakt

Cílem této práce je prozkoumat možnosti koprocesoru NEON. Porovnávají se grafické algoritmy napsané v jazyce C, jazyce symbolických adres, jazyce C s využitím intrinsických funkcí a automaticky vektorizovaný kód. Hlavním zjištěním je, že jde zkrátit délku výpočtu až 60 krát a díky tomu by bylo možné tyto algoritmy provádět v reálném čase na HD videu.

Abstract

The aim of this work is to examine capabilities of NEON coprocessor. Various implementations of the same algorithm are compared: language C, assembly language, language C with intinsic functions and automatically vectorized code. Main conslusion is, that computation time can be reduced up to 60 times, allowing real-time HD video processing.

Klíčová slova

ARM, NEON, GCC, Automatická vektorizace.

Keywords

ARM, NEON, GCC, Automatic vectorization.

Citace

Radim Kratochvíl: Akcelerace multimediálních aplikací pomocí koprocesoru NEON, bakalářská práce, Brno, FIT VUT v Brně, 2015

Akcelerace multimediálních aplikací pomocí koprocesoru NEON

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Viktorina

.....
Radim Kratochvíl
20. května 2015

Poděkování

Rád bych poděkoval mému vedoucímu práce Ing. Janu Viktorinovi, za ochotnou pomoc, cenné rady a věcné připomínky.

Obsah

Kapitola 1

Úvod

S rozmachem smartphonů a dalších chytrých zařízení došlo k poměrně velkému rozšíření procesorů od firmy ARM. Jak se tyto přístroje postupně vyvíjely a vylepšovaly, začaly mít vyšší požadavky také na procesor. Většinou tato zařízení slouží ke konzumaci obsahu, a to znamená, že jsou zde kladeny velké nároky na zobrazování. Práce s obrazem se používá i v embedded aplikacích, například pro detekci obrazu z průmyslových kamer. Proto, aby ARM dokázal lépe pracovat s obrazem, stal se součástí koprocesor NEON.

Tento koprocesor obsahuje FPU jednotku a navíc přidává SIMD (Single Instruction, Multiple Data) instrukce. Je to obdoba MMX (a jeho následovníků) na platformě x86. Koprocesor NEON dokáže provést jednoduchou operaci na více datech zároveň (čím je menší datová šířka zpracovávaného slova, tím více slov lze zpracovat najednou). Toto je ideální pro práci s obrazem, jelikož obsahuje mnoho pixelů a každý z nich má většinou (v dnešní době) jen 24-bitů. Toto platí v případě barevného obrazu, u černobílého je to jen 8-bitů.

Cílem mé bakalářské práce je ověření možností tohoto koprocesoru. V rámci této práce bude porovnáván program napsaný v jazyce C, jazyku symbolických adres a jazyce C s využitím intrinsických funkcí. Dále byla testována schopnost kompilátoru automatický vektorizovat kód. Jako operační systém bude použita Linuxová distribuce Buildroot. Testování bude provedeno na vybraných grafických algoritmech viz kapitola Algoritmy 3

Kapitola 2

ARM

ARM je referenční architektura mikroprocesoru od firmy ARM Holdings. Zkratka ARM znamená Advanced RISC Machine (nebo také Acorn RISC Machine), z čehož vyplývá že má redukovanou instrukční sadu. Společnost ARM Holdings tyto procesory nevyrábí, jen je licencuje jiných výrobcům. Tyto procesory mají šířku datové sběrnice 32 nebo 64 bitů. Jejich velkou předností je nízká spotřeba energie, a proto se využívají především v mobilních zařízeních a ve vestavěných systémech. [1]

Rozdělují se do tří skupin, a to:

- **Cortex-A** – Aplikační procesory, určené například do chytrých telefonů.
- **Cortex-M** – Mikrokontroléry.
- **Cortex-R** – Mikrokontroléry určené pro práci v reálném čase.

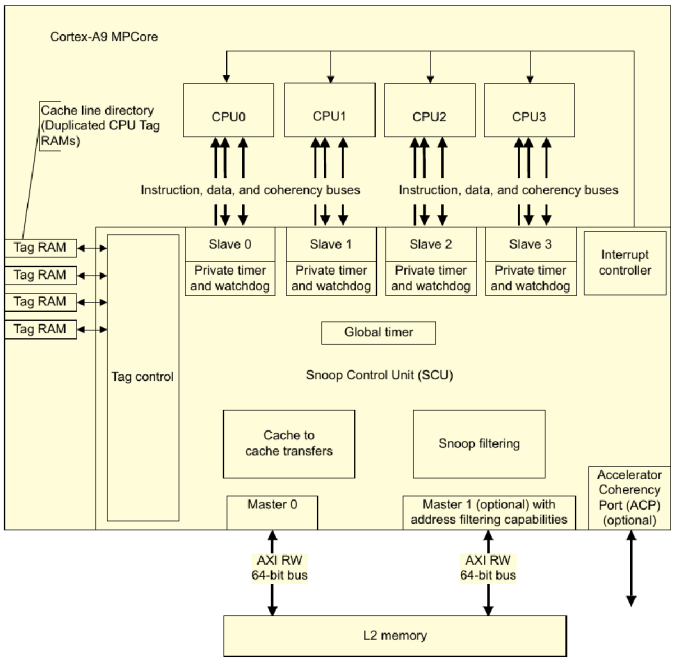
2.1 Cortex-A9

Cortex-A9 je výkonný procesor, který implementuje architekturu ARMv7-A a podporuje 32-bitové instrukce ARM, 16-bitové a 32-bitové Thumb instrukce a 8-bitový Java bytecodes. Jedná se o Harvardskou architekturu, tedy data jsou oddělena od instrukcí programu. Procesor má k dispozici duální 64-bitovou sběrnici AMBA 3 AXI, na které se přenáší data a instrukce nezávisle na sobě po vlastních vodičích. Je obousměrná a podporuje i zařízení s dlouhou dobou odezvy. Využívá super-skalárního zřetězení a je schopen dekódovat až 4 instrukce za takt. Taktovací kmitočet může být až 2000 MHz. Umožňuje implementovat až čtyř-jádrové procesory. Celý systém s více jádry se pak nazývá Cortex A9 MPCore. Jeho blokové schéma je znázorněno na obrázku 2.1. Komunikaci mezi jádry skrze sběrnici AXI řídí blok nazvaný Snoop Control Unit (SCU). SCU řídí práci s vyrovnávací pamětí a paměťové přesuny. Na tomto procesoru může běžet plnohodnotný operační systém a uživatelské aplikace.

Velikost vyrovnávací paměti L1 je 32 kB pro data a 32 kB pro instrukce. Každé jádro má svojí L1 vyrovnávací paměť. Sdílená vyrovnávací paměť L2 má velikost 512 kB.

K procesoru lze připojit jeden z koprocesorů, a to buď jednoduší, který obsahuje pouze FPU jednotku (pro práci s čísly v plovoucí desetinné čárce) a nebo koprocesor NEON (pro práci s multimédii). NEON kromě vektorových operací zvládá i práci s čísly v plovoucí desetinné čárce. Každé jádro má svůj koprocesor. [15]

Tento procesor je určen pro výkonná mobilní zařízení, síťová a automobilová zařízení.



Obrázek 2.1: Blokové schéma Cortex A9 MPCore [15]

2.2 Vývojová deska SoCrates

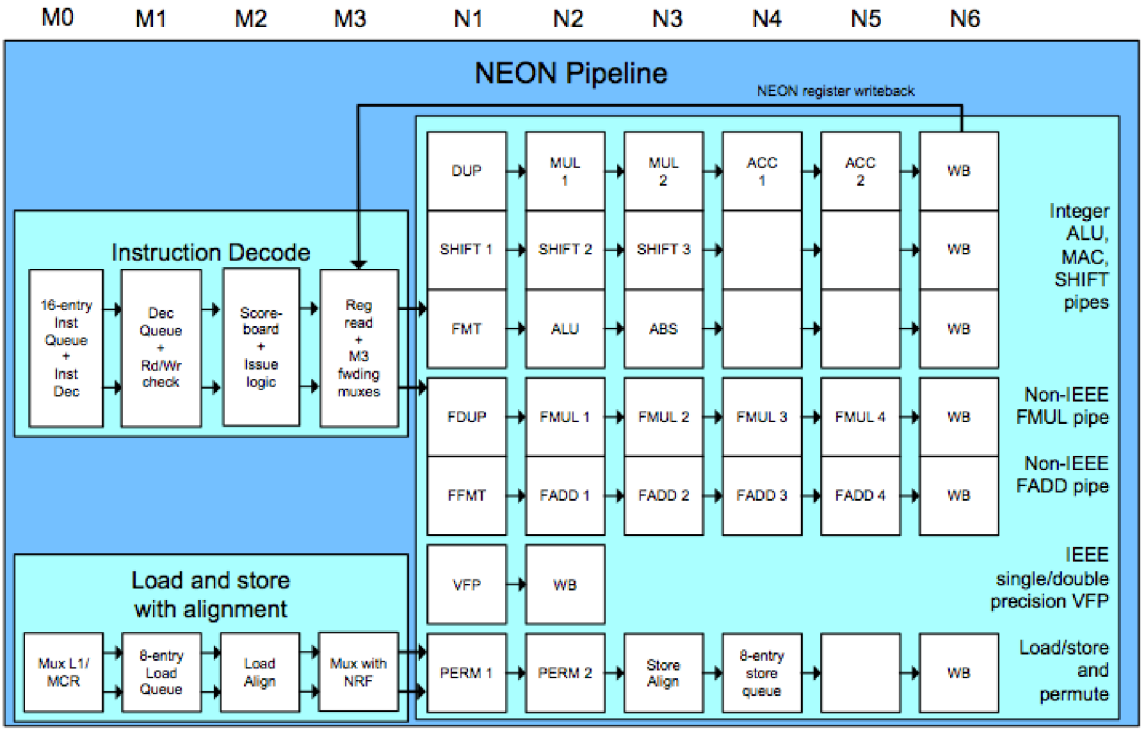
Pro testování byla použita vývojová deska SoCrates, která obsahuje SoC (System-on-Chip) Altera Cyclone V. Tento SoC kombinuje programovatelné hradlové pole FPGA a dvoujádrový ARM Cortex-A9, který běží na frekvenci 800 MHz. Operační paměť má velikost 1 GB a je typu DDR3. Hodinový signál má frekvenci 25 MHz. Deska obsahuje slot na microSD kartu, který se dá využít pro nahrávání operačního systému a ukládání dat. Dále je přítomný konektor pro 1 Gbps Ethernet, USB, I2C, SPI a senzor na měření teploty. Pro komunikaci s počítačem lze využít integrovaný převodník UART-USB. Na obrázku 2.7 je znázorněno celé blokové schéma. [11]

2.3 NEON

Koprocesor NEON rozšiřuje funkcionalitu procesoru Cortex-A9 o podporu instrukcí SIMD a vektorových operací v číslech s plovoucí desetinou čárkou (FP). Smyslem Koprocesoru NEON je urychlení multimediálních výpočtů pomocí vektorového zřetězeného zpracování aritmetických instrukcí. Je oddělen od jádra procesoru a dokáže nezávisle adresovat data v paměti (instrukce LOAD/STORE). Díky tomu je schopný pracovat paralelně s jádrem procesoru. K dalším výhodám tohoto koprocesoru patří podpora nezarovnaného přístupu do paměti a automatizované načítání struktur. Umožňuje načítat data s pevně daným rozestupem (1 až 4 Byty).

