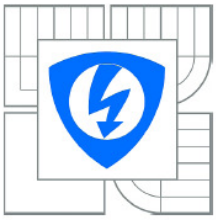
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

DETEKCE A POČÍTÁNÍ LED S VYUŽITÍM SSE INSTRUKCÍ

LED DETECTION USING SSE INSTRUCTION SET

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE — Bc. TOMÁŠ STACHERA
AUTHOR

VEDOUCÍ PRÁCE — Ing. PETER HONEC
SUPERVISOR

BRNO 2010

VYSOKÉ UČENÍ
TECHNICKÉ V BRNĚ
Fakulta elektrotechniky
a komunikačních technologií
Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student: Bc. Tomáš Stachera ***ID:*** 77696
Ročník: 2 ***Akademický rok:*** 2009/10

NÁZEV TÉMATU:

Detekce a počítání LED s využitím SSE instrukcí

POKYNY PRO VYPRACOVÁNÍ:

Navrhněte algoritmy pro detekci a počítání rozsvícených LED diod na ovládacím panelu výkonového zdroje. Algoritmy optimalizujte pro použití SSE instrukcí a porovnejte s C/C++ kompilátory. Algoritmy implementujte do GUI.

DOPORUČENÁ LITERATURA:

Dle vlastního literárního průzkumu a doporučení vedoucího práce.

Termín zadání: 8.2.2010 ***Termín odevzdání:*** 24.5.2010

Vedoucí práce: Ing. Peter Honec
Konzultanti diplomové práce:

prof. Ing. Pavel Jura, CSc.
předseda oborové rady

UPOZORNĚNÍ:

Autor diplomové práce nesmí při vytváření diplomové práce porušit autorská práva třetích osob, zejména nesmí zasahovat nedovoleným způsobem do cizích autorských práv osobnostních a musí si být plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení části druhé, hlavy VI. díl 4 Trestního zákoníku č. 40/2009 Sb.

Vysoké učení technické v Brně
Fakulta elektrotechniky a komunikačních technológii
Ústav automatizace a měříci techniky

Detekce a počítaní LED s využitím SSE instrukcí
Diplomová práce

Studijní obor : Kybernetika, automatizace a měření
Student : Bc.Tomáš Stachera
Vedouci projektu : Ing. Peter Honec

Abstrakt:

Témou tejto diplomovej práce je vytvoriť sadu inštrukcií pre detekovanie a počítanie LED na paneli výkonového zdroja s použitím algoritmov s SSE inštrukciami. Cieľom bude porovnať čas potrebný na spracovanie jednotlivých funkcií, ktoré nie su optimalizované s funkciami optimalizovanými pomocou SSE inštrukcií. V prvej časti práce sa budem venovať popisu SSE inštrukcií a popisu niektorých funkcií na spracovanie obrazu, ktoré budú využívať algoritmy s SSE inštrukciami. V druhej časti popíšem algoritmy na detekovanie a počítanie LED, ako aj možnosti ich optimalizácie pomocou SSE. Ďalej popíšem uživateľské rozhranie pre testovanie jednotlivých funkcií. Nakoniec zhodnotím dosiahnuté výsledky a porovnám jednotlivé merania.

Kľúčové slová: počítačové videnie, SSE inštrukcie, optimalizácia.

Brno University of Technology
The faculty of Electrical Engineering and Communication
Department of Control, Measurement and Instrumentation

Led detection using SSE instruction set

Master's thesis

Specialisation of study: Cybernetics, Control and Measurement
Student : Bc.Tomáš Stachera
Supervisor : Ing. Peter Honec

Abstract:

The themes of this work is to make an instruction set for detection and counting LED in power supply control panel, with using SSE instruction set algorithm. The goal will be to compare function time for optimized and not optimized functions. The first part focuses on SSE instructions introduction, as well as on introduction of some functions using SSE instructions. The second part is dealing with algorithm for detection and counting LEDs and possibility of those algorithms optimizing with SSE instruction set. Next part introduces user interface for developed functions testing. On the very end I compare results of all functions and evaluate these results.

Keywords: computer vision, SSE instruction set, to optimize.

Bibliografická citace

Stachera, Tomáš. *Detekce a počítaní LED s využitím SSE instrukcí*. Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2010. 58s.,1 příloha. Vedoucí práce: Ing. Peter Honec

Prohlášení

„Prohlašuji, že svou diplomovou práci na téma Detekce a počítaní LED s využitím SSE insttrukcí jsem vypracoval samostatně pod vedením vedoucího diplomové práce a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.“

V Brně dne : 10.5.2010 Podpis: 

Poděkování

Děkuji tímto Ing. Petru Honcovi za cenné připomínky a rady při vypracování diplomové práce.

V Brně dne : 10.5.2010 Podpis: 

OBSAH

SEZNAM OBRÁZKŮ

1. ÚVOD

V súčasnosti sa už bežne stretávame s digitalizovaným obrazom. Obraz sa skladá z množstva bodov, ktoré majú rôzny jas a spolu tak tvoria celistvý obraz. Počet týchto bodov, ktoré nazývame pixely závisí od rozlíšenia obrazu. Napríklad pre rozlíšenie obrazu 640 na 480 je počet pixelov obrazu 307200. Hodnoty pixelu bývajú väčšinou od 0 do 255, tým sa môže použiť na ich zapísanie do pamäte typ char. Vo veľkom množstve aplikácií je potrebné obrazové dáta spracovať do požadovanej podoby. Je potrebné na obraze meniť jas, kontrast, filtrovať šum, hľadať požadované objekty a podobne. Taktiež sa vyžaduje, aby sa dali obrazové dáta spracovávať aj pri zmenených podmienkach, napríklad osvetlenia. Aby sa jednotlivé operácie mohli uskutočňovať, sú nutné operácie s jednotlivými pixelmi obrazu. Keďže sa obraz skladá z množstva pixelov, vykoná sa množstvo operácií, ktoré samozrejme trvajú pomerne dlhý čas. Pri náročnejších operáciách s obrazom, ako je filtrácia maskou a podobne, kde sa s jednotlivými pixelmi pracuje viackrát, je veľmi dôležité tento čas čo najviac skrátiť. Jednou z možností ako tento čas skrátiť, je použitie SSE inštrukcií.

Množstvo priemyselných aplikácií využíva spracovanie obrazu pri rôznych procesoch, ktoré často nahrádzujú vizuálnu kontrolu človekom. Automatické spracovanie obrazu a jeho vyhodnotenie pomocou počítača má nesporné výhody oproti vizuálnej inšpekcii človekom. Ako už bolo spomenuté vyššie, algoritmy na spracovanie a vyhodnocovanie obrazu trvajú pomerne dlhý čas, ktorý sa môže javiť ako problém pri veľkosériovej výrobe. Skrátenie času týchto operácii prináša vyššiu efektivitu procesu a ušetrí náklady.

V tejto práci ukážem spôsob skrátenia času pomocou SSE inštrukcií v priemyselnej aplikácii, kde je potrebné automaticky kontrolovať počet svietiacich LED diód na LED paneli.

2. ZÁKLADNÝ POPIS SSE INŠTRUKCIÍ.

SSE inštrukčná sada pre x86 architektúru bola predstavená firmou Intel v roku 1999 na ich procesoroch série Pentium 3. SSE inšrukcie sú na princípe technológie SIMD (Single instruction,Multiple Data), čo voľne preložené znamená Jedna inštrukcia, viacnásobne dáta. Princíp SIMD spočíva v paralelnom vykonávaní jednej operácie s viacerými dátami. Jednou z prvých technológií predsatvených Intelom bola technológia MMX. Táto technológia priniesla 64-bitové registre, ktoré umožnili paralelne spracovávať 8 nezávislých dát o veľkosti 1 Byte, alebo 4 nezávislé dáta o veľkosti 2 Byty. Po technológii MMX nasledovala technológia SSE. Originálne SSE inštrukcie pridali 8 nových 128 bitových registrov XMM0 až XMM1. Ďalšie rozšírenia od AMD ako aj od Intelu pridali ďalších 8 registrov XMM8 až XMM15 [8]. Bol pridaný tiež nový 32 bitový control/status register MXCSR [8]. Hlavnou výhodou SSE inštrukcií je, že spracovávajú inštrukcie v balíkoch. Takže umožňujú spracovať naraz štyri 32 bitové hodnoty typu float, dve 64 bitové hodnoty typu double, štyri 32 bitové hodnoty typu integer, osem 16-bitových hodnôt typu short integer alebo 16 8-bitových hodnôt typu char [8]. Celočíselné inštrukcie majú inštrukcie pre znamienkové a neznamienkové premenné.

2.1 VÝVOJ SSE INŠTRUKCIÍ.

SSE inštrukcie sa postupne vylepšovali, pridávali sa nové. Ich vývoj bol nasledovný:

SSE2 Predstavené s Pentium 4. Tieto inštrukcie pridali nové matematické inštrukcie pre typ double a rozšírilo MMX inštrukcie na prácu s 128-bitovým XMM registrom [8].

SSE3 Pridali DSP orientované matematické inštrukcie a inštrukcie na manažment procesov [8].

SSSE3 Pridali nové inštrukcie pre prácu a pretypovanie premenných typu byte a word, násobenie 16 bitových čísel [8].

SSE4 Boli pridané ďalšie inštrukcie pre prácu s celočíselnými typmi.

SSE5 Nové inštrukcie ohlásené AMD

AVX je nová pokročilejšia verzia SSE ohlásená Intelom na rok 2010, rozširuje rozsah dát z 128 bitov na 256 bitov a inštrukcie s 3 operandami [8].

2.2 PRAKTICKÉ POUŽITIE SSE INŠTRUKCIÍ

SSE inštrukcie sú inštrukcie, ktoré sa programujú v assembleri. Preto je nutné, aby časť programu pre volanie SSE inštrukcií bola napísaná v assembleri. Výhoda SSE inštrukcií je v tom, že spracovávaju paralelne viacero hodnôt, a tým šetria čas pri práci s veľkými poliami a rozsiahlymi cyklami. Ako príklad uvediem sčítanie 2 polí reálnych čísel. Každé pole obsahuje 40 hodnot typu float. Program pre sčítanie 2 polí by v C vyzeral následovne:

```
float pole1[40];
float pole2[40];
float vysledok[40];

for(int I =0;I<40;I++)  vysledok[I]=pole[1] + pole[2];
```

Obrázok 1. Príklad sčítanie 40 hodnôt typu Float v C.

Pre súčet 40 hodnôt je potrebných 40 cyklov. Pri práci s SSE inštrukciami využijem to, že tieto inštrukcie dokážu spracovávať paralelne 4 hodnoty typu float. Teda na tento súčet je potrebných len 10 cyklov. Potom by program pre sčítanie 2 float čísiel mohol vyzerať ako na obrázku 2.

Kód s použitím SSE inštrukcií vyzerá zložitejšie ako kód v C, ale umožní nám použiť 4 násobne menší počet cyklov. Toto sa výrazne prejaví hlavne u veľkého množstva cyklov, ktoré sa pri spracovávaní obrazu používa. SSE

inštrukcie sú obsiahnuté vo funkcii asm. Sú použité 2 SSE inštrukcie movaps a addps. Inštrukcia movaps je vlastne modifikáciou inštrukcie move. Movaps presúva 128 bitov z jednej adresy pamäti do druhej. V tomto prípade funkcia načíta adresu pamätí poľa 1 a uloží ju do registra xmm0. Keďže je tento register 128 bitový, táto inštrukcia načíta naraz 4 float hodnoty z poľa 1. Rovnako inštrukcia presunie aj 4 hodnoty float poľa 2 do registra xmm1. Druhou SSE inštrukciou je inštrukcia addps.

```
 for (int i=0; i<10;i++)
{
  _asm_(
          movaps  xmm0, adresa_pole1
         movaps  xmm1, adresa_pole2
          addps  xmm0, xmm1
         movaps adresa_vysledok ,xmm0
)
adresa_pole1=adresa_pole1+16;
adresa_pole2=adresa_pole2+16;
adresa_vysledok=adresa_vysledok+16;
}
```

Obrázok 2. Príklad sčítania 40 hodnôt pomocou SSE inštrukcií.

Táto funkcia je definovaná pre paralelné sčítanie 4 float hodnôt v registroch xmm. V našom prípade sčíta registre xmm1 a xmm0 a výsledok posunie do registru xmm0.

Ďalšia inštrukcia movaps presunie obsah registru xmm0, teda výsledok, do poľa vysledok. Po funkcii asm nasleduje posunutie adresy polí o 16 bytov, keďže float hodnota má 4 byty a paralelne sa spracuvávajú 4 hodnoty.