Zřetězené zpracování tohoto koprocesoru je zobrazeno na obrázku 2.2. Vlevo je vstupní fronta instrukcí, která je plněna procesorem. Následuje dekódování instrukce v několika krocích. Dekódována instrukce se předá do aritmeticko logické jednotky. Jde vidět, že dekódování je zdvojené a lze tedy teoreticky zpracovávat dvě instrukce najednou. To je závislé na typu instrukce, protože koprocesor NEON má jen jednu jednotku pro celočíselné výpočty

a jednu pro výpočty v plovoucí desetinné čárce. Většinou se tedy zároveň zpracovává jedna instrukce pro výpočet a druhá pro vstupní, nebo výstupní operaci, která je nezávislá na výpočetních jednotkách. Výpočetní jednotky mají šest zřetězených stupňů. [14]



Obrázek 2.2: Vektorové zpracování na jednotce NEON. [8]

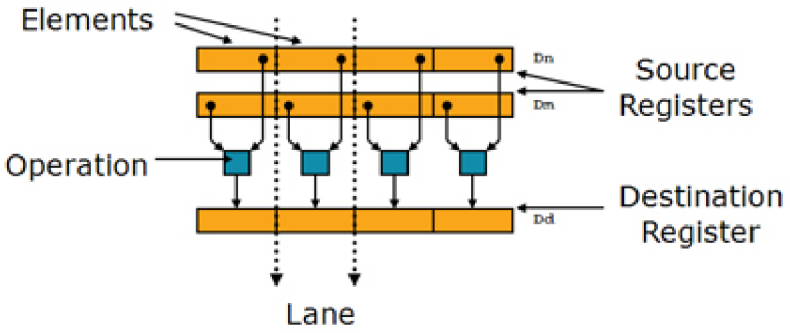
Koprocesor NEON podporuje všechny adresovací módy a operace zpracování dat jako instrukční sada ARMv7-A. Obsahuje dvě samostatné nezávislé linky zřetězeného zpracování, a to jednu pro SIMD a jednu pro FPU. Pro ukládání dat se zde nachází velké 128-bitové sdílené registry. Jsou adresovatelné jako třicet dva 64-bitových a nebo šestnáct 128-bitových registrů. Na obrázku 2.3 je nastíněna činnost koprocesoru při zpracování instrukce. [1]

NEON obsahuje vektorové SIMD registry pro:

- Znaménková a bez znaménková celá čísla.
- Jednobitové koeficientové polynomy.
- Čísla s plovoucí desetinou čárkou s jednoduchou přesností.

K dispozici jsou tyto vektorové operace:

- Sčítání a odčítání.
- Násobení s volitelnou akumulací.
- Vyhledání maxima a minima.
- Aproximace inverzní odmocniny.
- Nahrávání datových struktur.



Obrázek 2.3: Zřetězené zpracování na koprocesor NEON. [1]

2.3.1 Instrukční sada

Koprocesor NEON nabízí velké množství instrukcí. Od základních, jako je sčítání, odčítání, logické funkce, přes výběr minima a maxima až po komprese a dekomprese. Každá instrukce koprocesoru NEON začíná písmenem V, které značí vektor. [17]

Instrukce pro sčítání a odčítání

Ukázka syntaxe instrukcí pro sčítání a odčítání je následující:

$V\{Q\}op\{cond\}.datatype\ \{Qd\},\ Qn,\ Qm$
$V\{Q\}op\{cond\}.datatype\ \{Dd\},\ Dn,\ Dm$
$VopL\{cond\}.datatype\ Qd,\ Dn,\ Dm$
$VopW\{cond\}.datatype\ \{Qd\},\ Qn,\ Dm$

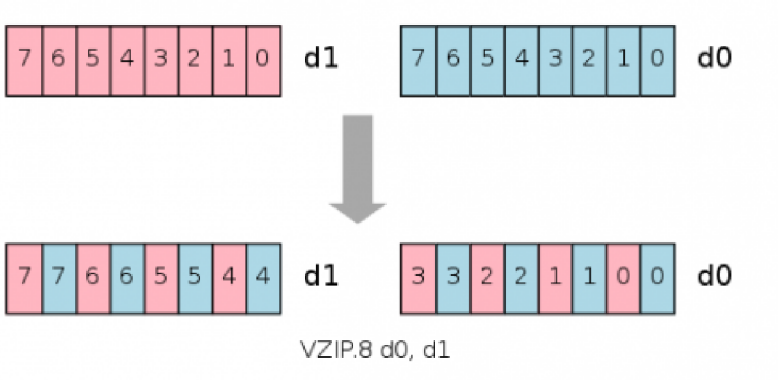
Ve složených závorkách jsou nepovinné parametry. Symboly mají následující význam:

- **Q** – Mění chování výpočtu při přetečení a podtečení. V případě přetečení je je výsledek nastaven na maximální hodnotu rozsahu, v případě podtečení na minimální hodnotu rozsahu.
- **op** – Určuje která operace se použije. Může být buď ADD (sčítání) a SUB (odčítání).
- **cond** – Určuje podmínku pro podmíněné provedení instrukce.
- **datatype** – Určuje datový typ, se kterým se bude pracovat.
- **Qd a Dd** – Určuje cílový registr, do kterého se uloží data. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qn a Dn** – Určuje registr ve kterém je první operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qm a Dm** – Určuje registr ve kterém je druhý operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **L** – Určuje použití takzvané Long instrukce. Jako operandy se použijí 64-bitové čísla a výsledek se uloží do 128-bitového čísla.

- **W** – Určuje použití takzvané Wide instrukce. Jako operand se použije jedno 128-bitové čísla a jedno 64-bitové. Výsledek se uloží do 128-bitového čísla.

Instrukce VZIP

Tato instrukce slouží pro proložení dvou registrů. Princip je ukázán na obrázku 2.4, kde nahoře jsou vstupní dva registry a dole jsou dva registry ve kterých jsou promíchány proměnné ze vstupních registrů. [17]



Obrázek 2.4: Princip funkce VZIP

Instrukce VMAX a VMIN

Instrukce VMAX a VMIN slouží pro nalezení maxima, či minima ze dvou vstupních registrů, které uloží do výstupního registru.

Verze těchto instrukcí VPMAX a VPMIN slouží k porovnání dvou registrů po párech a uložení, v závislosti, která funkce byla použita, většího či menšího čísla do výstupu. Předávají se jim dva vstupní registry a vrátí porovnává vždy dva prvky vedle sebe v jednom registru. V této verzi instrukční sady je varianta porovnávání jen pro 64 bitové registry. [17] Syntaxe je následující:

Vop{*cond*}.*datatype Qd, Qn, Qm*
Vop{*cond*}.*datatype Dd, Dn, Dm*
VPop{*cond*}.*datatype Dd, Dn, Dm*

Ve složených závorkách jsou nepovinné parametry. Symboly mají následující význam:

- **P** – Určuje, že se jedná o porovnávání po dvou prvcích registrů.
- **op** – Určuje která operace se použije. Může být buď MAX (maximum) a MIN (minimum).
- **cond** – Určuje podmínku pro podmíněné provedení instrukce.

- **datatype** – Určuje datový typ, se kterým se bude pracovat.
- **Qd a Dd** – Určuje cílový registr, do kterého se uloží data. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qn a Dn** – Určuje registr ve kterém je první operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qm a Dm** – Určuje registr ve kterém je druhý operand. Q značí 128-bitový registr a D je 64-bitový registr.

Instrukce pro porovnání

Pro porovnání existuje instrukce, která porovná dva vektory a jestli je podmínka výsledku porovnání pravdivá, tak se příslušný element výsledku nastaví na samé jedničky, pokud je nepravdivý, tak na samé nuly. [17]
Syntaxe je následující:

VCop{cond}.datatype Qd, Qn, Qm
VCop{cond}.datatype Dd, Dn, Dm
VCop{cond}.datatype Qd, Qn, #0
VCop{cond}.datatype Dd, Dn, #0

Ve složených závorkách jsou nepovinné parametry. Symboly mají následující význam:

- **op** – Určuje která operace se použije. Může být buď EQ (rovnost), GE (větší rovno), GT (větší než), LE (menší rovno), LT (menší než).
- **cond** – Určuje podmínku pro podmíněné provedení instrukce.
- **datatype** – Určuje datový typ, se kterým se bude pracovat.
- **Qd a Dd** – Určuje cílový registr, do kterého se uloží data. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qn a Dn** – Určuje registr ve kterém je první operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qm a Dm** – Určuje registr ve kterém je druhý operand. Q značí 128-bitový registr a D je 64-bitový registr.

Instrukce pro logické operace

Tato skupina instrukcí slouží pro logické operace, a to pro součin (AND), součet (OR), exklusivní disjunkce (XOR), negovaný součet (NOR) a negovaný součin (NAND). Operace provádí nad dvěma registry a výsledek uloží do cílového registru. [17]
Syntaxe je následující:

Vop{*cond*}{.*datatype*} *Qd, Qn, Qm*
Vop{*cond*}{.*datatype*} *Dd, Dn, Dm*

Ve složených závorkách jsou nepovinné parametry. Symboly mají následující význam:

- **op** – Určuje která operace se použije. AND, ORR (OR), EOR (XOR), BIC (NAND) a ORN (NOR).
- **cond** – Určuje podmínku pro podmíněné provedení instrukce.
- **datatype** – Určuje datový typ, se kterým se bude pracovat.
- **Qd a Dd** – Určuje cílový registr, do kterého se uloží data. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qn a Dn** – Určuje registr ve kterém je první operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qm a Dm** – Určuje registr ve kterém je druhý operand. Q značí 128-bitový registr a D je 64-bitový registr.

Instrukce pro násobení

Pro násobení je určena instrukce *VMUL*. Kromě této instrukce jsou k dispozici i instrukce pro vynásobení a následnému přičtení k výstupnímu registru (*VMLA*), či odečtení (*VMLS*) od výstupního registru. [17]
Syntaxe je následující:

Vop{*cond*}.*datatype Qd, Qn, Qm*
Vop{*cond*}.*datatype Dd, Dn, Dm*
VopL{*cond*}.*datatype Qd, Dn, Dm*

Ve složených závorkách jsou nepovinné parametry. Symboly mají následující význam:

- **op** – Určuje která operace se použije. Může být buď MUL (násobení), MLA (násobení s následnou akumulací), MLS (násobní s následným odečtem).
- **cond** – Určuje podmínku pro podmíněné provedení instrukce.
- **datatype** – Určuje datový typ, se kterým se bude pracovat.

- **Qd a Dd** – Určuje cílový registr, do kterého se uloží data. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qn a Dn** – Určuje registr ve kterém je první operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qm a Dm** – Určuje registr ve kterém je druhý operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **L** – Určuje použití takzvané Long instrukce. Jako operandy se použijí 64-bitové čísla a výsledek se uloží do 128-bitového čísla.

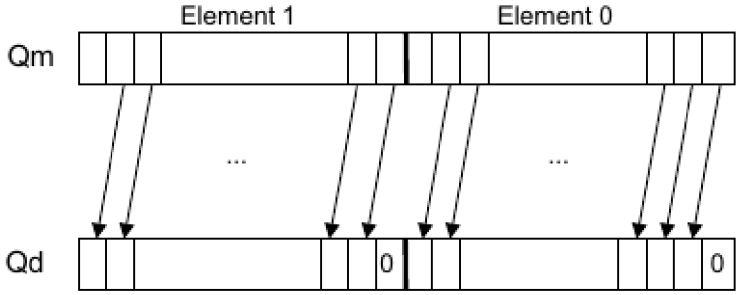
Instrukce pro posun

Pro posun je určena instrukce *VSHL* a její princip je naznacen na obrázku 2.5. [17]
Syntaxe je následující:

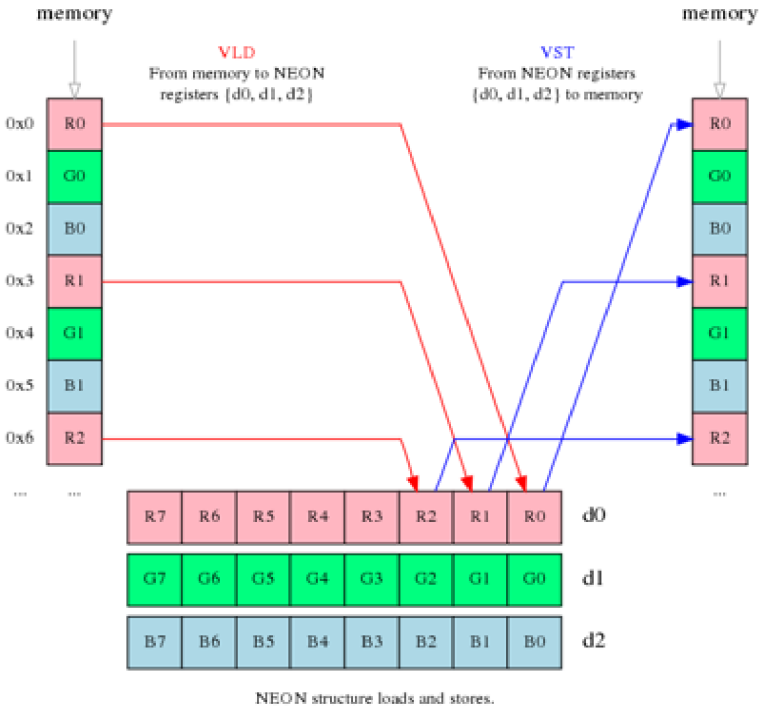
$V\{Q\}SHL\{U\}\{cond\}.datatype\ \{Qd\},\ Qm,\ \#imm$
$V\{Q\}SHL\{U\}\{cond\}.datatype\ \{Dd\},\ Dm,\ \#imm$
$VSHLL\{cond\}.datatype\ Qd,\ Dm,\ \#imm$

Ve složených závorkách jsou nepovinné parametry. Symboly mají následující význam:

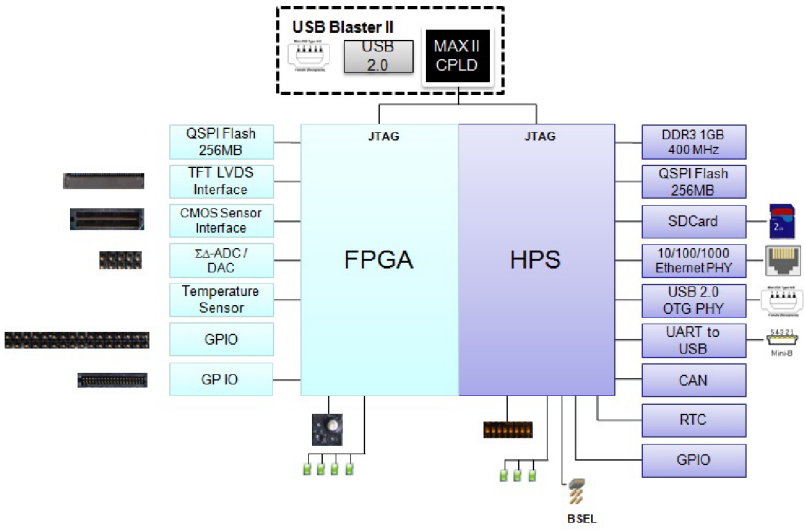
- **Q** – Mění chování výpočtu při přetečení a podtečení. V případě přetečení je je výsledek nastaven na maximální hodnotu rozsahu, v případě podtečení na minimální hodnotu rozsahu.
- **U** – příznak U způsobí, že výsledek bude vždy bez znaménkový i v případě, že jsou použita čísla se znaménkem. Příznak U musí být kombinován s příznakem Q.
- **cond** – Určuje podmínku pro podmíněné provedení instrukce.
- **datatype** – Určuje datový typ, se kterým se bude pracovat.
- **Qd a Dd** – Určuje cílový registr, do kterého se uloží data. Q značí 128-bitový registr a D je 64-bitový registr.
- **Qm a Dm** – Určuje registr ve kterém je první operand. Q značí 128-bitový registr a D je 64-bitový registr.
- **imm** – Je hodnota, o kolik se má registr posunout.



Obrázek 2.5: Princip instrukce VSHL [17]



Obrázek 2.6: Ukázka načítání a ukládání struktury z paměti



Obrázek 2.7: Blokové schéma vývojové desky SoCrates [4]

2.3.2 Načítání a ukládání dat

Při využití koprocesoru NEON se pro načítání a ukládání dat využívají instrukce *vld* a *vst*. Můžou se využít i pro načítání struktur. Například instrukce pro načtení tří prvkové struktury je *vld3*. Princip fungování instrukce *vld3* je vidět na obrázku 2.6, vlevo je paměť, ze které se načítají data, dole jsou znázorněny tři registry a vpravo je paměť, do které se ukládají data. V jazyce symbolických adres se místo struktur používá skupiny registrů.

2.4 Kompilátor GCC

GNU Compiler Collection[1] (zkráceně GCC) je sada překladačů vytvořených v rámci projektu GNU, které podporují řadu jazyků. Je šířen pod licencí GNU GPL. [6]

2.4.1 Intrinsické funkce

V kompilátoru GCC existuje možnost využití instrinsických funkcí. Jedná se o funkce, které odpovídají instrukcím koprocesoru NEON. Jejich syntaxe $< opname >< flags >< type >$, kde opname je název instrukce, flags jsou příznaky a type určuje datový typ operandů. Například funkce *uint16x8_t vaddl_u8(uint8x8_t a, uint8x8_t b)* odpovídá instrukci *VADDL.U8 q0,d0,d0*. Definice těchto funkcí se nachází v knihovně *arm_neon.h*, která je součástí překladače GCC. Kromě funkcí jsou v této knihovně i definice datových typů, které reprezentují registry, například pro 64-bitový registr, ve kterém jsou 8-bitové čísla, odpovídá proměnná *uint8x8_t*. Jsou v ní i definice struktur, které odpovídají skupině registrů. V případě, že třeba potřebujeme tři 64-bitové registry (každý pro jednu barevnou složku), z nichž každý obsahuje osm 8-bitových čísel, můžeme použít strukturu *uint8x8x3_t*. [16]

[1] https://gcc.gnu.org/

2.4.2 Inline assembler

Inline assembler je schopnost přímého vložení kódu napsaného v jazyce symbolických adres do kódu jazyka C. Tato schopnost umožňuje optimalizaci kritických částí programu, aniž by se musel celý přepisovat.

Pro vytvoření části s jazykem symbolických adres, se použije direktiva *asm()*, která musí být nastavena na *volatile*(tuto část kódu nebude kompilátor optimalizovat). Jako parametr se ji předá řetězec, který obsahuje program. Za tímto řetězcem musí být dvojtečka, po které následují výstupní proměnné, pak za další dvojtečkou vstupní proměnné a po poslední dvojtečce je seznam registrů, které jsou v tomto kódu používány. [5]

2.4.3 Automatická vektorizace

GCC nabízí možnost automatické vektorizace. V kódu 2.1 je ukázka zapnutí automatické vektorizace pro procesor, který se nachází na vývojové desce SoCrates 2.2. Automatická vektorizace je součástí optimalizací O3 a Ofast, aby fungovala, tak musí být zapnuty aspoň optimalizace O1. Kompilátor není schopný vektorizovat kód, ve kterém je pole indexováno hranatými závorkami a ukazatele, které se předávají funkci musí být označeny klíčovým slovem *__restrict*. [16]

Kód 2.1: Zapnutí automatické vektorizace

```
gcc -std=gnu99 -O1 -ftree-vectorize -ftree-vectorizer-verbose=1
    -march=armv7-a -mcpu=cortex-a9 -mfpu=neon -mfloat-abi=softfp
    -mvectorize-with-neon-quad jmeno_souboru.c
```

Jednotlivé parametry mají tento význam:

- **O1** – Určuje optimalizace první úrovně.
- **ftree-vectorize** – Zapne automatickou vektorizace.
- **ftree-vectorizer-verbose=1** – Zapne výpisy, ve kterých je řečeno, co se podařilo vektorizovat.
- **march=armv7-a** – Určuje architekturu.
- **mcpu=cortex-a9** – Určuje procesor, pro který se kompiluje.
- **mfpu=neon** – Určuje koprocesor, který je využit.
- **mfloat-abi=softfp** – Určuje využití hardwarové FPU jednotky, ale přes knihovní funkce.
- **mvectorize-with-neon-quad** – Určuje využití 128 bitových registrů.

2.5 Ukázka optimalizace

Na následujícím jednoduchém příkladu viz kód 2.2 si ukážeme možnosti optimalizace za pomoci využití intrinsických funkcí. Jedná se o jednoduchý cyklus, kde se ke každému prvku pole přičítá číslo tři (viz funkce *void Add3 (uint8_t *src, unsigned N)*). Čísla v poli mají datovou šířku 8-bitů, takže při použití 128-bitového registru, můžeme zpracovat 16 položek v jednom taktu. Teoreticky by se měla smyčka zrychlit šestnáctkrát. Nevýhoda je, že velikost pole musí být násobek šestnácti.

Kód 2.2: Ukázka optimalizace

```
void Add3 (uint8_t *src, unsigned N) {
        for(int i = 0; i < N; i++){
                src[i] += 3;
        }
}

void NEONAdd3 (uint8_t *src, unsigned N) {
        uint8x16_t data;                    // definovani registru
        uint8x16_t three = vmovq_n_u8(3);// naplneni trojkama
        unsigned tempN = N / 16;

        for(unsigned i = 0; i < tempN; i++){
                data = vld1q_u8(src);     // nahrani do registru
                data = vaddq_u8(data, three);// vektorovy soucet
                vst1_u8(src, data);       // ulozeni zpet do pameti
                src+= 16;                 // posun ukazatele
        }
}
```

Kapitola 3

Algoritmy

Pro testování byly použity tyto algoritmy: převod na stupně šedi, mediánový filtr, dolní propust, horní propust a sobelův operátor. V této kapitole jsou popsány jednotlivé použité algoritmy včetně ukázek na testovacích obrázcích.

3.1 Konverze na stupně šedi

Tento algoritmus se používá pro převod z barevného obrázku zakódovaném v RGB (tři barevné složky, a to červená, zelená a modrá) na obrázek ve stupních šedi.