3. VÝVOJOVÉ PROSTRIEDKY PRE VYTVORENIE TESTOVACEJ APLIKÁCIE

Pre vytvorenie testovacej aplikácie, pri ktorej overím možnosti spracovania obrazu je potrebné vybrať si vývojové prostredie, pod ktorým sa bude dať vytvoriť jednak kód v c++, a jednak bude možné používať inštrukcie assembleru. Pri tomto projekte som sa rozhodol využívať nekomerčné vývojové prostredia. Moja voľba padla na Open source prekladač gcc, ktorý umožňuje písať programy v C a aj v C++. Ďalej je možné v ňom implementovať funkcie, do ktorých sa budú dať písať inštrukcie assembleru. Tento prekladač je možné stiahnuť na http://gcc.gnu.org/. Pre uľahčenie vývoja aplikácie som použil freevarové IDE Code Block, ktoré dokáže implementovať prekladač gcc. Toto IDE je možné stiahnuť na http://www.codeblocks.org/. Aj prekladač, aj IDE sú multiplatformné, takže sa dajú použiť pre viaceré OS (Windows,Linux,Mac..)

3.1 POUŽITIE INŠTRUKCIÍ ASSEMBLERU V PREKLADAČI GCC

Pre vývoj testovacej aplikácie som použil prekladač gcc. Tento prekladač umožňuje vkladanie assemblerovských inštrukcií do programov písaných v C/C++. Je možné používať funkciu asm(), kde sa dajú do riadkov písať jedntlivé inštrukcie assembleru. Je možné používať aj zápis __asm__ (), obidva zápisy sú rovnocenné. Ten druhý sa používa hlavne preto, aby nedošlo ku konfliktu názvov v programe. Ďalšou možnosťou je pridanie do deklarácie volanie __volatile__. Toto volanie znamená, že kód sa vykoná presne v mieste a v poradí, v akom je napísaný. Nepodlieha optimalizácii pri preklade. Celá funkcia pre vykonanie assemlerovských inštrukcí tak bude vyzerať __asm__ __volatile__(). Prekladač načítava jednotlivé assemblerovské inštrukcie ako reťazec, preto sa jednotlivé inštrukcie zapisujú ako textové reťazce s oddeľovacími znakmi \t\n. Celý kód vysvetlím na obrázku 3.

```
__asm__ __volatile__ (
                        “movups    (%0), %%xmm0\t\n”
                        “movups     (%1), %%xmm1\t\n”
                         “addps      %%xmm1, %%xmm0 \t\n”
                         “movups     %%xmm0 , (%2) ”
    /*vystupy*/       : “+r” (pole1) , “+r” (pole2), “+r” (vysledok)
    /*vstupy*/        :
   /*registri*/       : “xmm0”,”xmm1”)
```

Obrázok 3. Príklad funkcie __asm__ __volatile__ ()

Na obrázku 3 je asm funkcia pre spočítavanie dvoch polí reálnych čísel. Jej funkcia je rovnaká ako na obrázku 2. Prvý riadok je funkcia movups, ktorá načíta hodnotu nultej premennej funkcie do registru xmm0. Pri vykonávaní inštrukcií tejto funkcie je výsledok operácie uložený do druhého prvku, zatiaľ čo, keby sme písali inštrukcie v „klasickom“ prekladači sa výsledok operácie ukladá do prvého prvku. Na druhom riadku je rovnako funkcia movups, ktorá načíta hodnotu prvej premennej do registru xmm1. Následuje inštrukcia addps sčítajúca prvky v registroch xmm1 a xmm0, pričom výsledok uloží do registru xmm0. Následuje posledná inštrukcia movups, ktorá uloží obsah registru xmm0, teda výsledok do druhej premennej. Za touto inštrukciou už nenasleduje žiadna iná, nedávame nakoniec ukončovacie znaky \t\n.

Za inštrukciami nasleduje deklarácia výstupných premenných. Pokiaľ chceme vstupné premenné predávať ako adresy, napíšeme ich taktiež do výstupných premenných. Písmeno r v úvodzovkách znamená, že sa jedná o premennú alebo konštantu, plus znamená, že sa jedná o adresu. Podľa zápisu zľava sa ráta nultý prvok, prvý prvok atď. Ďalší riadok sú vstupné premenné, v našom príklade ich nepoužívame. Pokiaľ by boli použité, tak by sa poradie prvkov rátalo ako pokračovanie po výstupných premenných.

Teda, keby v mojom príklade bola definovaná jedna vstupná premenná, bol by to pre funkciu tretí prvok. Posledný riadok funkcie je deklarácia všetkých registrov, ktoré používame.

3.2 KNIŽNICE NA SPRACOVANIE OBRAZOVÝCH DÁT

Na to, aby sa dalo pracovať s jednotlivými pixelmi a ostatnými dátami z obrazu, je potrebné tieto dáta získať z obrazových súborov. Taktiež je dobré, keď sa výsledok operácií s obrazmi dá zobraziť vo výslednom obrazovom súbore. Jednou z najlepších možností sa mi zdalo použiť knižnice obrazových funkcií OpenCV. Takéto knižnice sú freewarové a podľa licencie sa môžu bezplatne používať na komerčné aj nekomerčné aplikácie jako aj na množstvá operácií s obrazom. OpenCV sa dá stiahnuť na [4].

3.3 MERANIE ČASU JEDNOTLIVÝCH OPERÁCIÍ

Keďže budem porovnávať rýchlosť spracovania obrazu pri použití klasických funkcií a SSE inštrukcií, je potrebné merať čas vykonávania jednotlivých operácií. Na toto meranie som použil hlavičkový súbor rdtsc.h, ktorý som stiahol z [6]. Tento súbor obsahuje funkciu rdtsc() merajúcu počet cyklov CPU. Potom stačí pred funkciu, ktorá vykonáva obrazové operácie načítať hodnotu rdtsc() do nejakej premennej, po ukončení funkcie znovu načítať hodnotu rdtsc() a napokon obidve hodnoty odčítať. Výsledná hodnota je počet cyklov CPU za dobu vykonávania funkcie s obrazovými operáciami. Pokiaľ chceme vedieť čas vykonávania tejto inštrukcie, napríklad v mikrosekundách, môžeme buď zistiť hodnotu frekvencie CPU, alebo použiť funkciu Sleep() a načítať hodnoty rdtsc() pred a za funkciou, pričom podľa známeho času čakania funkcie Sleep() určíme hodnotu, ktorou musíme vydeliť počet cyklov CPU, aby sme dostali čas v mikrosekundách.

4. PRÍKLADY FUNKCIÍ NA SPRACOVANIE OBRAZOVÝCH DÁT

Poznatky z práce z SSE inštrukciami som využil pri vytvorení dll knižnice. V tejto knižnici sú tri funkcie na spracovanie obrazu s použitím SSE inštrukcií a tri rovnaké funkcie bez SSE inštrukcií, aby sa dal porovnávať výsledok operácií s obrazom. Funkcie na spracovanie obrazu sú: Sčítanie dvoch obrazov, Inverzia obrazu a detekcia hrán na obraze s použitím masky so Sobelovým operátorom. Algoritmy pre programovanie týchto funkcií som čerpal z [1],[2],[5], [7].

4.1 SÚČET OBRAZOV

Súčet dvoch obrazov som naprogramoval tak, že som jednotlivé pixely obrazu sčítal a potom vydelil dvomi. Súčet jednotlivých pixelov som počítal v cykle for. Cyklus som opakoval pre každý pixel. Funkcia pre súčet obrazov bez použitia SSE inštrukcií má tvar: SucetObrazov(unsigned char* f1,unsigned char* f2,unsigned char*f3,long int n). Prvý parameter je pointer na data obrazu 1. Druhý parameter funkcie je pointer na data obrazu 2. Tretí parameter funkcie je pointer na data výsledného obrazu. Všetky tri parametre musia byť typu unsigned char. Štvrtý parameter je počet pixelov výsledného obrazu. Funcia na sčítanie obrazu s použitím SSE inštrukcií má tvar: SSE_SucetObrazov(unsigned char* f1,unsigned char* f2,unsigned char *f3,long int n). Parametre má rovnaké ako funkcia na sčítanie obrazu bez SSE inštrukcií. Keďže použité SSE inštrukcie môžu paralelne spracovávať až 8 pixelov počet cyklov je 8krát menší. Keďže celkový počet pixelov nemusí byť deliteľný ôsmimi, je potrebné na koniec funkcie vykonať súčet obrazov „klasicky“ pre počet pixelov, ktorý sa rovná zvyšku po delení celkového počtu pixelov ôsmimi. Asemblerovská funkcia pre súčet obrazov je na obrázku 4.

```
__asm__ __volatile__(
    "xorps      %%xmm0, %%xmm0\n\t"
    "movups     (%1), %%xmm0\n\t"
    "movups     (%2), %%xmm1\n\t"
    "add        $8,%1\n\t"
    "add        $8,%2\n\t"
    "pavgb      %%xmm1, %%xmm0\n\t"
    "movups     %%xmm0, (%0)\n\t"
    "add        $8, %0"

    : /* outputs */ "+r" (f3),"+r" ( f1 ), "+r" ( f2 )
    : /* inputs */
    : /* clobbered */ "xmm0", "xmm1");
```

Obrázok 4. Assemblerovská funkcia pre súčet obrazov

Prvý riadok funkcie je inštrukcia xorps, ktorá je definovaná ako exlusive-OR. Táto funkcia urobí exclusive-OR medzi registrom xmm0 a xmm0. Výsledkom tejto operácie je, že sa register xmm0 vymaže. Ďaľšiu inštrukciou je inštrukcia movups, ktorá načíta hodnotu prvého prvku, teda f1 do registra xmm0. Iná inštrukcia movups načíta hodnotu druhého prvku, teda f2 do registru xmm1. Nasleduje inštrukcia add, ktorá nie je SSE inštrukcia, pričom posunie adresu prvého prvku o 8 bytov. Tým sa zabezpečí, že pri ďalšom cykle sa bude načítavať pole f1 posunuté o bytov. Nasledovná inštrukcia add posunie o 8 bytov adresu 2. prvku, teda f2. Ďalšou inštrukciou je inštrukcia pavgb popísaná v [9]. Táto funkcia umožňuje pracovať v 64 bitovom alebo v 128 bitovom režime. Ja som použil 64 bitový režim. Jej funkčnosť je urobiť priemer dvoch registrov. Teda vlastne sčíta hodnoty registra xmm1 a xmm0 a vydelí ich dvomi. To je presne tá operácia, ktorú potrebujeme pri súčte dvoch obrazov. V 64 bitovom mode spracováva paralelne 8 hodnôt typu byte. Nasleduje inštrukcia movups, ktorá zapíše výsledok predchadzajúcej operácie z registru xmm0 do nultého prvku, teda f3, čo je pole výsledného obrazu.

4.2 INVERZIA OBRAZU

Inverzia obrazu typu unsigned char je definovaná vzťahom 1.1 [7].

$$g(x,y)=255-f(x,y) \qquad (1.1)$$

Kde g(x,y) je výstupný obraz a f(x,y) je vstupný obraz. Inverziu obrazu vykonáva funkcia InverziaObrazu(unsigned char* f1,unsigned char*f3,long int n). Táto funkcia má prvý parameter pointer na pole vstupného obrazu. Druhý parameter je pointer na pole výstupného obrazu. Tretím parametrom je počet pixelov obrazu. Funkcia v cykle for odpočítava od hodnoty 255 postupne všetky pixely obrazu a výsledok ukladá do poľa výstupného obrazu. Počet cyklov for je rovný počtu pixelov obrazu.

Pre inverziu obrazu s použitím SSE inštrukcií je použitá funkcia SSE_InverziaObrazu(unsigned char* f1,unsigned char*f3,long int n). Parametre funkcie sú rovnaké ako v predchádzajúcej funkcií. Táto funkcia obsahuje asseblerovskú funkciu, ktorá bude paralelne odčítavať 16 hodnôt. Preto bude počet cyklov for 16krát menší ako je počet pixelov. Keďže celkový počet pixelov nemusí byť deliteľný šestnástimi, je na koniec funkcie treba zaradiť cyklus, ktorý klasickým spôsobom vypočíta počet pixelov zvyšku po celočíselnom delení šestnástimi.Keďže budeme odčítavať od šestnástich hodnôt 255 hodnoty vstupného obrazu, musíme si na začiatku funkcie definovať pole šestnástich hodnôt 255, ktoré potom predáme ako parameter asm funkcii. Asm funkcia pre inverziu obrazu je na obrázku 5. Prvou inštrukciou funkcie je funkcia xorps, ktorá vynuluje register xmm0. Druhá inštrukcia movups načíta pole hodnôt 255 (odcitan) do registru xmm0. Ďaľšia inštrukcia movups načíta do registru xmm1 pole vstupného obrazu (f1). Nasledovná inštrukcia add posunie adresu poľa vstpného obrazu o 16 bytov.