Existuje více přístupů, z nichž je nejjednodušší aritmetický průměr všech tří barevných složek, viz vzorec 3.1.

$$I = \frac{R + G + B}{3} \tag{3.1}$$

Lidské oko je různě citlivé na jednotlivé barvy, proto se pro převod používá vzorec 3.2, který nastavuje jednotlivé barevné kanály podle vlastností oka. Výsledky převodu můžeme vidět na obrázku 3.1. [18]

$$I = 0.299 * R + 0.587 * G + 0.114 * B \tag{3.2}$$

Pseudokód naivní implementace algoritmu převodu na stupně šedi, podle vzorce 3.2 můžeme vidět v kódu 3.1. Kde konstanta *height* určuje výšku a konstanta *width* šířku obrázku. Pole *in* obsahuje vstupní obrázek a do pole *out* se ukládá výstup.

Kód 3.1: Pseudokód převodu na stupně šedi

```
for(y = 0; y < height; y++){
  for(x = 0; x < width; x++){
    out[x][y] = in[x][y].red * 77;
    out[x][y] += in[x][y].green * 151;
    out[x][y] += in[x][y].blue * 28;
    out[x][y] = out[x][y] / 256;
  }
}
```

Obrázek 3.1: Vlevo je původní obrázek v RGB. Uprostřed převedený na stupně šedi za pomoci aritmetického průměru. Vpravo převedený za pomoci poměrů barev [13]

3.2 Mediánový filtr

Jedná se o nelineární algoritmus pro odstranění šumu. Vykazuje dobré výsledky například v případě takzvaného šumu soli a pepře (šum černých a bílých bodů v obraze). Mediánový filtr pracuje s předem určeným okolím. Algoritmus pro každý pixel obrazu vezme hodnoty okolních pixelů, včetně daného pixelu a provede nad nimi medián. Medián je prostřední hodnota seřazeného pole. V případě lichého počtu prvků se vezmou dva prostřední prvky a udělá se z nich aritmetický průměr. Na obrázku 3.2 je znázorněn výsledek, při použití masky o velikosti 5×5. [18]

Obrázek 3.2: Vlevo se nachází obrázek, který obsahuje šum. Vpravo je obrázek po použití mediánového filtru [7]

Pro realizaci mediánového filtru existuje více algoritmů. Využil jsem jednoduchý algoritmus, který využívá řazení prvků. Naivní implementace tohoto algoritmu je znázorněna v kódu 3.2. V tomto kódu je význam symbolů následující: *in* je vstupní obrázek, *out* je výstup algoritmu, *window* je pomocné pole, do kterého se uloží prvky pod maskou, *widthOfWindow* a *heightOfWindow* jsou výška a šířka tohoto okna a *sort* je řadící funkce.

Úzké hrdlo algoritmu s řazením prvků je právě toto řazení. Existují rychlejší algoritmy, které například místo řazení používají histogramy.

Kód 3.2: Pseudokód mediánového filtru

```
window[widthOfWindow * heightOfWindow];
startx = widthOfWindow / 2;
starty = heightOfWindow / 2;
for(y = starty; y < height - starty; y++){
  for(x = startx; x < width - startx; x++){
    for(wy = 0; y < heightOfWindow; wy++){
      for(wx = 0; x < widthOfWindow; wx++){
        window[widthOfWindow * wy + wx] =
        in[x + fx - startx ][y + fy - starty];
      }
    }
    sort(window);
    out[x][y] = window[(heightOfWindow * widthOfWindow) / 2];
  }
}
```

3.3 Dolní propust

Dolní propust je lineární filtr, který slouží k potlačení zbytečných detailů obrazu. Používá se na vyhlazování obrazu a odstranění šumu. Funguje na principu průměrování pixelů z okolí. Na určení okolí a váhy pixelů se používá maska, která je následně vydělena součtem všech jejích hodnot [18].

$$W_l = \frac{1}{9} * \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} \tag{3.3}$$

$$W_l = \frac{1}{16} * \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} \tag{3.4}$$

Maska ve vzorci 3.3 uloží průměr z okolí 3×3 a uloží ho do aktuálního pixelu. Na rozdíl od masky ve vzorci 3.3 má maska, která je ve vzorci 3.4 určené váhy okolních pixelů. A to tak, že čím je pixel blíže aktuálnímu, tím větší váhu má. Můžou se použít různě velké masky, které berou v potaz různé množství pixelů. Na obrázku 3.3 je vidět rozdíly při použití různě velkých masek.

Pseudokód pro obecnou masku je znázorněn v kódu 3.3. Jednotlivé proměnné mají tento význam: *height* je výška obrázku, *width* je šířka obrázku, *out* je ukazatel do paměti, kam se má uložit výsledek, *in* je vstupní obrázek, *divNum* je konstanta, kterou se má výsledek matice vydělit (viz vzorec 3.4) *mask* je maska, která se má aplikovat na obrázek, *heightOfMask* a *widthOfMask* je šířka a výška masky.

Kód 3.3: Pseudokód dolní propusti se všeobecnou maskou

```
for(y = 0; y < height; y++){
  for(x = 0; x < width; x++){
    out[x][y] = 0;
    for(yMask = 0; yMask < heightOfMask; yMask++){
      for(xMask = 0; xMask < widthOfMask; xMask++){
        out[x][y]+=in[x+xMask-widthOfMask/2][y+Mask-heightOfMask/2]
                  * mask[xMask][yMask];
      }
    }
    out[x][y] = out[x][y] / divNum;
  }
}
```

Při použití určité masky, v tomto případě masky ve vzorci 3.4 můžeme tento všeobecný kód 3.3 rozepsat do optimalizovanému kódu 3.4.

Kód 3.4: Pseudokód dolní propusti s určitou maskou

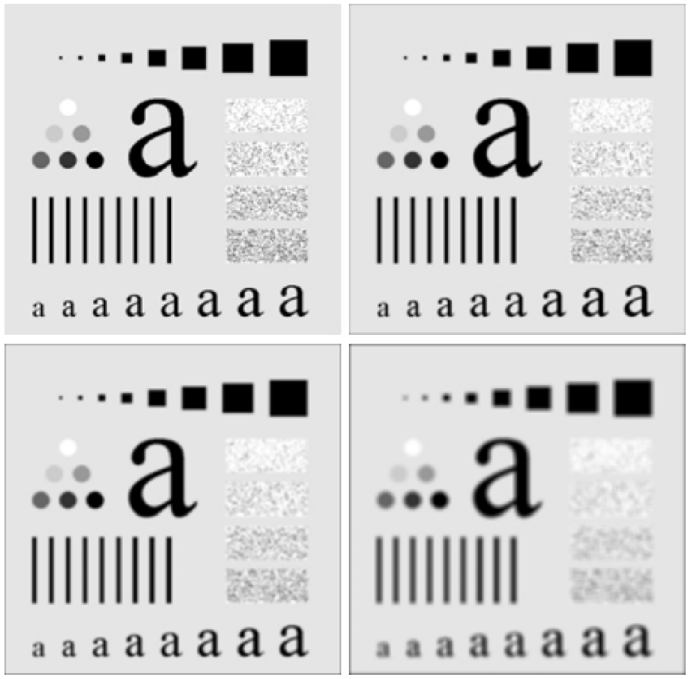
```
for(y = 0; y < height; y++){
  for(x = 0; x < width; x++){
    out[x][y] = (in[x - 1][y -1] * 1 +
    in[x][y -1] * 2 +
    in[x + 1][y -1] * 1 +
    in[x - 1][y] * 2 +
    in[x][y] * 4 +
    in[x + 1][y] * 2 +
    in[x - 1][y + 1] * 1 +
    in[x][y + 1] * 2 +
    in[x + 1][y + 1] * 1) / 16;
  }
}
```

3.4 Horní propust

Algoritmus sloužící pro zostření obrazu, který se řadí mezi lineární filtry. Abychom získali masku pro horní propust, musíme od masky všepásmové propusti, která je ve vzorci 3.5 odečíst masku dolní propusti, která je ve vzorci 3.3. Jedna z možností je maska bez vah, která je ve vzorci 3.6 a nebo maska s vahami viz vzorec 3.7. [18]

$$W_a = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} \tag{3.5}$$

$$W_h = W_a - W_l = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} - \frac{1}{9} * \begin{pmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{pmatrix} = \frac{1}{9} * \begin{pmatrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{pmatrix} \tag{3.6}$$



Obrázek 3.3: Vlevo nahoře je původní obrázek. Následující tři obrázky jsou po použití masky dolní propusti bez vah o velikosti 3×3, 5×5 a 9×9 [9]

$$W_h = W_a - W_l = \begin{pmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 0 \end{pmatrix} - \frac{1}{16} * \begin{pmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{pmatrix} = \frac{1}{16} * \begin{pmatrix} -1 & -2 & -1 \\ -2 & 12 & -2 \\ -1 & -2 & -1 \end{pmatrix} \quad (3.7)$$

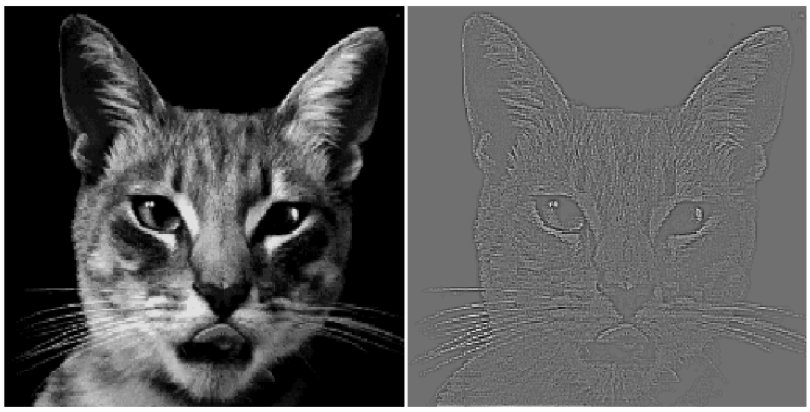
Pro realizaci algoritmu horní propusti lze využít kód 3.3, je by se do funkce předala jiná maska. Pokud kód pro všeobecnou masku, která je ve vzorci 3.3 specifikujeme pro masku, která je ve vzorci 3.7, tak bude výsledek vypadat jak je znázorněno v kódu 3.5. U masky, která je ve vzorci 3.7 se musí hlídat jestli nedošlo k podtečení a pokud ano, musí se aktuální pixel nastavit na nulu.

V kódu 3.5 je význam symbolů následující: *height* je výška obrázku, *width* je šířka obrázku, *out* je ukazatel do paměti, kam se má uložit výsledek, *in* je vstupní obrázek a *temp* je proměnná, do které se ukládá mezivýsledek.

Kód 3.5: Pseudokód horní propusti s určitou maskou

```
for(y = 0; y < height; y++){
  for(x = 0; x < width; x++){
    temp = (in[x - 1][y -1] * -1 +
    in[x][y -1] * -2 +
    in[x + 1][y -1] * -1 +
    in[x - 1][y] * -2 +
    in[x][y] * 12 +
    in[x + 1][y] * -2 +
    in[x - 1][y + 1] * -1 +
    in[x][y + 1] * -2 +
    in[x + 1][y + 1] * -1) / 16;
    if(temp < 0)
      dest[x][y] = 0;
    else
      dest[x][y] = temp;
  }
}
```

Na obrázku 3.4 je výsledek po použití filtru horní propust. Na výsledném obrázku je vidět, že tento filtr zvýrazňuje přechody mezi barvami.



Obrázek 3.4: Vlevo je původní obrázek a vpravo je po použití filtru horní propust [10]

3.5 Sobelův operátor

Sobelův operátor se často využívá pro detekci vodorovných a svislých hran. Pro svislé se používá matice viz vzorec 3.8 a pro vodorovné matice viz vzorec 3.9. Těmito maskami se posouvá po obrazu a pro každý pixel se vypočítá nová hodnota. Pootočením matice viz 3.10 můžeme docílit i detekci šikmých hran. [18]

$$G_y = \begin{pmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{pmatrix} \tag{3.8}$$

$$G_x = \begin{pmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{pmatrix} \tag{3.9}$$

$$G = \begin{pmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{pmatrix} \tag{3.10}$$

Pseudokód vychází z všeobecného kódu pro zpracování masky na obrázku viz kód 3.3. Po rozbalení vnitřních dvou smyček a odstranění řádků, kde je hodnota masky nulová, dostaneme kód 3.6. V tomto kódu může dojít jak k podtečení, tak i přetečení výsledku a proto je potřeba to kontrolovat a případně přepsat výsledek minimální a nebo maximální hodnotou.

Kód 3.6: Pseudokód sobelova operátoru

```
for(y = 0; y < height; y++){
  for(x = 0; x < width; x++){
    temp = in[x - 1][y - 1] * -1 +
    in[x - 1][y] * -2 +
    in[x - 1][y + 1] * -1 +
    in[x + 1][y - 1] * 1 +
    in[x + 1][y] * 2 +
    in[x + 1][y + 1] * 1;
    if(temp < 0)
      dest[x][y] = 0;
    else if(temp > 255)
      dest[x][y] = 255;
    else
      dest[x][y] = temp;
  }
}
```

V kódu 3.6 je význam symbolů následující: *height* je výška obrázku, *width* je šířka obrázku, *out* je ukazatel do paměti, kam se má uložit výsledek, *in* je vstupní obrázek a *temp* je proměnná, do které se ukládá mezivýsledek.

Výsledek tohoto algoritmu můžeme vidět na obrázku 3.5

Obrázek 3.5: Vlevo je původní obrázek a vpravo po aplikaci Sobelova operátoru [12]

Kapitola 4

Návrh a implementace

Program bude jednoúčelová aplikace, která načte obrázek, provede nad ním dané algoritmy, které byly popsány v kapitola 3. Aplikace nebude obsahovat grafické uživatelské rozhraní a bude se ovládat z terminálu. Na cílovém zařízení totiž poběží operační systém, který neumožňuje spouštět takové aplikace (neobsahuje Xserver). Navíc je to zbytečné pro tento typ aplikace.

Program zpracovává barevné obrázky ve formátu BMP. Pro samotné načítání a ukládání obrázků použiji volně dostupnou knihovnu.

Součástí je i soubor Makefile, jenž obsahuje instrukce pro program make, jak má zkompilovat zdrojové kódy. Kompiluje se vícekrát, a to pokaždé s jinými přepínači kompilátoru GCC, aby se ukázalo jaká kombinace je nejvýhodnější.

4.1 Použité nástroje

Jako operační systém pro vývojovou desku SoCrates použiji Linuxovou distribuci Buildroot. Je to skupina skriptů, které stáhnou potřebné zdrojové kódy a zkompilují je pro určenou vývojovou desku. Jsou plně konfigurovatelné a tudíž jde vytvořit distribuci přímo na míru. Kromě operačního systému nabízí i základní nástroje a další jsou volitelné. Vytvoří také kompilátory pro hostitelský počítač, za pomocí nich se můžou zkompilovat zdrojové kódy pro takto vytvořený systém.

Samotná aplikace se bude kompilovat za pomoci GCC pro architekturu ARM. Jedním z cílů této bakalářské práce je vyzkoušet možnosti optimalizace kódu tohoto kompilátoru.

Pro ladění programu jsem použil GDB, který patří do základních nástrojů GNU. Tento program umí kromě ladění programů napsaných v jazyce C i zobrazit kód přeložený do jazyk symbolických adres.

4.2 Knihovny

Při vývoji aplikace jsem použil tři knihovny. Jako standardní knihovnu jsem použil uClibc. Tato knihovna obsahuje všechny potřebné funkce, ale je mnohem menší než pro Linux typická GNU C Library. Tuto knihovnu jsem využil pro práci se standardním vstupem a výstupem, práci s řetězci a měření času.

Dále jsem využil knihovnu *arm_neon.h* pro koprocesor NEON, která je součástí kompilátoru GCC. Tato knihovna obsahuje definice proměnných a funkcí pro procesor ARM. Také pak v ní najdeme definici intrinsických funkcí.

Jako poslední jsem použil knihovnu pro načítání a ukládání obrázku ve formátu BMP, kterou jsem převzal od Malcolm Mclean [3]. V knihovně se nachází dvě jednoduché funkce, a to pro načítání *loadbmp()* a pro ukládání *savebmp()*. Jako parametr se jim předává jméno souboru a vrací pole s daty, výšku a šířku obrázku.

4.3 Provádění testů

Délku provádění algoritmů jsem se rozhodl měřit za pomoci časovačů. Ve standardní knihovně je hlavičkový soubor *time.h*, který nabízí funkce pro práci s časem. Využil jsem funkci *clock_gettime()*, která načte aktuální čas do struktury *timespec*. Tato struktura, jejíž definice je taktéž v *time.h*, obsahuje dvě položky, jež jsou sekundy a nanosekundy.

Samotné měření je naznačeno v kódu 4.1. Jednotlivé symboly mají význam: *clocl_gettime()* je funkce pro získání aktuálního času, *start* a *stop* jsou struktury typu *struct timespec* a ukládá se v nich čas, *out* je výstupní pole, *pic* je vstupní obrázek, *w* a *h* jsou šířka a výška obrázku a do *time* se uloží výsledný čas.

Kód 4.1: Měření času

```
if(clock_gettime(CLOCK_REALTIME, &start) == -1 ){
    printf( "clock gettime error" );
    return -1;
}
algorithm(out, pic, w, h);
if(clock_gettime(CLOCK_REALTIME, &stop) == -1 ){
    printf( "clock gettime error" );
    return -1;
}

time = ( stop.tv_sec - start.tv_sec ) * BILLION
        + ( stop.tv_nsec - start.tv_nsec );
```

Pro zpřehlednění jsem vytvořil funkci *doTest()*, které se předají názvy souborů se vstupním, výstupním obrázkem a odkaz na požadovanou funkci, která se má testovat. Začne s tím, že načte obrázek za pomoci funkce *loadbmp()*. Následně vytvoří a vynuluje výstupní pole, které poslouží jak výstup pro funkci s algoritmem. Uloží aktuální čas a zavolá funkci s algoritmem. Po dokončení vypočítá délku trvání a uloží obrázek. Nakonec vrátí naměřený čas.

4.4 Algoritmy

Každý algoritmus je implementován ve svém souboru, kde kromě daného algoritmu jsou i pomocné funkce. Ke každému souboru s koncovkou .c je i odpovídající hlavičkový soubor (přípona .h), které obsahují definice funkcí, které je možné volat z jiného souboru.

4.4.1 Převod na stupně šedi

Převod na stupně šedi vychází z kódu 3.1. Vytvořil jsem tři funkce a to v jazyce C, jazyku symbolických adres a jazyce C s intrinsickými funkcemi. V první funkci, která je napsaná v jazyce C jsem na rozdíl od vzorce 3.2 upravil konstanty, kterými se násobí jednotlivé barevné

složky pixelů viz kód 4.2. Konstanty jsem vynásobil 256 a po výpočtu jsem výsledek zrotoval o 8 bitů doprava. Pro toto řešení jsem se rozhodl z důvodu, abych se vyhnul využití FPU jednotky a dělení. Výsledek se uloží do každé barevné složky.

Kód 4.2: Výpočet převodu na stupně šedi v jazyce C

```
int r = pic[i]; // load red
int g = pic[i+1]; // load green
int b = pic[i+2]; // load blue
// build weighted average:
int y = (r*77)+(g*151)+(b*28);
// undo the scale by 256 and write to memory:
dest[i] = (y>>8);
dest[i+1] = dest[i];
dest[i+2] = dest[i];
```

Výpočet za pomoci intrinsických funkcí je naznačen v kódu 4.3, význam symbolu je následující: struktura *vector* obsahuje vstupní data a je typu *uint8x8x3_t*, proměnná *temp* je typu *uint16x8_t* a ukládá se do ní mezivýsledek, proměnné *rfac*, *gfac* a *bfac* jsou typu *uint8x8_t* a obsahují konstanty pro jednotlivé barevné složky a *outVec* je struktura do které se ukládá výsledek.

Kód 4.3: Výpočet převodu na stupně šedi za pomoci intrinsických funkcí

```
int end = (width * height) / 8;

for (int i=0; i < end; ++i){

  uint8x8x3_t vector  = vld3_u8 (pic);

  temp = vmull_u8 (vector.val[0],       rfac);
  temp = vmlal_u8 (temp,vector.val[1], gfac);
  temp = vmlal_u8 (temp,vector.val[2], bfac);

  outVec.val[0] = vshrn_n_u16 (temp, 8);
  outVec.val[1] = outVec.val[0];
  outVec.val[2] = outVec.val[0];
  vst3_u8 (dest, outVec);
  pic  += 8 * 3;
  dest += 8 * 3;
}
```

Pro samotný výpočet jsem využil funkce *vmull_u8()* pro první vynásobení a pak *vmlal_u8()*, která vynásobí dvě čísla a přičte je k předchozímu výsledku. Na výsledek těchto funkcí jsem použil funkci *vshrn_n_u16()*, která provede posun o 8 bitů doprava (podělí 256) a uloží jen spodních 8 bitů.

V kódu 4.4 jde vidět, že implementace v jazyce symbolických adres vychází z kódu napsaného za pomoci intrinsických funkcí. Jediné rozdíly jsou v použití registrů místo proměnných a ve smyčce.

Kód 4.4: Výpočet převodu na stupně šedi v jazyce symbolických adres

```
.loop:
vld3.8      {d0-d2}, [%1]!

vmull.u8    q7, d0, d4
vmlal.u8    q7, d1, d5
vmlal.u8    q7, d2, d6

vshrn.u16   d0, q7, #8
vmov.u8     d1, d0
vmov.u8     d2, d0
vst3.8      {d0-d2}, [%0]!
subs        %2, %2, #1
bne         .loop
```

4.4.2 Dolní propust, horní propust a Sobelův operátor

Tyto algoritmy jsou si velice podobné. Na všechny by se dal použít pseudokód 3.3 jenom by se použila jiná maska a bylo by potřeba hlídat podtečení a přetečení výsledku. Rozhodl jsem se však použít pro každý algoritmus specializované kódy. Pro dolní propust kód 3.4, horní propust kód 3.5 a pro Sobelův operátor kód 3.6.