Nasleduje inštrukcia psubusb, ktorá je popísaná v [9]. Táto funkcia odčítava paralelne 16 hodnôt typu byte. V našej funkcii odčíta register xmm1 od registra xmm0 a výsledok dá do registra xmm0. Nasleduje inštrukcia movups, ktorá presunie obsah registru xmm0 do nultého prvku, teda poľa výstupného obrazu (f3).

Posledná inštrukcia add posunie adresu výstupneho poľa o 16 bytov na zápis v ďalšom cykle.

```
__asm__ __volatile__(
        "xorps      %%xmm0, %%xmm0\n\t"
        "movups     (%2), %%xmm0\n\t"
        "movups     (%1), %%xmm1\n\t"
        "add        $16,%1\n\t"
        "psubusb     %%xmm1, %%xmm0\n\t"
        "movups     %%xmm0, (%0)\n\t
        "add        $16, %0

        : /* outputs */ "+r" (f3),"+r" ( f1 )
        : /* inputs */  "r" (odcitan)
        : /* clobbered */ "xmm0", "xmm1");
```

Obrázok 5. Asm funkcia pre inverziu obrazu.

4.3 DETEKCIA HRAN V OBRAZE KONVOLUČNOU MASKOU SO SOBELOVYM OPERÁTOROM.

Funkcia na detekciu hran konvolučnou maskou sa používa na nájdenie hrán v obraze. Konvolúcia je danná vzťahom 1.2 [7].

$$g(x,y) = \sum_{s=-s_{max}}^{s_{max}} \sum_{t=-t_{max}}^{t_{max}} h(s,t).f(x-s,y-s) \qquad (1.2)$$

Vo vzorci 1.2 je g(x,y) pixel výstupneho obrazu, f(x,y), h(s,t) je matica konvolučnej masky. Výsledný pixel obrazu je tak súčtom súčinov pixelov okolitých bodov s pixelmi masky. Pre Sobelov operátor sa použijú dve masky 3x3 Gx a Gy. Tieto masky sú zobrazené na obrázku 6.

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

Gx

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

Gy

Obrázok 6. Konvolučné masky so Sobelovym operátorom [7].

Výpočet pixelu prebieha tak, že najskôr sa vykoná konvolúcia s maskou Gx a výsledok sa bude GX. Potom sa vykoná konvolúcia s maskou Gy a výsledok bude GY. Výsledný pixel G sa vypočíta podľa vzťahu 1.3

$$G = \sqrt{Gx^2 + Gy^2} \qquad (1.3)$$

Vzťah 1.3 môžeme približne aproximovať vzťahom 1.4. Táto aproximácia odstraňuje umocňovanie a odmocňovanie a nahradzuje ho absolutnými hodnotami.

$$G = |Gx| + |Gy| \qquad (1.4)$$

Funkcia pre detekciu hran je SobelMask(unsigned char*f1,unsigned char* f3, unsigned int sirka,unsigned int dlzka). Algoritmus pre túto funkciu som čerpal z [5]. Táto funkcia má prvý parameter pointer na vstupný obraz, druhý parameter je pointer na výstupný obraz. Tretím parametrom je šírka obrazu v pixeloch a štvrtý parameter je dĺžka obrazu v pixelox. Funkcia má deklarované masky GX a GY ako dvojrozmerné pole. Nasledujú dva cykly for, prvý prechádza každý pixel po dĺžke a druhý prechádza každý pixel po šírke. Celkovo obidva cykly prejdú každý pixel

v obraze. Keďže sa používa maska 3x3, krajné pixely sa nedajú maskovať pomocou masky, tak sú nahradené hodnotou 0. Nasledujú dva cykly for, ktoré vypočítajú hodnotu Gx (sumX) a hodnotu Gy (sumY) podľa vzťahu 1.2. Následne sa vypočíta súčet absolutných hodnôt Gx a Gy. Potom sa zabezpečí podmienkami, aby sa hodnoty väčšie ako 255 zapísali ako 255. Nakoniec sa výsledok pretypuje na unsigned char a zapíše do poľa výsledku. Z tejto funkcie vidno, že je potrebné na každý pixel 9 cyklov for. Paralelne spracovanie hodnôt, ktoré poskytujú SSE inštrukcie by tak výpočet mohlo veľmi urýchliť.

Funkcia pre detekciu hran v obraze s použitím SSE inštrukcií je SSE_SobelMask(unsigned char*f1,unsigned char* f2,unsigned int sirka,unsigned int dlzka). Táto funkcia má rovnaké parametre ako predchádzajúca funkcia.
V tejto funkcii bude možné spracovávať paralelne 8 hodnôt, keďže sa používa násobenie, pri ktorom je potrebné hodnoty pretypovať na typ word, ktorý je 16 bitový. Na začiatku som si nadefinoval pole masiek so Sobelovým operátorom.

Kvôli jednoduchosti výpočtu s SSE inštrukciami som použil jednorozmerné pole. Každá hodnota je v poli 8krát, takže obidve polia masiek majú 72 prvkov.Nasledujú dva cykly for. Prvý prechádza obraz po dĺžke od druhého pixelu po predposledný. Prvý a posledný pixel nepočítam z dôvodu toho, že masky by pri tomto pixeli zasahovali mimo obraz. Druhý cyklus for prechádza pixely po šírke od prvého po posledný s tým rozdielom, že sa v ňom bude paralelne spracovávať 8 pixelov, takže počet cyklov je 8krát menší ako je šírka obrazu –2. Keďže šírka obrazu –2 nemusí byť deliteľná ôsmimi, je po tomto cykle for ďalší cyklus for o počte cyklov rovnajúcemu sa zvyšku po celočíselnom delení ôsmimi. V druhom cykle for je na začiatku nulovanie pointerov na pomocné premenné p_sx a p_sy. To sú polia, kde sa budú ukladať vypočítane hodnoty po konovlúcii masky GX a GY. Nulovanie je robené asm funkciou a to tak, že sa najskôr vynuluje register xmm0 a potom sa zapíše do obidvoch premenných. Nasledujú dva cykly for, ktoré prejdú všetky hodnoty masky s určenými pixelmi obrazu. Na začiatku cyklu sa nastaví adresa pixelu vstupného poľa, ktorá sa bude násobiť s maskou GX a GY. Násobiť sa bude paralelne 8 pixelov od nastavenej adresy. Prvých 5 inštrukcií vo

funkcií asm bude xorps, čo znamená že sa vynuluje všetkých 5 registrov, ktoré budeme používať (xmm0 až xmm5).

Nasleduje 5 inštrukcií movups, kde prvá načíta hodnotu masky GXP do registra xmm3, druhá načíta adresu vstupného poľa do registru xmm1. Tretia načíta adresu premennej medzsúčtu p_sx do registru xmm0. Štvrtá načíta adresu premennej medzisúčtu p_sy do registru xmm4 a nakoniec posledná načíta hodnotu masky GYP do registru xmm5. Keďže na násobenie s SSE inštrukciami je iba pre typ word, to znamená že je potrebné urobiť pretypovanie hodnôt poľa vstupného obrazu ktoré sú typu byte. Táto operácia je možná pomocou SSE inštrukcie punpcklbw. Táto inštrukcia je popísaná v [9]. Táto inštrukcia spája dve hodnoty byte do jednej hodnoty word. Túto operáciu vykonáva paralelne s ôsmimi dvojicami. Jeden byte použijem zo vstupného poľa a druhý pripočítam ako nulový byte.
Na pripočítanie nulového bytu použijem register xmm2, ktorý je vynulovaný. Výsledna inštrukcia tak načíta 1 byte z registru xmm2, ktorý je nulový a potom načíta druhý byte z registru xmm1, čo je hodnota vstupného poľa. Výsledný typ byte uloží do registru xmm1. Potom nasleduje násobenie tohto registru maskou pomocou inštrukcie pmullw. Táto inštrukcia je popísaná v [9]. Je to násobenie 2 registrov dolné slovo. Keďže násobím hodnoty maximálne 255 s dvomi, nie je potrebné násobiť aj horné slovo Prvá funkcia pmullw vynásobí vstupné pole s maskou GX a druhá vynásobí vstupné pole s maskou GY.

Nasleduje inštrukcia paddw popísaná v [9]. Táto inštrukcia spočíta súčet ôsmich dvojíc hodnôt typu byte. Prvá spočíta pomocný súčet premennej p_sx s registrom výsledku súčinu masky GX a pixelu vstupného obrazu. Druhá spočíta pomocný súčet premennej p_sy s registrom výsledku súčinu masky GY a pixelu obrazu. Nasledujú dev inštrukcie movups, ktoré zapíšu výsledky predchádzajúcich operácií do premenných p_sx a p_sy.

Za touto asm funkciou nasleduje výpočet absolutných hodnôt pomocných súčtov p_sx a p_sy. Tento výpočet sa vykonáva pomocou funkcie C abs(). Dôvod, prečo nie je použitý výpočet absolútnej hodnoty SSE inštrukciou je ten, že som nedokázal vyriešiť problémy, ktoré mi spôsoboval výpočet absolutnej hodnoty s SSE inštrukciou. Po tomto výpočte nasleduje posledná asm funkcia, ktorá je na obrázku 8.

```
                __asm__ __volatile__(
    "xorps      %%xmm0, %%xmm0\n\t"
    "xorps      %%xmm1, %%xmm1\n\t"
    "xorps      %%xmm2, %%xmm2\n\t"
    "xorps      %%xmm3, %%xmm3\n\t"
    "xorps      %%xmm4, %%xmm4\n\t"
    "xorps      %%xmm5, %%xmm5\n\t"
    "movups     (%1), %%xmm3\n\t"
    "movups     (%0), %%xmm1\n\t"
    "movups     (%2), %%xmm0\n\t"
    "movups     (%3), %%xmm4\n\t"
    "movups     (%4), %%xmm5\n\t"
    "punpcklbw   %%xmm2, %%xmm1\n\t"

     "pmullw     %%xmm1, %%xmm3\n\t"
     "pmullw     %%xmm1, %%xmm5\n\t"
     "paddw      %%xmm3, %%xmm0\n\t"
     "paddw      %%xmm5, %%xmm4\n\t"
     "movups     %%xmm4, (%3)\n\t"
     "movups     %%xmm0, (%2)"

  : /* outputs */ "+r" ( f3 ),"+r" (GXP),"+r" (p_sx),"+r" (p_sy),"+r" (GYP)
 : /* inputs */
   : /* clobbered */ "xmm0", "xmm1","xmm2","xmm3","xmm4","xmm5");
```

Obrázok 7. Výpočet hodnôt konvolúcie s maskami GX a GY

Táto funkcia má znovu prvé štyri inštrukcie xorps a tieto vynulujú všetky registre, s ktorými v tejto funkcii pracujem. Po nej nasledujú tri inštrukcie movups. Prvá načíta hodnotu p_sx do registru xmm1, druhá načíta hodnotu p_sy do registru xmm2. Tretia načíta pole hodnôt 255 do registru xmm4. Ďalej nasledujú dve inštrukcie paddw, ktoré spočítajú súčet hodnôt premnných p_sx a p_sy. Potom je potrebné výsledok tohto súčtu pretypovať na typ byte. To je možné pomocou SSE inštrukcie packuswb popísanej v [9].