V kódu 4.5 je znázorněn výpočet Sobelova operátoru za pomoci intrinsických funkcí, kde je význam symbolů následující: struktury *vector*, jenž jsou typu *uint8x8x3_t* obsahují načtená data ze vstupního obrázku, *i* označuje, která barevná složka se zrovna zpracovává, *temp* a *temp2* je typu *uint16x8_t* a slouží jako pomocná proměnná a *two*, *one*, *zero* a *consta* jsou typu *uint16x8_t* a slouží jako konstanty.

Kód 4.5: Výpočet Sobelova operátoru za pomoci instinsických funkcí

```
temp = vmull_u8 (vector7.val[i], one);
temp = vmlal_u8 (temp,vector8.val[i], two);
temp = vmlal_u8 (temp,vector9.val[i], one);
temp = vmlsl_u8 (temp,vector1.val[i], one);
temp = vmlsl_u8 (temp,vector2.val[i], two);
temp = vmlsl_u8 (temp,vector3.val[i], one);

temp2 = vcltq_u16(temp, zero);
temp = vandq_u16(temp, temp2);

temp2 = vcltq_u16(temp, consta);
temp = vornq_u16(temp, temp2);

outVec.val[i] = vmovn_u16(temp);
```

V případě jazyku symbolických adres kód vychází z implementace kódu za pomoci instrinsických funkcí. Instrukce pro násobení, nemá příznak Q, který by automaticky hlídal meze. Z 32 registrů jsem u dolní propusti využil 27 registrů pro vstupní data (9 prvků v masce a 3 barvy na každý pixel). Pak bylo potřeba do 2 registrů ukládat mezivýsledek, protože se jednalo o 16 bitové číslo. Pak už zbyly jen tři registry, které jsem využil na

konstanty v masce.

U Sobelova operátoru jsem ušetřil 9 registrů díky tomu, že jsou v masce tři nulové prvky. Ale z nich jsem 6 musel použít na porovnávání (porovnávaly se 16 bitové čísla).

U horní propusti jsem narazil na problém nedostatku registrů. Bylo jich potřeba použít stejně jako v případě dolní propusti, ale musel jsem hlídat podtečení. Abych toto mohl udělat, potřeboval jsem další čtyři registry (porovnávání 16 bitových čísel). Proto jsem musel konstanty z masky po každém porovnání znovu načítat a musel jsem si zálohovat jeden registr s daty. Navíc před každým porovnáním bylo potřeba nahrát konstantu 256, která se používala pro porovnávání.

4.4.3 Mediánový filtr

Při psání kódu jsem vycházel ze kódu 3.2, s tím, že jsem rozbalil dvě vnitřní smyčky, které slouží pro procházení maskou. V jazyce C jsem využil dva algoritmy pro řazení pole. A to Quick sort, který se nachází ve standardní knihovně jako funkce *qsort()*. Její parametry jsou neseřazené pole, počet prvků v poli, velikost jednoho prvku a porovnávací funkce. Tu jsem definoval ve funkci *comp()*. Jedná se o jednoduchá porovnání dvou prvků typu *uint8_t* (bez-znaménkové 8 bitové číslo). Druhý řadící algoritmus, který jsem použil je Bitonic sort pro 9 čísel. [2]

U intrinsických funkcí jsem kvůli zvýšení přehlednosti kód rozdělil do více funkcí, které jsem označil jako *inline*. Pro řazení jsem použil řadící algoritmus Bitonic sort pro 9 čísel. Je to řazení vhodné pro paralelizaci, které je založeno na řadících sítích. Ve funkci *sortNEON()* najdeme volání jednotlivých porovnání. Samotné porovnávání je ve funkci *getMaxMin()*, která je ukázaná v kódu 4.6, kde význam symbolů je následující: *a* a *b* jsou vstupní dva vektory, které se mají porovnat a zároveň slouží jako výstup a *pom* je pomocná proměnná.

Kód 4.6: Porovnání dvou vektorů

```
uint8x8x2_t pom;
pom = vzip_u8(*a, *b);
*a = vpmin_u8(pom.val[0], pom.val[1]);
*b = vpmax_u8(pom.val[0], pom.val[1]);
```

Při implementaci kódu v jazyce symbolických adres jsem vycházel z kódu využívajícího intrinsické funkce. Největší rozdíl mezi těmito kódy je ten, že v jazyce symbolických adres není kód rozdělen do více funkcí. Pro tento krok jsem se rozhodl nejen kvůli vyšší rychlosti, ale hlavně proto, že by tato funkce měla jen čtyři instrukce (plus pár dalších kvůli režie) a pro zavolání by bylo potřeba tři instrukce. Dvě pro načtení vstupních parametrů a jednu pro samotné zavolání. Tudíž by se tato varianta moc nevyplatila, ani vzhledem k délce kódu. Další rozdíl oproti implementaci s intrinsickými funkcemi je ten, že je rozbalena smyčka pro jednotlivé barevné složky.

4.5 Spuštění programu

Příkaz *make* vezme zdrojové kódy a zkompiluje je pětkrát (různé úrovně optimalizace). Vznikne tedy pět spustitelných souborů a to neon, neonO1, neonO2, neonO3, neonOfast.

Programu se při spuštění jako argumenty předají vstupní obrázky. Při běhu vypisuje na standardní výstup název aplikace, jaký vstupní soubor se zpracovává a jednotlivé výsledky pro každý algoritmus a výsledné obrázky uloží do složky *output*. Formát názvu výstupních obrázků je číslo (které reprezentuje kolikátý vstupní obrázek se zpracoval), název algoritmu

a jaká metoda se použila. Toto může být buď NEON (intrinsické funkce), NEONasm (jazyk symbolických adres) a nebo bez pro čistý jazyk C. Na obrázku 4.1 můžeme vidět aplikaci po skončení výpočtů.

```
root@buildroot:~/bp$ ./neon ./image/cLena-640x640.bmp
./neon
./image/cLena-640x640.bmp
prevod na stupne sedi - ciste C: 37675530
prevod na stupne sedi - intrinsicke funkce: 18288670
prevod na stupne sedi - assembler: 4840000
dolni propust - ciste C:309854260
dolni propust - intrinsicke funkce: 103256510
dolni propust - assembler: 8447640
horni propust - ciste C: 333013010
horni propust - intrinsicke funkce: 120389720
horni propust - assembler: 10591080
sobeluv operator - ciste C: 229378210
sobeluv operator - intrinsicke funkce: 98930250
sobeluv operator - assembler: 9053960
medianovy filter quick sort - ciste C: 1869841570
medianovy filter bubble sort - ciste C: 2404897320
medianovy filter - intrinsicke funkce: 493724950
medianovy filter - assembler: 28197750
```

Obrázek 4.1: Aplikace po skončení výpočtů

Kapitola 5

Výsledky

Tato kapitola obsahuje výsledky měření jednotlivých algoritmů. U překladu byly použity optimalizace O1, O2, O3, Ofast, bez optimalizace a automatická vektorizace. Testoval jsem na sadě obrázků, která obsahuje různé obrázky o rozdílných velikostech, a to 128*128 pixelů, 256*256 pixelů, 512*512 pixelů, 640*640 pixelů a 900*900 pixelů. Náhledy použitých obrázků jsou ukázány na obrázku 5.1. Postupně se načítaly jednotlivé obrázky a použily se na ně implementované algoritmy. Měření jsem opakoval dvacetkrát, výsledky jsem zprůměroval a sečetl trvání výpočtu pro všechny obrázky.

Program vypisuje čísla v nanosekundách a kvůli zvýšení přehlednosti jsem tyto nanosekundy převedl na milisekundy a zaokrouhlil na tři desetinná místa. Tyto čísla pak najdeme v tabulkách u každého algoritmu. Vytvořil jsem další tabulky, kde čísla odpovídají tomu, kolikrát je daná metoda rychlejší oproti neoptimalizovanému kódu, napsaného v jazyce C.

Obrázek 5.1: Náhledy použitých obrázků

5.1 Konverze na stupně šedi

V tabulce 5.1 a tabulce 5.2 lze vidět, že kód napsaný v jazyce symbolických adres vykazuje stabilní urychlení výpočtu díky označení kódu jako *volatile*, a to přibližně 7,5 krát. Kód s

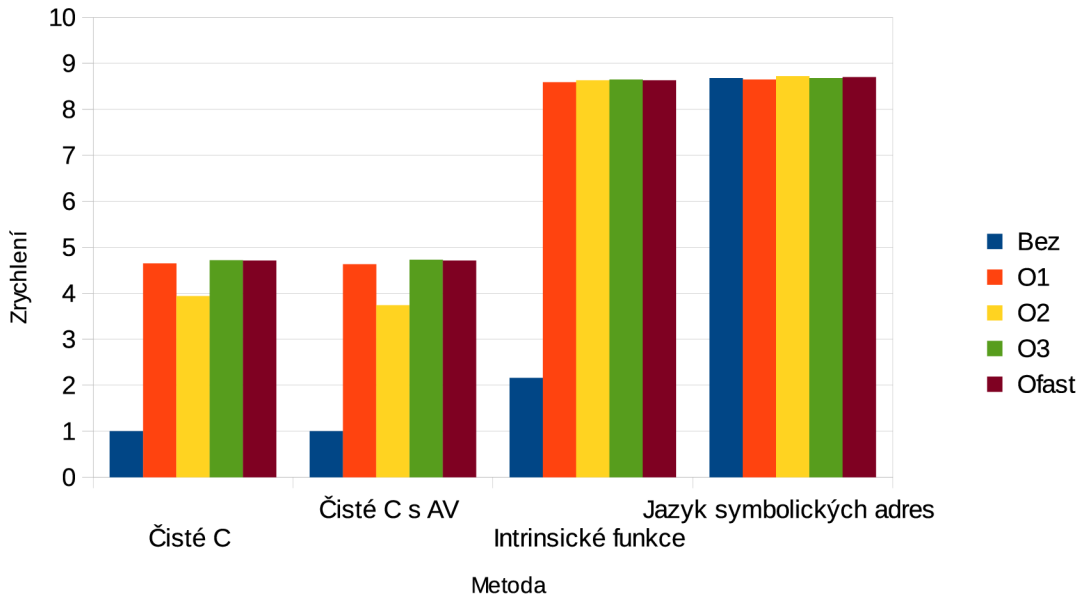
intrinsickými funkcemi od optimalizace O1 dosahuje stejných výsledků jako kód napsaný v jazyce symbolických adres. V jazyce C jde jen za pomoci optimalizací dosáhnout kódu rychlejší 4.7 krát (u optimalizace O3 a Ofast). Automatické vektorizace u tohoto kódu fungovala, ale kód, který vytvořila, nebyl rychlejší a dokonce v případě optimalizace O2 byl pomalejší o 5%. Automatická vektorizace vytvořila kód, který měl skoro 250 řádků a oproti tomu kód, který jsem já napsal v jazyce symbolických adres má jen 23 řádků. Tabulka 5.2 je znázorněna v grafu 5.2.

Čisté C (ms)	160,720	34,514	40,763	34,072	34,182
Čisté C s AV (ms)	160,763	34,638	42,775	33,956	34,103
Intrinsické funkce (ms)	74,409	18,537	18,432	18,389	18,436
Jazyk symbolických adres (ms)	18,336	18,392	18,256	18,333	18,279

Tabulka 5.1: Konverze na stupně – celkový čas

Optimalizace	Bez	O1	O2	O3	Ofast
Čisté C (krát)	1,00	4,65	3,94	4,72	4,71
Čisté C s AV (krát)	1,00	4,63	3,74	4,73	4,71
Intrinsické funkce (krát)	2,16	8,59	8,63	8,65	8,63
Jazyk symbolických adres (krát)	8,68	8,65	8,72	8,68	8,70

Tabulka 5.2: Konverze na stupně – celkové zrychlení



Obrázek 5.2: Zrychlení u konverze na stupně šedi

5.2 Mediánový filtr

Výsledky Mediánového filtru jsou zobrazeny v tabulce 5.3 a tabulce 5.4, která je znázorněna v grafu 5.3. V případě využití funkce pro řazení ze standardní knihovny byly schopnosti kompilátoru optimalizovat velmi malé a kvůli tomu při zapnutých optimalizacích dopadl hůře než Bitonic sort, který už neoptimalizovaný byl rychlejší o 26%. Při zapnutých optimalizacích O3 a Ofast dosáhl algoritmus využívající Bitonic sort zrychlení až o 3,8%. Automatická vektorizace u Mediánového filtru nefungovala.

Kód v jazyce symbolických adres byl 60 krát rychlejší než kód v jazyce C s řadícím algoritmem Quick sort. U inrinsických funkcí došlo k velkému poklesu výkonu oproti jazyku symbolických adres, a to hlavně v neoptimalizovaném kódu. K tomuto poklesu došlo kvůli rozdělení algoritmu do více funkcí.

Čisté C – Quick sort (ms)	6593,822	7527,928	7946,822	8318,693	7869,804
Čisté C s AV - Quick sort (ms)	6587,817	7518,646	7800,748	8301,741	7865,919
Čisté C – Bitonic sort (ms)	5229,458	1853,750	1758,335	1687,451	1699,215
Čisté C s AV - Bitonic sort (ms)	5228,221	1845,630	1735,335	1684,574	1703,512
Intrinsické funkce (ms)	1918,009	280,640	251,095	250,966	250,809
Jazyk symbolických adres (ms)	108,216	108,156	108,294	108,206	108,176

Tabulka 5.3: Mediánový filtr – celkový čas

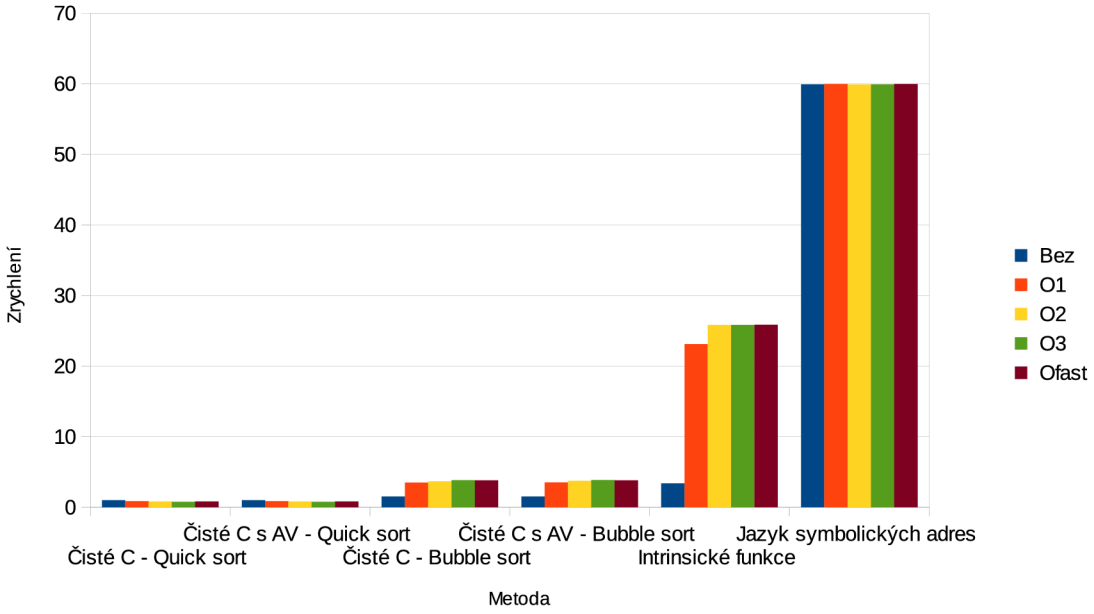
Optimalizace	Bez	O1	O2	O3	Ofast
Čisté C – Quick sort (krát)	1,00	0,86	0,81	0,78	0,82
Čisté C s AV - Quick sort (krát)	1,00	0,86	0,83	0,78	0,82
Čisté C – Bitonic sort (krát)	1,26	3,50	3,69	3,84	3,82
Čisté C s AV - Bitonic sort (krát)	1,26	3,51	3,74	3,85	3,81
Intrinsické funkce (krát)	3,38	23,11	25,83	25,84	25,86
Jazyk symbolických adres (krát)	59,92	59,95	59,89	59,92	59,95

Tabulka 5.4: Mediánový filtr – celkové zrychlení

5.3 Dolní propust

Výsledky měření algoritmu Dolní propust jsou zobrazeny v tabulce 5.5 a tabulce 5.6, která je znázorněna v grafu 5.4. Za pomoci optimalizací se podařilo zrychlit kód napsaný v jazyce C až 12,5 krát. Automatická vektorizace nebyla schopna nic vektorizovat a proto má stejné výsledky jako kód bez automatické vektorizace.

Kód napsaný v jazyce symbolických adres byl 37 krát rychlejší, než neoptimalizovaný kód v jazyce C. Kód využívající intrinsické funkce při optimalizacích O3 a Ofast byl rychlejší, než kód napsaný v jazyce symbolických adres, a to o 6%. Při porovnání kódu napsaného v jazyce symbolických adres a přeloženého (optimalizovaného) kódu využívajícího intrinsické funkce, vyšlo najevo, že na rozdíl od mého kódu, kde se data načtou před samotným výpočtem, kdežto vygenerovaný kód načítá data postupně během výpočtu.



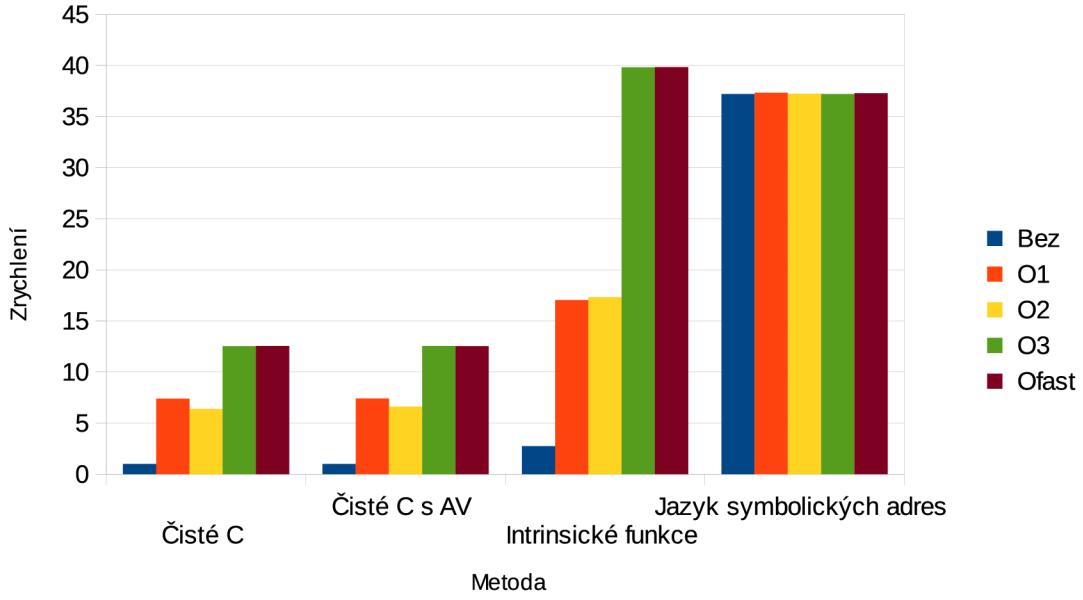
Obrázek 5.3: Zrychlení u Mediánového filtru

Čisté C (ms)	1209,850	163,626	189,685	96,570	96,433
Čisté C s AV (ms)	1209,939	163,570	183,161	96,369	96,518
Intrinsické funkce (ms)	443,124	70,996	69,821	30,289	30,274
Jazyk symbolických adres (ms)	32,442	32,352	32,463	32,495	32,390

Tabulka 5.5: Dolní propust – celkový čas

Optimalizace	Bez	O1	O2	O3	Ofast
Čisté C (krát)	1,00	7,39	6,38	12,53	12,54
Čisté C s AV (krát)	1,00	7,40	6,60	12,55	12,53
Intrinsické funkce (krát)	2,73	17,04	17,32	39,81	39,83
Jazyk symbolických adres (krát)	37,21	37,33	37,23	37,20	37,28

Tabulka 5.6: Dolní propust – celkové zrychlení



Obrázek 5.4: Zrychlení u dolní propusti

5.4 Horní propust

Výsledky měření algoritmu Horní propust jsou zobrazeny v tabulce 5.7 a tabulce 5.8, která je znázorněna v grafu 5.5. Kód napsaný v jazyce C už při optimalizaci O1 vykazoval zrychlení 8 krát a při Ofast až 10 krát. Automatická vektorizace nebyla schopna vektorizovat ani jednu smyčku.

V kódu s intrinsickými funkcemi za pomoci optimalizací se dalo docílit zrychlení až 41 krát, což je lepší výsledek, než jaký měl kód napsaný v jazyce symbolických adres, který dosahoval zrychlení 32 krát. Při psaní kódu Horní propusti jsem narazil na nedostatek registrů. Kompilátor vytvořil kód ve kterém instrukcím změnil pořadí tak, aby mohl data načítat postupně během výpočtu, tyto úpravy způsobily 27% nárůst rychlosti oproti kódu v jazyce symbolických adres.