Táto inštrukcia z premenných typu word vytvorí premennú typu byte saturáciou. Prvý parameter tejto inštrukcie je nulové pole xmm3, ktoré sa dá na pozíciu horných 8 bytov, pričom ich nebudem používať. Druhým parametrom je výsledný pixel, ktorý sa pretypuje na byte. Pokiaľ by jeho hodnota bola väčšia ako 255, je zapísaný ako 255. Keďže počítam súčet absolutných hodnôt tento súčet

nemôže byť záporný, takže tento prípad netreba ošetrovať. Nasleduje inštrukcia psubusb, ktorá odočíta výsledok od hodnoty 255, čím sa tak vytvorí inverzia a čiary budú na obraze zobrazené ako tmavé. Posednou inštrukciou je inštrukcia movups, ktorá uloží výsledný pixel do poľa výstupného obrazu.

```
__asm__ __volatile__(
        "xorps      %%xmm0, %%xmm0\n\t"
        "xorps      %%xmm1, %%xmm1\n\t"
        "xorps      %%xmm2, %%xmm2\n\t"
        "xorps      %%xmm3, %%xmm3\n\t"
        "movups     (%1), %%xmm1\n\t"
        "movups     (%2), %%xmm2\n\t"
        "movups     (%3), %%xmm4\n\t"
        "paddw      %%xmm1, %%xmm0\n\t"
        "paddw      %%xmm2, %%xmm0\n\t"
        "packuswb    %%xmm3, %%xmm0\n\t"
       "psubusb     %%xmm0, %%xmm4\n\t"
        "movups      %%xmm4, (%0)"

    : /* outputs */ "+r" ( f4 ),"+r" (p_sx),"+r" (p_sy)
   : /* inputs */    "r" (odcitanec)
    : /* clobbered */ "xmm0", "xmm1","xmm2","xmm3","xmm4");
```

Obrázok 8. Asm funkcia na výpočet pixelov výstupného obrazu.

4.4 PRAHOVANIE OBRAZU.

Prahovanie obrazu je jednou zo základných funkcií využívaných pri spracovaní obrazu. Prahovanie obrazu vytvorí zo vstupného obrazu binárny obraz, podľa určenej hodnoty prahu. Pokiaľ je hodnota pixelu vstupného obrazu menšia ako hodnota prahu pixel vo výstupnom obraze má hodnotu 0. Ak je hodnota pixelu vstupného obrazu väčšia, prípadne rovná hodnote prahu, pixel výstupného obrazu má hodnotu 255. Vhodnou voľbou veľkosti prahu môžeme oddeliť od pozadia objekty,

ktoré chceme detekovať. Prahovať môžeme jedným prahom, viacerými prahmi, prípadne adaptívnym prahom. Funkcia, ktorú vytvorím bude mať jeden prah, ktorého hodnotu budeme definovať ako parameter funkcie. Funkcia je v knižnici Led_pan.dll a má hlavičku: int DLL_EXPORT Prahovanie(unsigned char* f1,unsigned char*f3,unsigned int dx,unsigned int dy,unsigned char prah). Prvým parametrom je pointer na obrazové dáta vstupného obrazu, druhým pointer na obrazové dáta výstupného obrazu, tretím je šírka obrazu, štvrtým je výška obrazu a piatym je hodnota prahu. Hodnota prahu môže mať rozsah 0 až 255. Funkcia na prahovanie optimalizovaná pomocou SSE inštrukcií má rovnaké parametre ako vyššie uvedená funkcia a jej názov je SSE_Prahovanie.

```
__asm__ __volatile__(
        "xorps      %%xmm0, %%xmm0\n\t"
        "xorps      %%xmm1, %%xmm1\n\t"
         "xorps      %%xmm2, %%xmm2\n\t"
         "xorps      %%xmm3, %%xmm3\n\t"
         "xorps      %%xmm4, %%xmm4\n\t"
         "movups     (%2), %%xmm0\n\t"
         "movups     (%1), %%xmm1\n\t"
         "movups     (%3), %%xmm4\n\t"
         "punpcklbw  %%xmm2, %%xmm1\n\t"
         "pcmpgtw     %%xmm0, %%xmm1\n\t"
         "psubusw     %%xmm4, %%xmm1\n\t"
         "packuswb    %%xmm3, %%xmm1\n\t"
         "movups     %%xmm1, (%0)\n\t"
         "add        $8,%1\n\t"
         "add        $8, %0"
         : /* outputs */ "+r" (f3),"+r" (f1)
         : /* inputs */  "r" (prh),"r" (odc)
          : /* clobbered */ "xmm0", "xmm1","xmm2","xmm3","xmm4")
```

Obrázok 9. Asm funkcia na výpočet prahu pomocou inštrukcií SSE.

Optimalizovaná funkcia na prahovanie pomocou SSE inštrukcií je na obrázku 10. Táto funkcia dokáže paralelne prahovať 8 pixelov. Parametre asm funkcie sú: *f3*

je pointer na výstupné obrazové dáta, *f1* je pointer na vstupné obrazové dáta, *prh* je pole typu short int o ôsmich prvkoch, kde každý prvok je rovnaký a je rovný hodnote prahu, a *odc* je pole typu short int o ôsmich prvkoch, kde každý prvok má hodnotu 0xFF00. Pole *odc* je potrebné pri pretypovaní a porovnávaní.

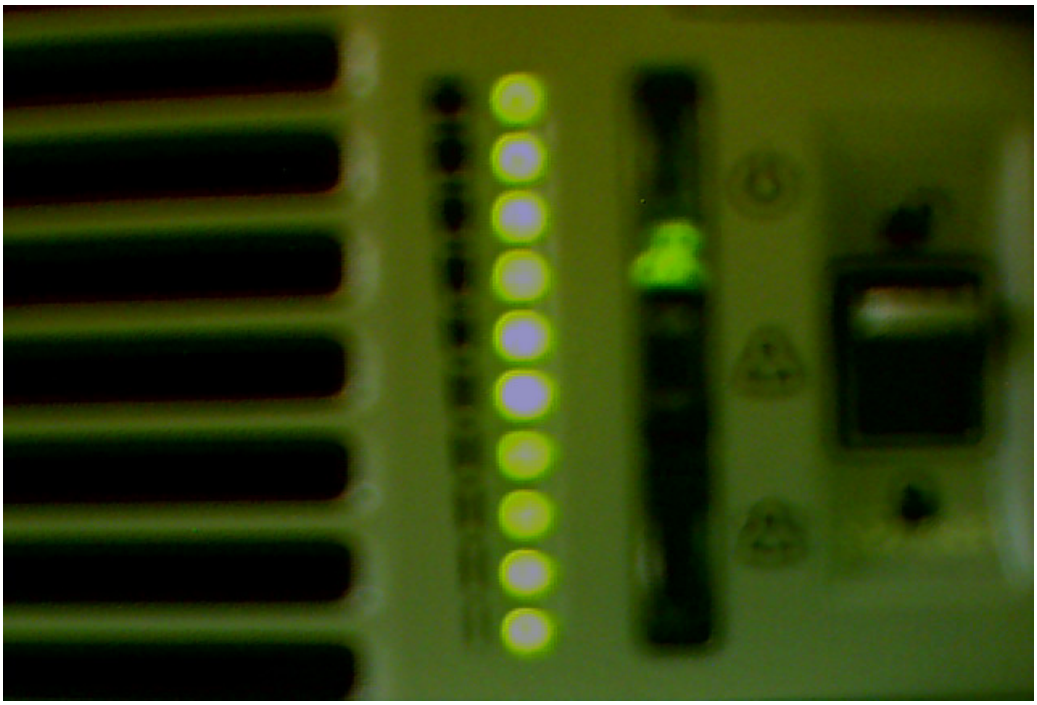
Funkcia používa 5 xmm registrov. V prvých piatich riadkoch kódu je funkcia xorps, táto vynuluje všetky xmm registre. Potom nasledujú 4 funkcie movups, ktoré postupne načítajú hodnotu prahu, obrazové data vstupného obrazu, a pomocný parameter *odc*. Nasleduje funkcia *punpcklbw,* ktorá pretypuje vstupné obrazové dáta na typ short int. Nasleduje funkcia *pcmpgtw* porovnávajúca hodnotu prahu s hodnotou pixelov vstupného poľa. Pokiaľ je hodnota pixelov vstupného poľa väčšia ako hodnota prahu, funkcia nastaví do výstupného registra hodnotu *0xFFFF*. Ak je menšia, nastaví do výstupného registra hodnotu *0x0000*. Táto funkcia je aj dôvod prečo musíme pretypovávať na 16-bitové hodnoty. Existuje aj SSE funkcia na porovnávanie 8-bitových hodnôt, vie však porovnávať iba znamienkové hodnoty, takže by nedokázala porovnať hodnotu väčšiu ako 127. Ďaľšou funkciou je funkcia *psubusw*, ktorá odpočíta od výsledného registru hodnotu *0xFF00*. Následuje funkcia *packuswb*, táto výstupné dáta pretypuje na 8-bitovú hodnotu. Funkcia *movups* presunie výsledok z registru do poľa výstupného obrazu. Posledné dve funkcie *add* posunú ukazatele na polia vstupného a výstupného obrazu o 8 pixelov, aby boli pripravené na ďalší cyklus.

5. POČÍTANIE LED NA LED PANELI .

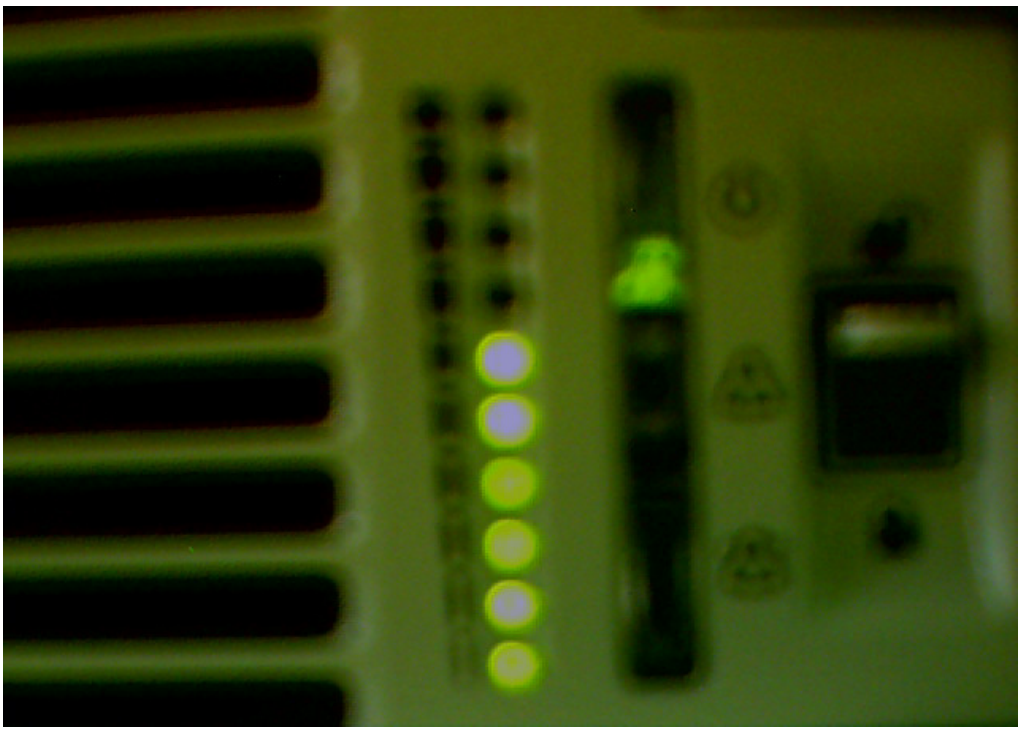
V tejto kapitole sa budem venovať praktickej aplikácii, kde budú použité funkcie na spracovanie obrazu a ich optimalizácia pomocou SSE inštrukcií. Cieľom je ukázať praktické využitie SSE inštrukcií a ich prínos pre prax.

5.1 POŽIADAVKY NA DETEKCIU A KONTROLU LED.

Na začiatku je potrebné formulovať požiadavky, ktoré musí aplikácia spĺňať na správnu detekciu a počítanie LED na LED paneli. Vytvorené funkcie v dll súboroch by mali byť použiteľné v automatickom testovacom systéme, ktorý výrobok s LED panelom kontroluje. Samotný výrobok je AC/DC výkonový zdroj, kde odoberaný výkon je signalizovaný na LED paneli. V závislosti od dodávaného výkonu sa rozsvecujú jednotlivé LED a ich počet indikuje dodávaný výkon. Pri automatickom teste sa testuje počet rozsvietených LED pri rôznych režimoch výkonu. V pôvodnom riešení sa počet rozsvietených LED kontroluje opticky operátorom. Pre plne automatizovaný proces sa javí ako najlepšie riešenie kontrolovanie LED panelu pomocou kamery a automatické vyhodnotenie pomocou počítača. Uvedený proces je potrebné optimalizovať na čo možno najväčšiu rýchlosť a dosiahnuť 100% spoľahlivosť pri detekovaní. Snímky sa budú získavať pomocou USB kamery . Na obrázku 11 je snímka LED panelu pri plnom výkone a na obrázku 12 je snímka LED panelu pri 60% výkone. LED panel, ktorý je potrebné kontrolovať je stredný stĺpec LED diód a tento sa mení v závislosti od dodávaného výkonu. Ako vidno pri plnom výkone svieti maximálny počet LED diód, teda 10. Pri 60% výkone ich svieti 6. Úlohou je testovať, či pri každom výkonom režime svieti požadovaný počet LED. Z uvedených snímok taktiež vidno, že v blízkosti LED panela je ďalšia LED, ktorá nesmie daný test ovplyvňovať.