Optimalizace	Bez	O1	O2	O3	Ofast
Čisté C (ms)	1317,715	165,705	219,703	134,824	134,971
Čisté C s AV (ms)	1317,737	165,874	224,329	134,860	134,996
Intrinsické funkce (ms)	523,256	77,693	76,390	32,184	32,101
Jazyk symbolických adres (ms)	40,987	40,984	40,991	40,913	40,955

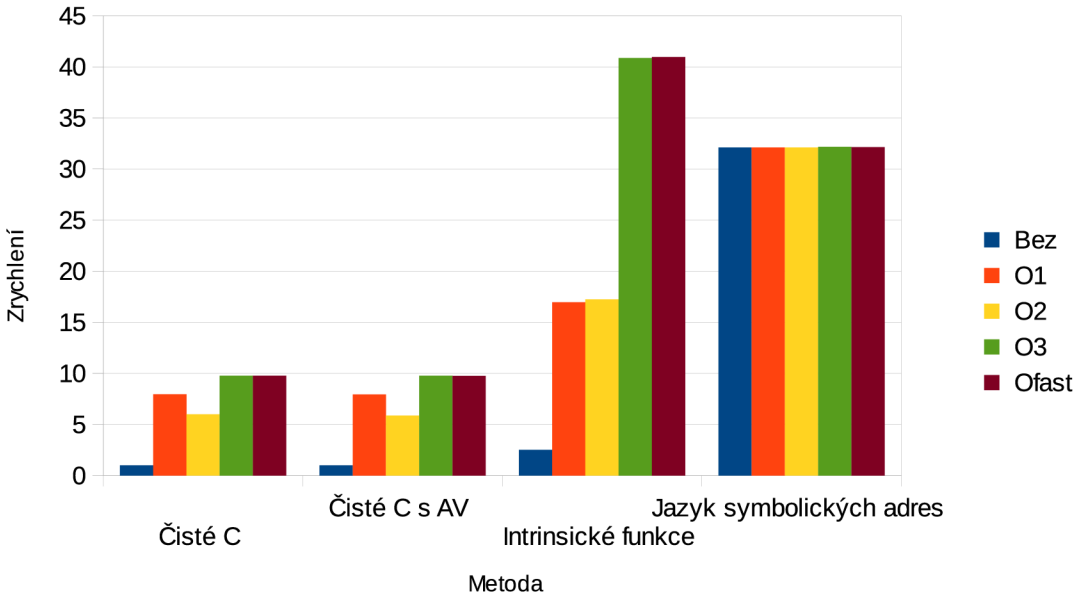
Tabulka 5.7: Horní propust – celkový čas

5.5 Sobelův operátor

Výsledky měření algoritmu Horní propust jsou zobrazeny v tabulce 5.9 a tabulce 5.10, která je znázorněna v grafu 5.6. Kód napsaný v jazyce C za pomoci optimalizací dosáhl zrychlení až 8 krát. Automatická vektorizace nebyla schopna vektorizovat kód.

Optimalizace	Bez	O1	O2	O3	Ofast
Čisté C (krát)	1,00	7,95	6,00	9,77	9,77
Čisté C s AV (krát)	1,00	7,94	5,87	9,77	9,76
Intrinsické funkce (krát)	2,52	16,96	17,25	40,87	40,97
Jazyk symbolických adres (krát)	32,12	32,12	32,11	32,18	32,14

Tabulka 5.8: Horní Propust – celkové zrychlení



Obrázek 5.5: Zrychlení u horní propusti

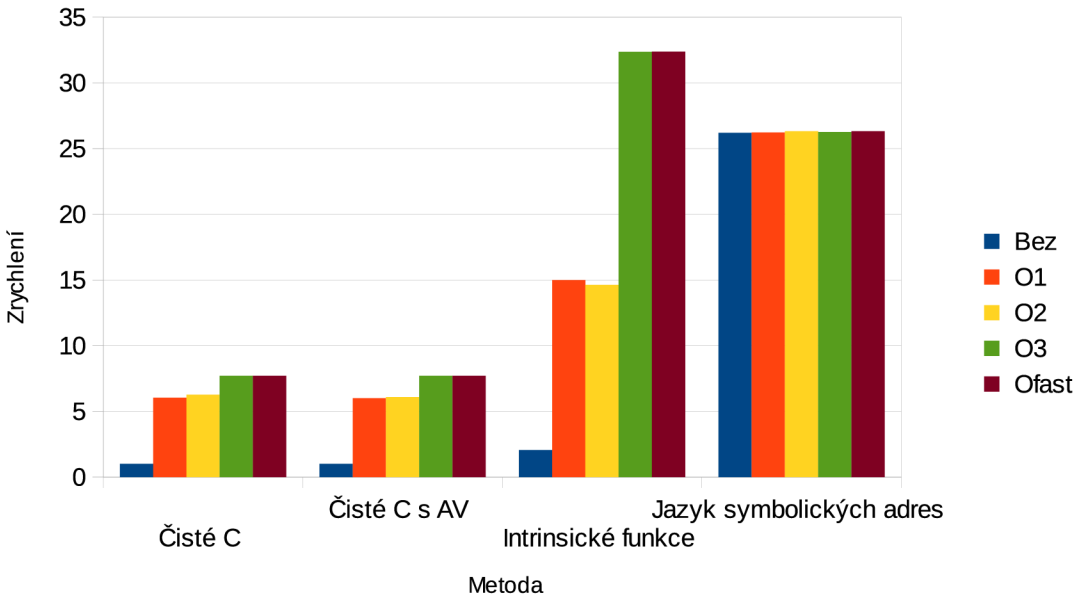
Kód napsaný v jazyce symbolických instrukcí dosáhl zrychlení 26 krát oproti neoptimalizovanému kódu v jazyce C. Kód s intrinsickými funkcemi dosáhl při optimalizaci O3 a Ofast zrychlení 32 krát, což je o 23% lepší výsledek než kód napsaný v jazyce symbolických instrukcí. Při pohledu do vygenerovaného kódu jsem zjistil, že tohoto zrychlení kompilátor dosáhl díky změně pořadí provádění instrukcí, které mu umožnilo načítat data během samotného výpočtu.

Čisté C (ms)	915,263	151,654	145,727	118,856	118,738
Čisté C s AV (ms)	915,371	152,646	150,512	118,694	118,772
Intrinsické funkce (ms)	444,904	61,047	62,532	28,200	28,172
Jazyk symbolických adres (ms)	34,861	34,832	34,720	34,827	34,751

Tabulka 5.9: Sobelův operátor – celkový čas

Optimalizace	Bez	O1	O2	O3	Ofast
Čisté C (krát)	1,00	6,03	6,27	7,70	7,71
Čisté C s AV (krát)	1,00	6,00	6,08	7,71	7,71
Intrinsické funkce (krát)	2,06	14,99	14,63	32,36	32,38
Jazyk symbolických adres (krát)	26,21	26,23	26,32	26,27	26,32

Tabulka 5.10: Sobelův operátor – celkové zrychlení



Obrázek 5.6: Zrychlení u Sobelova operátoru

Kapitola 6

Závěr

Ve své práci jsem se zabýval schopnostmi a možnostmi využití koprocesoru NEON. Pro potřeby návrhu aplikace jsem prostudoval architekturu, programovací jazyky a nástroje pro koprocesor NEON. Zjištěné poznatky jsou uvedeny v kapitole 2. Dále jsem prostudoval grafické algoritmy, které se měly akcelerovat na koprocesoru NEON. Vysvětlení, popis a ukázky jednotlivých algoritmů se nacházejí v kapitole 3. Vytvořil jsem konsolový program, který provádí tyto grafické algoritmy, za pomoci různých metod a měří délku výpočtu, viz kapitola 4.

Koprocesor NEON se ukázal jako velmi výkonný. Při využití jazyka symbolických adres a nebo instrinsických funkcí (Při optimalizaci O3 a nebo Ofast) by všechny algoritmy, kromě Mediánového filtru, zvládly v reálném čase zpracovávat HD video a převod na stupně šedi by zvládl i fullHD video.

Automatická vektorizace nebyla moc účinná. I přes úpravy kódu, aby bylo možné provést automatickou vektorizaci buď kompilátor nedokázal nic vektorizovat a nebo vytvořil velký komplikovaný kód, který byl ještě pomalejší než původní verze. Je na zvážení, zda by automatická vektorizace nebyla účinnější, pokud by byla provedena větší úprava kódu, tak aby jim překladač lépe rozuměl.

Jako nejvýhodnější možnost využití koprocesoru NEON se jeví intrinsické funkce s optimalizacemi O3 a nebo Ofast. Ve většině případů dosahují podobných a nebo i lepších výsledků než jazyk symbolických adres. Samotné programování s využitím intrinsických funkcí je mnohem jednoduší než psát program v jazyce symbolických adres a to zvláště při urychlování kritických částí.

Literatura

[1] ARM – The Architecture for the Digital World [online]. http://www.arm.com, 2014 [cit. 2015-1-28].

[2] Bitonic sort [online]. http://www.iti.fh-flensburg.de/lang/algorithmen/sortieren/bitonic/bitonicen.htm, 2014 [cit. 2015-1-28].

[3] Data Density Program [online]. http://www.malcolmmclean.site11.com/www/datadensity/DataDensity.html, 2014 [cit. 2015-1-28].

[4] EBV Socrates [online]. http://www.rocketboards.org/foswiki/Documentation/SoCrates, 2015 [cit. 2015-1-28].

[5] Extended Asm – Using the GNU Compiler Collectiob (GCC) [online]. https://gcc.gnu.org/onlinedocs/gcc/Extended-Asm.html, 2015 [cit. 2015-1-28].

[6] GCC, the GNU Compiler Collection [online]. https://gcc.gnu.org/, 2015 [cit. 2015-1-28].

[7] Minimum, maximum and median filters – Graphics Mill [online]. http://www.graphicsmill.com/docs/gm/minimum-maximum-median-filters.htm, 2015 [cit. 2015-1-28].

[8] NEON - ARM [online]. http://www.arm.com/products/processors/technologies/neon.php, 2015 [cit. 2015-1-28].

[9] OpenCV w Pythonie [online]. http://www.fizyka.umk.pl/ gniewko/python/OpenCV_przetwarzanie, 2015 [cit. 2015-1-28].

[10] Sharpening[online]. http://fourier.eng.hmc.edu/e161/lectures/gradient/node1.html, 2015 [cit. 2015-1-28].

[11] SoCrates: SoC has Landed [online]. http://www.ebv.com/products/product-details/5797/SoCrates, 2015 [cit. 2015-1-28].

[12] State of The Art Report on GPU [online].
https://www.comp.leeds.ac.uk/viznet/reports/GPU_report
/GPUSTARReport_html.html, 2015 [cit. 2015-1-28].

[13] Steloflute [online]. http://steloflute.tistory.com/681, 2015 [cit. 2015-1-28].

[14] ARM: Cortex-A9 NEON Media Processing Engine: Technical Reference Manual [online]. infocenter.arm.com/help/topic/com.arm.doc.ddi0409e
/DDI0409E_cortex_a9_neon_mpe_r2p0_trm.pdf, 2009 [cit. 2015-1-28].

[15] ARM: Cortex-A9: Technical Reference Manual [online].
http://infocenter.arm.com/help/topic/com.arm.doc.ddi0388e
/DDI0388E_cortex_a9_r2p0_trm.pdf, 2009 [cit. 2015-1-28].

[16] ARM: Introducing NEON Development Article [online].
infocenter.arm.com/help/topic/com.arm.doc.dht0002a
/DHT0002A_introducing_neon.pdf, 2009 [cit. 2015-1-28].

[17] ARM: ARM Compiler toolchain [online].
infocenter.arm.com/help/topic/com.arm.doc.dui0489g
/DUI0489G_arm_assembler_reference.pdf, 2012 [cit. 2015-1-28].

[18] Gonzalez R. C., W. R.: *Digital Image Processing*. Prentice Hall, 2007, iSBN-10: 013168728X.

Příloha A

Obsah CD

Na CD se nachází zdrojové kódy programu, které se můžou zkompilovat za pomoci přiloženého souboru Makefile. Dále se na disku nachází soubor README s instrukcemi pro zkompilování.

Složky mají tento obsah:

- **image** – Obsahuje obrázky, které byly použity na testování.
- **output** – Do složky output se ukládají obrázky, na které byli aplikovány algoritmy.
- **text** – Obsahuje elektronickou verzi bakalářské práce, včetně zdrojových souborů.
- **bin** – Obsahuje zkompilované binární soubory.