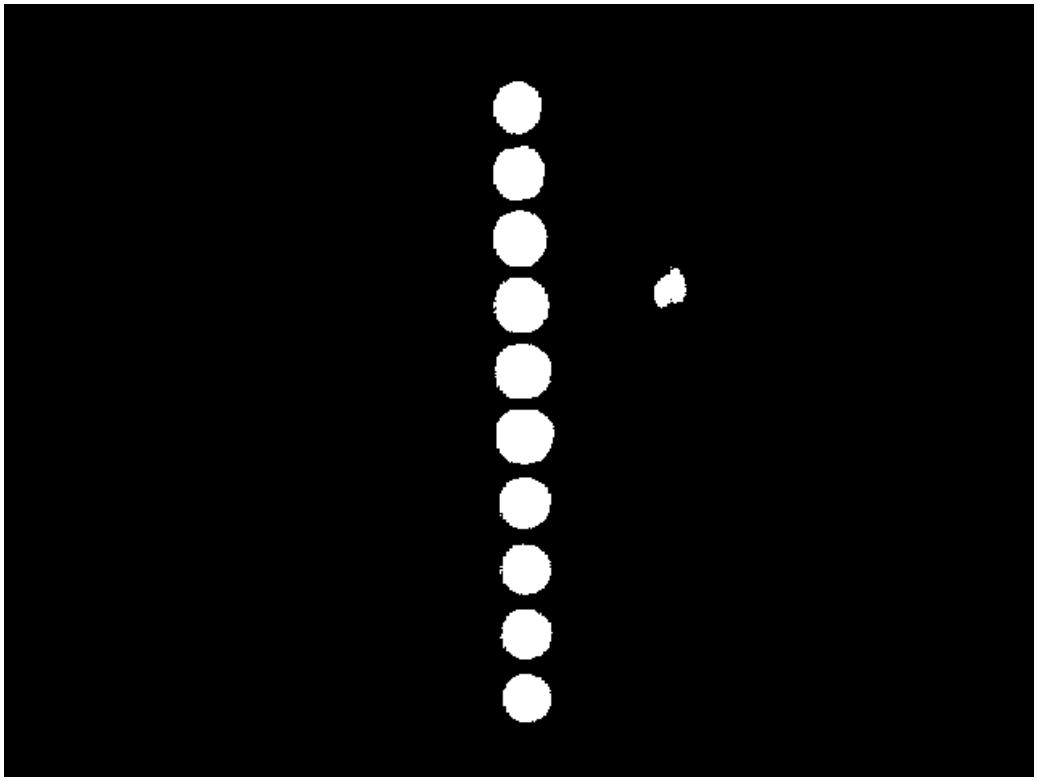
Obrázok 10. Snímka LED panela pomocou USB kamery pri plnom výkone.



Obrázok 11. Snímka LED panela pomocou USB kamery pri 60% výkone.

5.2 VYTVORENIE BINÁRNEHO OBRAZU

Prvou požiadavkou na použitie algoritmov pre vyhľadávanie objektov v obraze je vytvorenie binárneho obrazu. Požadované je, aby svietiace LED mali v binárnom obraze hodnotu pixelov 255 a okolie hodnotu 0. Ďalej je potrebné, aby svietiaca LED vytvorila samostatný objekt a nebola spojená so žiadnou susednou LED. Na prahovanie som využil už vytvorenú a popísanú funkciu *Prahovanie* a jej optimalizovanú variantu pomocou SSE *SSE_Prahovanie*. Najlepšie výsledky som dosahoval s prahom 130. Naprahovaný obrázok LED panelu pri 100% výkone je na obrázku 13.



Obrázok 12. Naprahovaný obraz LED panelu pri 100% výkone s prahom 130.

Z obrázku 13 vidno, že naprahovaním sme dokázali oddeliť svietiace LED od pozadia, každá LED tvorí samostatný objekt oddelený od ostatných LED. Na obrázku vidno aj LED, ktorá nie je súčasťou indikačného panelu, a teda sa pri

počítaní nesmie brať do úvahy. Riešením je to, že algoritmus bude prehľadávať iba tú oblasť obrazu, kde je indikačný LED panel. Takto naprahovaný obraz je možné použiť na ďalšie spracovanie.

5.3 NÁVRH ALGORITMOV NA POČÍTANIE LED.

Pri návrhu metódy na počítanie LED musíme vytvoriť algoritmus, ktorý dokáže pracovať iba v určitej oblasti obrazu, aby sa zabránilo ovplyvňovaniu výsledkov objektami, ktoré nesúvisia s dekovaným LED panelom. Ďalšou požiadavkou je, aby algoritmus spoľahlivo spočítal počet svietiacich LED, pri rôznom počte rozsvietených LED.

Na riešenie tohto problému je možné použiť rôzne algoritmy, ktoré sú dobre popísané v [5]. Ja som sa rozhodol vyskúšať dve metódy, porovnať ich výsledky, optimalizovať pomocou SSE inštrukcií a zhodnotiť dosiahnuté výsledky.

Prvá metóda vychádza z toho, že svietiace LED tvoria pomerne rovnaké objekty majúce približne rovnaký počet pixelov. Princíp metódy spočíta v danej oblasti počet svietiacich pixelov a vydelí ich priemerným počtom pixelov na jeden svietiaci objekt.

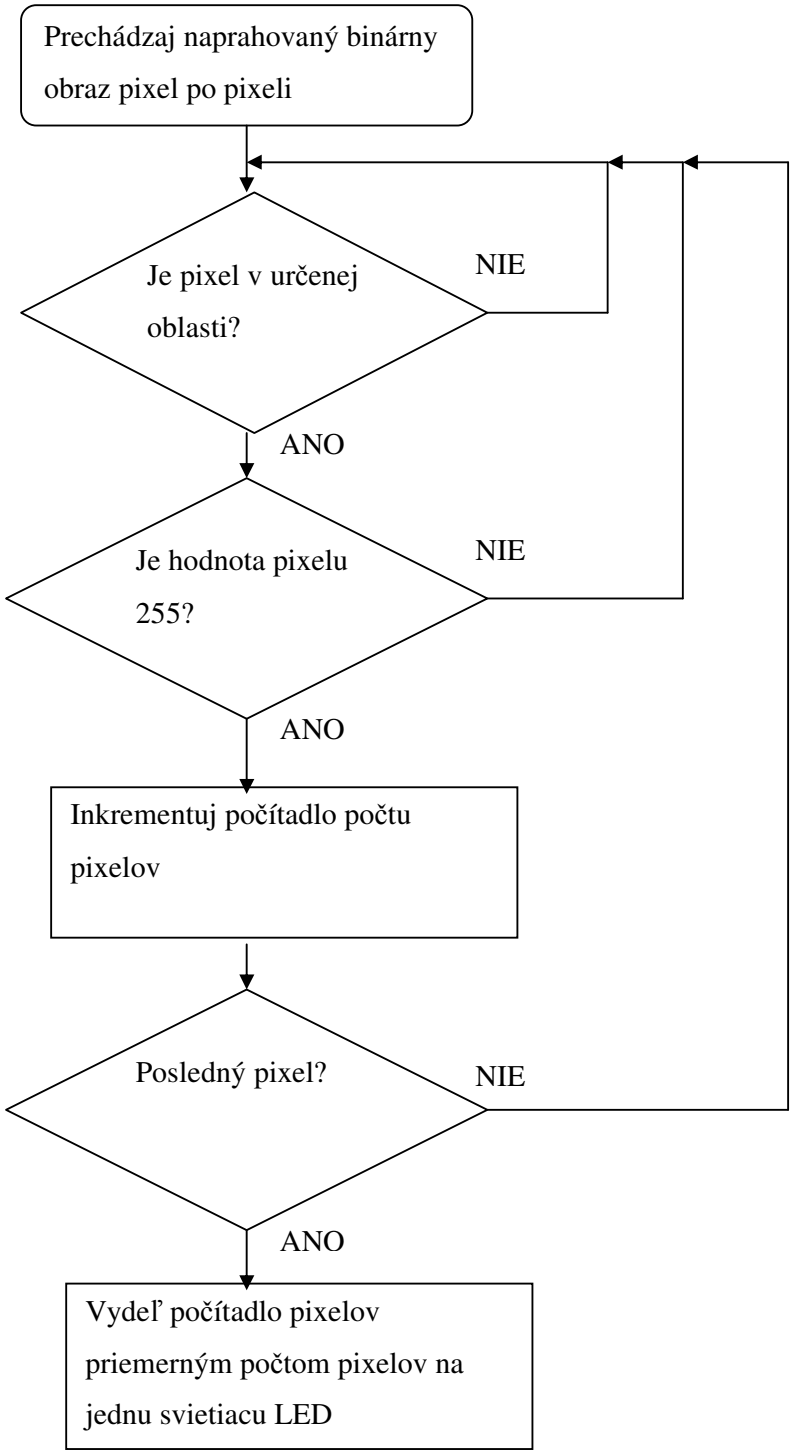
Druhá metóda vychádza z toho, že jednotlivé LED tvoria oddelené svietiace objekty, ktorých počet treba zrátať. Každý svietiaci objekt bude mať pridelené číslo objektu, potom len stačí určiť najväčšie číslo objektu, ktoré odpovedá počtu svietiacich LED. Označenie objektov som urobil pomocou štvorokolia.

5.4 POČÍTANIE LED POMOCOU PRIEMERNEHO POČTU PIXELOV.

V predchádajúcom odstavci som načrtol základný princíp tejto metódy. Motóda má následujúce kroky :

- Načítaj obrazové dáta vstupného obrazu.
- Naprahuj vstupný obraz pevným prahom.
- Urči oblasť v obraze, kde budú počítané pixely
- Prechádzaj určenú oblasť pixel po pixeli. Ak má pixel hodnotu 255, inkrementuj počítadlo počtu pixelov.
- Urči priemerný počet pixelov na jednu svietiacu LED.
- Vydeľ celkový počet pixelov, priemerným počtom pixelov na jednu LED.
- Celočíselný výsledok delenia určuje počet svietiacich LED.

Algoritmus naprogramovaný v jazyku c popisuje vývojový diagram na obrázku 14. Funkcia na počítanie LED pomocou priemerneho počtu pixelov je v dll súbore Led_pan.dll. Názov funkcie je *LedTestPriemer*. Funkcia používa klasické algoritmy naprogramované v jazyku c. Funkcia má 10 vstupných parametrov. Prvým je pointer na obrazové dáta vstupného obrazu. Druhým je pointer na obrazové dáta výstupného obrazu, tretím je šírka obrazu, štvrtým výška. Piatym parametrom je hodnota prahu. Šiesty a siedmy parameter sú x-ové hodnoty hranice oblasti, kde prebieha vyhľadávanie. Ďaľšie dva parametre sú y-ové hodnoty hranice oblasti, kde prebieha vyhľadávanie. Posledným parametrom je priemerná hodnota pixelu jednej svietiacej LED. Funkcia vracia návratovú hodnotu vo forate *int*, ktorá určuje počet svietiacich LED v danej oblasti. Funkcia má vo svojom tele volania dvoch funkcií. Prvou je funkcia *Prahovanie*. Táto funkcia už bola vysvetlená v predchádzajúcej kapitole. Ďalšou funkciou je *TestPocitanie*. V tejto funkcii sa aplikuje algoritmus popísaný na obrázku 14. Funkcia vracia celkový počet pixelov v danej oblasti, ktoré sa následne vydelia priemernym počtom pixelov na jednu LED a výsledok je predaný ako návratová hodnota funkcie *LedTestPriemer*.



Obrázok 13. Algoritmus pre počítanie LED pomocou priemerného počtu pixelov.

5.5 POČÍTANIE LED POMOCOU PRIMERNEHO POČTU PIXELOV S POUŽITÍM SSE INŠTRUKCIÍ.

Funkcia *LedTestPriemer* využívala pre výpočet klasické funkcia napísané v jazyku c. Túto funkciu budem optimalizovať pomocou SSE inštrukcií a následne budem obidve funkcie porovnávať. Optimalizovaná funckia má názov *SSE_LedTestPrimer* a má rovnaké vstupne parametre ako funkcia *LedTestPriemer.* Je taktiež umiestnená v knižnici *Led_pan.dll*. Táto funkcia najskôr volá funkciu *SSE_Prahovanie,* ktorá bola vysvetlená v predchádzajúcej kapitole. Funkcia prevedie vstupný obraz na binárny, na ktorom sú oddelené sietiace LED od pozadia. Ďalej je volaná funkcia *SSE_TestPocitanie*. Táto funkcia vracia počet pixelov v danej oblasti a je optimalizovaná s použitím SSE inštrukcií.

Funkcia *SSE_TestPocitanie* má v sebe cyklus for, ktorý prechádza vybranú oblasť po ôsmich pixeloch a vo vnútri cyklu je asm funkcia, ktorá je zobrazená na obrázku 15.

```
__asm__ __volatile__(
        "xorps      %%xmm0, %%xmm0\n\t"
        "xorps      %%xmm1, %%xmm1\n\t"
        "xorps      %%xmm2, %%xmm2\n\t"
        "xorps      %%xmm3, %%xmm3\n\t"
        "movups     (%0), %%xmm0\n\t"
        "movups     (%1), %%xmm1\n\t"
        "movups     (%2), %%xmm2\n\t"
        "pand       %%xmm0, %%xmm2\n\t"
        "punpcklbw  %%xmm3, %%xmm2\n\t"
        "paddw      %%xmm2, %%xmm1\n\t"
        "movups     %%xmm1, (%1)"
        : /* outputs */ "+r" (f1),"+r" (f2)
        : /* inputs */  "r" (odc)
        : /* clobbered */ "xmm0", "xmm1","xmm2","xmm3");

    f1=f1+8;
   }
```

Obrázok 14. Asm funkcia pre počítanie pixelov.

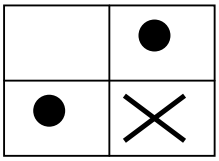
Funkcia má prvý parameter pole dát vstupného binárneho obrazu, druhý parameter je pole výstupných dat typu short int, kde sa bude ukladať počet pixelov.

Tretí parameterer *odc* je pole typu unsigned char so 16-timi rovnakými hodnotami 0x01, ktoré slúžia ako maska pri logickom súčte. Funkcia využíva štyri xmm registre. Prvé štyri riadky sú funkcie *xorps* vynulujúce všetky registre. Ďalšie tri funkcie sú funkcie *movups,* ktoré prevedú vstupné perametre do xmm registrov. Nasleduje funkcia *pand.* Táto funkcia spraví logický súčin medzi registrom, kde je načítaný vstupný binárny obraz a registrom, kde sú hodnoty masky 0x01. Dôvod tejto funkcie je ten, že binárny obraz má dve hodnoty 255 a 0. Pre ďalšie spracovanie je potrebné, aby namiesto hodnoty 255 bola hodnota 1. Logicky súčin s touto maskou zabezpečí nahradenie hodnoty 255 hodnotou 1. Pokračuje funkcia *punpcklbw,* táto pretapuje hodnotu výsledku logického súčinu, ktorá je typu unsigned char, na short int. Ďalej nasleduje funkcia *paddw* pripočítajúca do poľa výsledkov hodnotu po logickom súčine. Pokiaľ bola výsledná hodnota logického súčinu rovná jedna, výsledok sa zvýši o jednotku. Ak sa rovnala nule, ostane rovnaký. Takýmto spôsobom funkcia spočíta počet pixelov v danej oblasti. Posledná funkcia *movups* prevedie výsledok z registra po poľa výsledkov.

Po funkci asm je ešte potrebné spočítať všetky prvky výstupného poľa f2, lebo výsledok je súčtom všetkých ôsmich prvkov tohto poľa. Po tomto spočítaní dostaneme hodnotu počtu pixelov svietiacich LED v celej prehľadávanej oblasti.

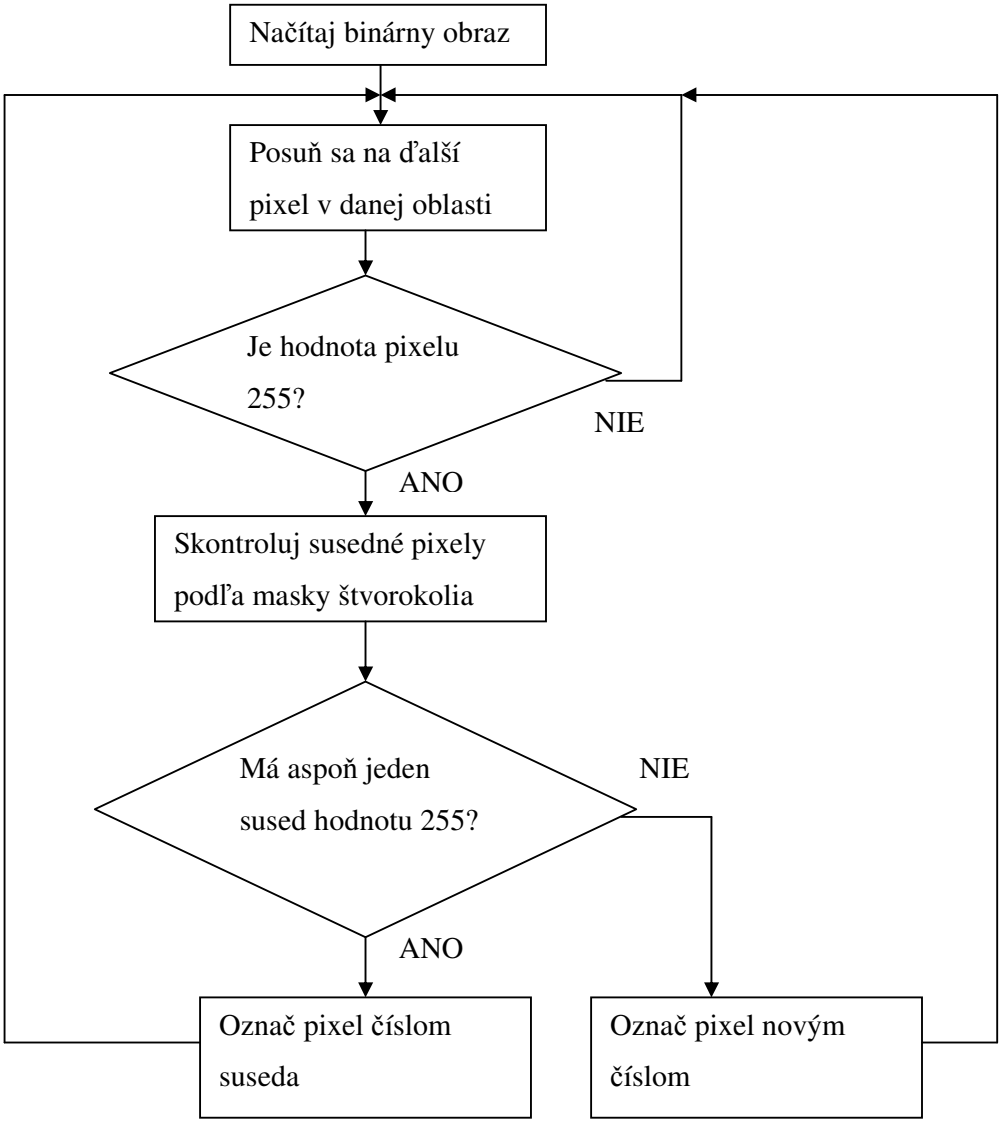
5.6 POČÍTANIE LED POMOCOU ŠTVOROKOLIA .

Pre popísanie objektov s pomocou štvorokolia som využil algiritmus popísaný v [27]. Využívam to, že každá svietiaca LED na binárnom obraze je samostatný objekt oddelený od ostatných susedov. Pre spoľahlivé počítanie objektov treba každý objekt označiť jedinečným číslom, potom stačí spočítať, koľko bolo pridelených čísel, prípadne zistiť najväčšie číslo, ak bolo číslovanie postupné.



Obrázok 15. Sledované pixely pre 4-okolie [7].

Na obrázku 16 je znázornené, ktoré pixely sa sledujú pomocou štvorokolia. Krížikom je označený aktuálny pixel a bodkou sú sledované susedné pixely. Princíp metódy je ten, že sa prechádza binárna obraz pixel po pixeli a pokiaľ je zistený pixel s hodnotou 255, skontrolujú sa susedné pixely podľa masky na obrázku 16. Ak tie pixely ešte nie su označené, pixel sa označí novým číslom. Ak označené sú, tak sa priradí číslo susedného pixelu.



Obrázok 16. Algoritmus označenia objektu pomocou štvorokolia

Funkcia na počítanie LED pomocou štvorokolia je súčasťou dll súboru Led_pan.dll a má názov *LedTestStvorOkolie*. Má 10 vstupných parametrov. Prvým je pointer na obrazové dáta vstupného obrazu. Druhým je pointer na obrazové dáta výstupného obrazu, tretím je šírka obrazu a štvrtým výška. Piatym parametrom je hodnota prahu. Šiesty a siedmy parameter sú x-ové hodnoty hranice oblasti, kde prebieha vyhľadávanie. Ďaľšie dva parametre sú y-ové hodnoty hranice oblasti, kde prebieha vyhľadávanie. Posledným parametrom je minimálny počet pixelov, ktorá má najmenšia svietiaca LED. Tento parameter je tam z toho dôvodu, aby sa medzi svietiace LED nazarátali aj malé oblasti, ktoré sa môžu vyskytnúť v obraze, no nepredstavujú svietiaci bod, ale šum. Funkcia vracia návratovú hodnotu vo formáte *int* určujúcom počet svietiacich LED v dannej oblasti.

Funkcia má vo svojom tele volania dvoch funkcií. Prvou je funkcia *Prahovanie*. Táto funkcia už bola vysvetlená v predchádzajúcej kapitole. Druhou funkciou je funkcia *TestZnacenie*. Funkcia má v sebe cyklus for, ktorý obsahuje algoritmus vysvetlený vývojovým diagramom na obrázku 17. Tento prechádza všetky pixely v určenej oblasti. Pokiaľ má niektorý pixel hodnotu 255, volaná funkcia *HladajObj* pomocou masky štvorokolia prehľadá susedné pixely a podľa nich priradí pixelu číslo oblasti, do ktorej patrí. Potom nasleduje funkcia *RatanieBlokov*, táto rieši situáciu, pri ktorých má 1 pixel susedov s rozdielnymi číslami blokov. Funkcia upraví čísla blokov a zlúči bloky majúce priamu hranicu. Nakoniec zráta počet blokov, odfiltruje len tie, ktoré majú väčší počet pixelov ako bol nami určený minimálny počet a vracia počet blokov. Tento počet blokov sa rovná počtu svietiacich LED.

5.7 POČÍTANIE LED POMOCOU ŠTVOROKOLIA S POUŽITÍM SSE INŠTRUKCIÍ.

Funkciu na počítanie LED pomocou štvorokolia som optimalizoval pomocou SSE inštrukcií. Funkcia má názov *SSE_LedTestStvorOkolie* a je nakompilovaná v dll súbore *Led_pan.dll*. Funkcia má v sebe volania dvoch funkcií. Prvou je funkcia *SSE_Prahovanie*, ktorá vytvorí binárny obraz podľa zadaného prahu. Funkciu sme už spomínali v predchádzajúcej kapitole. Ďalšou funkciou je *SSE_TestZnacenie*. Táto

funkcia v sebe aplikuje algoritmus vysvetlený vývojovým diagramom na obrázku 17. Rovnako ako funkcia *TestZnacenie* má v sebe cyklus for prehľadávajúci určenú oblasť pixel po pixeli. Pokiaľ sa hodnota pixelu rovná 255, je volaná funkcia *SSE_HladajObj*. Táto funkcia je optimalizovaná SSE inštrukciami a prehľadáva susedné pixely podľa masky štvorokolia.

```
__asm__ __volatile__(
        "xorps      %%xmm0, %%xmm0\n\t"
        "xorps      %%xmm1, %%xmm1\n\t"
        "xorps      %%xmm2, %%xmm2\n\t"
        "movups     (%1), %%xmm0\n\t"
        "movups     (%2), %%xmm1\n\t"
        "movups     (%3), %%xmm2\n\t"
        "pcmpeqd    %%xmm0, %%xmm1\n\t"
        "pcmpeqd    %%xmm0, %%xmm2\n\t"
        "por        %%xmm2, %%xmm1\n\t"
        "movups     %%xmm1, (%0)"
        : /* outputs */"+r" (vys),"+r" (dat_n)
       : /* inputs */ "r" (pn_x1),"r" (pn_y1)
       : /* clobbered */ "xmm0", "xmm1","xmm2");
```

Obrázok 17. Asm funkcia pre prehľadávanie susedných pixelov

Na obrázku 18 je asm funkcia prehľadávajúca susedné pixely pomocou masky štvorokolia. Funkcia má štyri vstupné parametre. Prvým je *vys*, čo je pole výsledkov, do ktorého sa ukladajú výsledky z porovnávania susedných pixelov. Druhým je parameter *dat_n,* čo je pole, kde su uložené čísla oblastí všetkých predchádzajúcich pixelov. Tretí parameter je *pn_x1*, čo je susedný pixel v x-ovej osi. Posledným parametrom je *pn_y1,* ktorý je susedný pixel v y-ovej oblasti.

Prvé tri riadky sú inštrukcie *xorps* vynulujúce všetky 3 xmm registre, ktoré sa v tejto funkcií používajú. Nasledujú 3 riadky s inštrukciami *movup.* Tieto načítajú tri vstupné parametre do xmm registrov. Nasleduje prvá inštrukcia *pcmpeqd* porovnávajúca dva registre s hodnotami poľa čísel všetkých predchádzajúcich pixelov a susedného pixelu v x-ovej osi. Táto inštrukcia zapíše do výstupneho registra hodnotu 255, ak sa registre rovnajú a 0 ak sa registre nerovnajú. Nasleduje rovnaká inštrukcia *pcmpeqd* porovnávajúca register s hodnotami všetkých

predchádzajúcich pixelov so susedným pixelom v y-ovej osi. Nasleduje inštrukcia *por*, ktorá vykoná logický súčet medzi dvoma registrami s výsledkom porovnávania. Pokiaľ aspoň jeden zo susedných pixelov bol už predtým označený, výsledok porovnávania a logického súčtu bude 255. Posledná inštrukcia *movups* presunie výsledok z registra do poľa výsledkov. Ak bol nájdený aspoň jeden sused s prideleným číslom, číslo tohto suseda je v nasledovných funkciach priradené aktuálnemu pixelu. Pokiaľ sa žiadny sused nenašiel, pridelí sa mu nové číslo. Funkcia *SSE_TestZnacenie* nakoniec zavolá funkciu *RatanieBlokov*, ktorá zlúči oblasti priamo susediace a vráti počet objektov, ktoré sú väčšie ako je minimálna hranica.

5.8 NAMERANÉ VÝSLEDKY NA TESTOVACÍCH SNÍMKACH.

Testovanie funkcií na detekovanie počtu svietiacich LED na LED paneli som uskutočnil na 34 testovacích snímkach. Snímky boli robené USB kamerou, pri rôznych režimoch LED panelu, teda pri rôznom počte svietiacich LED. Snímky boli robené vo viacerých skupinách, takže kamera nemala vždy úplne rovnakú polohu a taktiež svetelné podmienky boli rôzne. Testovacie snímky pokrývajú všetky kombinácie počtu svietiacich LED.

Testovanie bolo robené pomocou GUI aplikácie popísanej v ďalšej kapitole. Počítač, na ktorom bolo testovanie urobené bol osadený procesorom INTEL Core TM2 Duo, rýchlosť procesora 2GHz, procesor podporoval MMX,SSE aj SSE2 technológiu.

Testoval som všetky 4 funkcie, teda *LedTestPriemer,SSE_LedTestPriemer, LedTestStvorOkolie a SSE_LedTestStvorOkolie.* Výsledky sú zobrazené v Tabuľke 1. Stĺpec Testovacia snímka má dva podstĺpce. Prvým je číslo snímky a druhým je skutočný počet svietiacich LED na snímke. Potom nasledujú štyri stĺpce, kde každá funkcia má jeden a každý stĺpec má dva podstĺpce. Prvým je čas vykonávania funkcie v milisekundách a druhým je počet detekovaných LED funkciou. Všetky funkcie mali nastavené oblasti na snímku :X1=270, X2=345,Y1=10,Y2=470. Hodnotu prahu mali všetky funkcie 130. Funkcie *LedTestPriemer a SSE_LedTestPriemer* mali priemernu hodnotu oblasti 800 pixelov. Funkcie

LedTestStvorOkolie a SSE_LedTestStvorOkolie mali minimálnu hodnotu oblasti 500 pixelov.

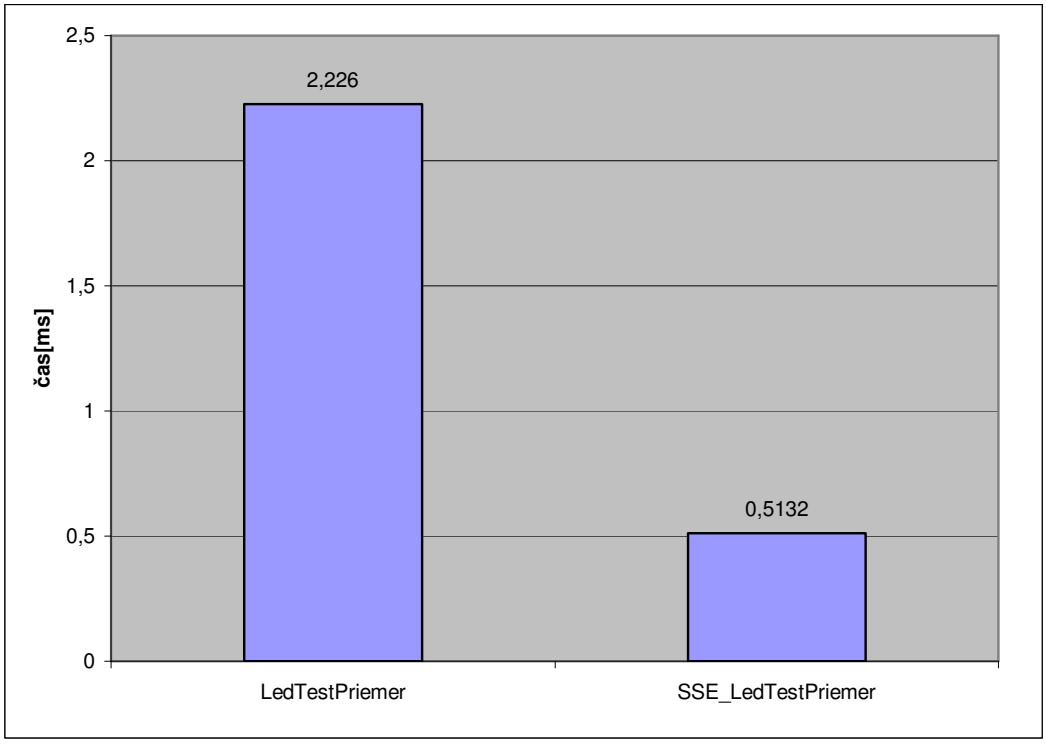
Testovacia snímka		LedTest Priemer		SSE_LedTest Priemer		LedTest StvorOkolie		SSE_LedTest StvorOkolie	
Číslo	Počet LED	Čas [ms]	Detek. Led	Čas [ms]	Detek. Led	Čas [ms]	Detek. Led	Čas [ms]	Detek. Led
1	9	2,197	9	0,505	9	195,795	9	92,31	9
2	10	2,224	10	0,509	10	237,443	10	111,766	10
3	7	2,227	7	0,505	7	130,421	7	63,059	7
4	6	2,191	6	0,509	6	88,452	6	42,616	6
5	3	2,196	3	0,506	3	24,868	3	12,824	3
6	2	2,194	2	0,506	2	13,932	2	5,582	2
7	1	2,184	1	0,506	1	4,347	1	3,244	1
8	0	2,22	0	0,508	0	2,521	0	0,668	0
9	4	2,198	4	0,503	4	47,104	4	23,617	4
10	5	2,202	6	0,532	6	80,327	5	38,041	5
11	6	2,211	7	0,505	7	119,88	6	57,081	6
12	7	2,218	8	0,505	8	161,87	7	76,004	7
13	7	2,216	9	0,506	9	178,674	7	82,736	7
14	8	2,247	9	0,549	9	199,159	8	92,817	8
15	8	2,226	9	0,503	9	179,774	8	82,534	8
16	9	2,223	10	0,508	10	231,695	9	108,551	9
17	10	2,234	12	0,504	12	295,031	10	138,59	10
18	7	2,209	8	0,552	8	145,92	7	70,378	7
19	7	2,22	9	0,508	9	170,104	7	81,209	7
20	7	2,614	8	0,556	8	136,822	7	64,284	7
21	7	2,253	8	0,503	8	138,378	7	65,75	7
22	6	2,339	6	0,537	6	96,247	6	46,096	6
23	5	2,238	5	0,505	5	67,878	5	32,522	5
24	4	2,145	4	0,522	4	40,835	4	20,108	4
25	**3**	2,198	3	0,507	3	23,975	3	13,537	3
26	2	2,195	2	0,528	2	11,569	2	6,937	2
27	1	2,19	0	0,507	0	5,823	1	1,871	1
28	5	2,195	5	0,503	5	70,643	5	34,24	5
29	6	2,211	6	0,507	6	106,216	6	50,166	6
30	7	2,216	8	0,505	8	140,857	7	66,959	7
31	8	2,22	8	0,509	8	181,981	8	85,674	8
32	8	2,209	9	0,509	9	194,083	8	91,349	8
33	10	2,245	11	0,519	11	284,543	10	131,876	10
34	9	2,2	10	0,505	10	237,598	9	110,797	9

Tabuľka 1. Namerané hodnoty funkcií na testovacích snímkach.

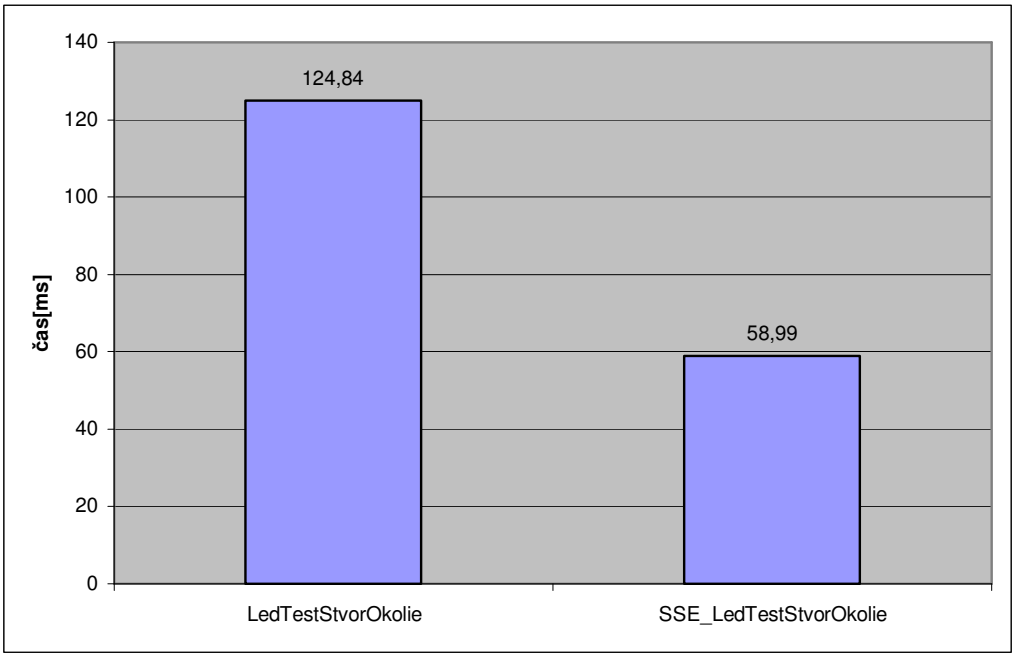
	LedTest Priemer	SSE_LedTest Priemer	LedTest StvorOkolie	SSE_LedTest StvorOkolie
Počet správne detekovaných snímkov	**17**	**17**	**34**	**34**
Percento správne detekovaných snímkov	**50%**	**50%**	**100%**	**100%**

Tabuľka 2. Úspešnosť detekovania počtu LED na jednotlivých snímkoch funkciami.

V Tabuľke 2 je počet správne detekovaných snímkov jednotlivými funkciami. Správne detekovaný snímok znamená, že návratová hodnota funkcie sa rovná skutočnému počtu svietiacich LED na snímke.

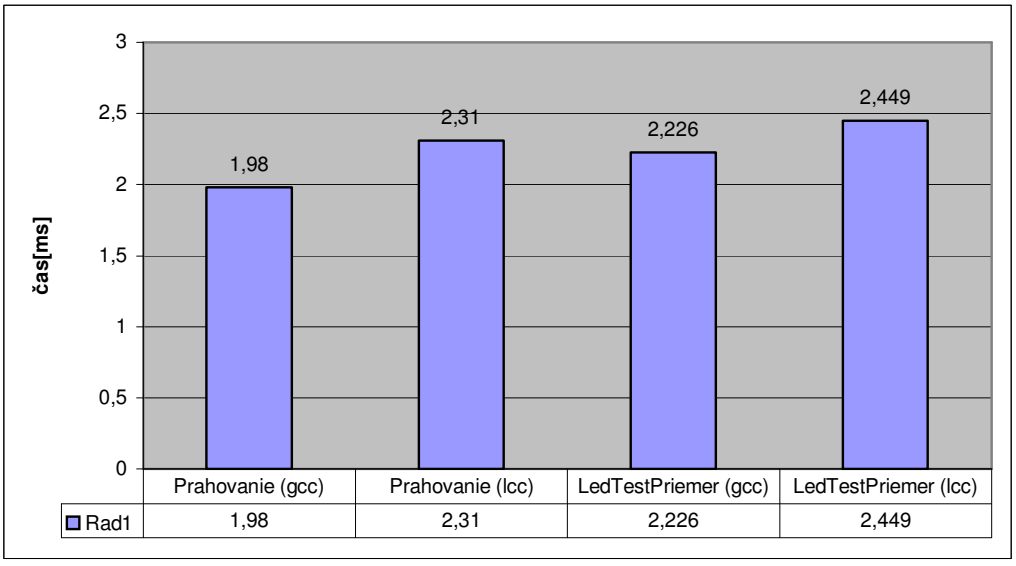


Obrázok 18. Priemerný čas funkcií *LedTestPriemer a SSE LedTestPriemer*



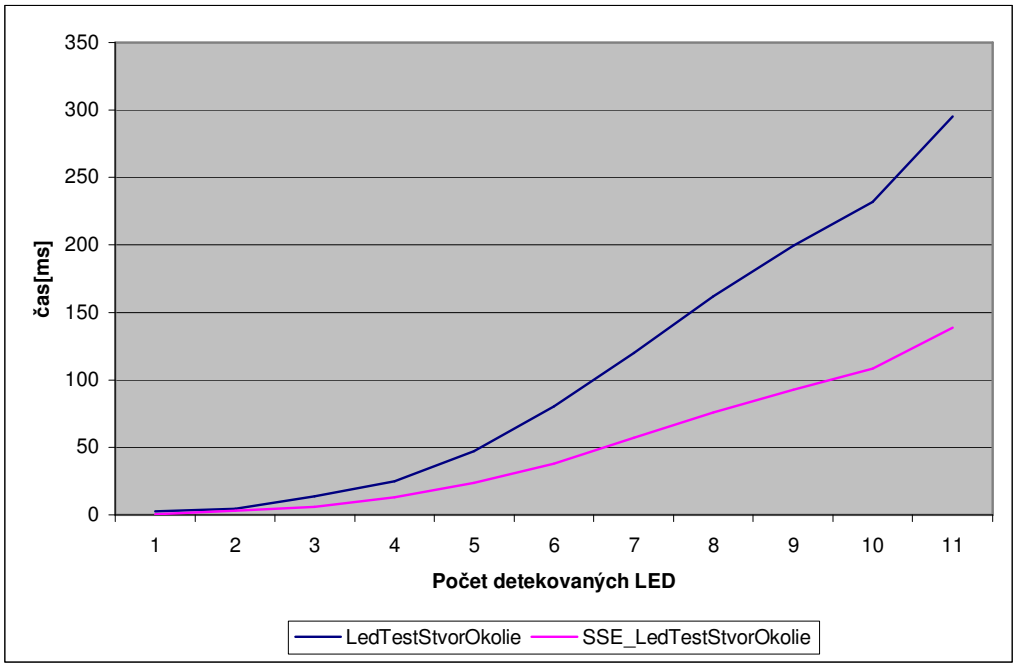
Obrázok 19. Priemerný čas funkcií *LedTestStvorOkolie a SSE_LedTestStvorOkolie*

Na obrázkoch 19 a 20 je porovnanie času funkcií bez použitia SSE inštrukcií a s použitím SSE inštrukcií. Z tabuľky 1 som vypočítal priemerne časy jednotlivých funkcií.



Obrázok 20. Porovnanie funkcií kompilovaných dvomi kompilátormi.

Na obrázku 21 je porovnanie dvoch funkcií *Prahovanie* a *LedTestPriemer*. Obidve funkcie boli skompilované dvomi kompilátormi. Prvý bol kompilátor gcc, na ktorom sú skompilované aj všetky ostatné funkcie. Druhým bol kompilátor LCC, na ktorom boli skompilované iba tieto dve funkcie. Skompilované DLL funkcie kompilátorom LCC sú v dll súbore test_obr_lcc.dll. Z nameraných údajom môžeme porovnať, ako jednotlivé kompilátory ovplyvňujé časy jednotlivých funkcií.



Obrázok 21. Závislosť trvania funkcie LedTestStvorOkolie od počtu LED.

Na obrázku 22 je grafická závislosť trvania funkcií LedTestStvorOkolie a SSE_LedTestStvorOkolie od počtu svietiacich LED. Časy obidvoch funkcií narastajú s počtom detekovaných LED z dôvodu stále väčšieho poľa na ukladanie výsledkov susedných hodnôt, ktoré treba pri každom pixeli prehľadávať.

6. UŽIVATEĽSKÉ ROZHRANIE

Pre prácu s funkciami je potrebné vytvoriť uživateľské rozhranie, prostredníctvom ktorého budú uživatelia volať jednotlivé funkcie a zobrazovať výsledky.

6.1 POŽIADAVKY NA UŽIVATEĽSKÉ ROZHRANIE

Uživateľské rozhranie slúži na komunikáciu uživateľa s programom. Keďže funkcie sú zakomponované v dll súboroch, je zrejmé, že program musí pracovať v operačnom systéme Windows, pre ktorý sú dll súbory určené. Každopádne aj ostatné operačné systémy majú svoje dynamické knižnice, v ktorých by dané funkcie mohli pracovať aj pod inými operačnými systémami. V tomto prípade sa budem zaoberať iba operačným systémom Windows, avšak bolo by dobré, pokiaľ by mal program možnosť, využitia aj pre iné operačné systémy. Pravda, s menšími úpravami. Prvou požiadavkou je teda, aby bol program z uživateľským rozhraním spustiteľný pod operačným systémom Windows.

Pri tomto operačnom systéme je možnosť použiť konzolovú aplikáciu alebo grafické rozhranie (GUI rozhranie). Vzhľadom na to že konzolové aplikácie nie su uživateľsky veľmi priateľské, na zadanie úlohy plynie druhá požiadavka, a síce, aby mal program grafické rozhranie.

Program bude spracovávať Dll funkcie vytvárajúce sae nezávisle od uživatľského rozhrania, preto je nutné, aby mal program možnosť pridať Dll funkcie bez nutnosti kompilovania programu. I,nými slovami treťou požiadavkou je, aby program umožňoval dynamicky pridávať Dll funkcie.

Keď to zhrniem, program musí byť spustiteľný pod operačným systémom Windows, má mať grafické rozhranie jako aj umožňovať dynamicky pridávať Dll funkcie.

6.2 VÝBER APLIKÁCIE PRE TVORBU UŽIVATEĽSKÉHO ROZHRANIA

Programovanie grafického rozhrania pod operačným systémom Windows je možné robiť viacerými spôsobmi. Programy komunikujú so systémom Windows pomocou Windows API rozhrania. Je možné teda používať priamo API funkcie, ktoré sa nachádzajú v hlavičkovom súbore windows.h. Používanie týchto funkcií je však málo pohodlné, napríklad pre vykreslenie jednoduchého okna je potrebné napísať niekoľko riadkov kódu. Pre uľahčenie práce programátorom je vytvorených viacero projektov, ktoré používajú knižnice, kde sú vytvorené funkcie oveľa jednoduchšie a prehľadnejšie. Existuje množstvo projektov, či už platených alebo neplatených. Niektoré sú „Open Source“, čo znamená, že majú otvorený kód a užívateľ ho môže podľa potreby meniť. V tejto práci chcem používať programy voľne šíriteľné. Jedným z takýchto projektov je projekt wxWidgets [10]. Tento projekt je „Open Source“, voľne šírieľný a aj multiplatformný projekt, ktorý možno používať okrem operačného systému Windows aj v iných operačných systémoch (Unix, Linux, MAC OS). Vzhľadom na tieto vlastnosti aj fakt, že je projekt dobre zdokumentovaný, moja voľba padla práve na tento projekt.

6.3 POPIS KNIŽNÍC WXWIDGETS

Knižnice wxWidgets je možné stiahnuť z [10]. Knižnice wxWidgets obsahujú množstvo tried a metód pre vytváranie grafického užívateľského rozhrania. Aplikácie zobrazujú okná a ostatné užívateľské nástroje, prijímajú vstupy z myši klávesnice i ostatných zariadení. Taktiež aplikácie dávajú výstup užívateľovi a ovládajú rôzne periférne zariadenia [3].

WxWidgets sa sťahuje v zdrojových kódoch, je potrebné ich priamo skompilovať a nakonfigurovať pre daný operačný systém, na ktorom budú pracovať. Možno ich skompilovať, používať v rôznych kompilátoroch a v rôznych vývojových prostrediach. Ja som sa rozhodol používať kompilátor gcc, použitý aj pre tvorbu dll knižníc. Rovnako ako pri dll knižniciach, som použil IDE Code Block.

Toto IDE má aj tú výhodu, že umožňuje automaticky generovať šablóny funkcií wxWidgets. Má grafický nástroj wxSmith, pomocou ktorého je možné graficky

vytvárať jednotlivé okná a automaticky generuje wxWidgets funkcie. Kompilácia wxWidgets pod Windows za pomoci prekladača gcc je popísaná v [3]. Keďže pri tejto aplikácii budem používať aj knižnice OpenCV, je potrebné pred kompiláciou wxWidgets upraviť súbor setup.h v adresári wxWidgets/wx/msw/setup.h. V tomto súbore sa nastavujú jednotlivé makrá, a tak sa upravujú výsledné wxWidgets knižnice podľa potreby programátora. Je potrebné nastaviť na 0 makrá: wxUSE_DRAG_AND_DROP, wxUSE_CLIPBOARD, wxUSE_OLE, wxUSE_OLE_AUTOMATION [3].

Knižnica wxWidgets využíva koncepciu objektovo orientovaného programovania. Každé okno je reprezentované ako C++ objekt. Každý objekt má dobre definované správanie, môže prijímať a reagovať na jednotlivé udalosti [3].
Príklad na zobrazenie základného okna s jedným menu na ukončenie programu je na obrázku 23. Obrázok vysvetľuje základnú filozofiu knižníc wxWidgets. Najskôr sa definuje trieda MyApp, ktorá je odvodená od triedy wxApp, čo je trieda knižníc wxWidgets. Táto trieda definuje jednu funkciu OnInit(). Funkcia je vždy volaná pri spustení programu a v nej je potom možné volať triedy na vykreslenie okna. Ďalšou deklarovanou triedou je trieda MyFrame, odvodená z triedy wxFrame, čo je rovnako trieda wxWidgets. V tejto triede je definovaný konštruktor, ktorý má ako parameter názov okna. Ďalej je deklarovaná funkcia, ktorá sa spustí pri kliknutí v menu na Quit. Táto funkcia má ako parameter udalosť, ktorá ju volá a bude deklarovaná ďalej. Tieto udalosti sú deklarované v EVENT_TABLE. Ide o tabuľku zobrazujúcu všetky udalosti, ktoré nastanú v danej triede, v našom prípade je to kliknutie na menu. Tabuľka aj definuje, ktorá funkcia je pri tejto udalosti volaná. Samotné vykreslenie okna zabezpečí funkcia OnInit(). Vykreslenie okna je na dvoch riadkoch, pričom najskôr sa vytvorí objekt okna frame. Konštruktoru sa dáva ako parameter názov okna. Potom je týmto objektom volaná funkcia Show(true), ktorá pri parametri true vykreslí okno. Pri kliknutí na menu Exit sa zavolá funkcia OnQuit() volajúca funkciu Close(), ktorá odstráni okno a ukončí sa program.

```
class MyApp:public wxApp
{
 public:
        virtual bool OnInit();
};

class MyFrame:public wxFrame
{
   MyFrame(const wxString& title);
   void OnQuit(wxCommandEvent& event);
private:
  DECLARE_EVENT_TABLE()
};

DECLARE_APP(MyAPP)
IMPLEMENT_APP(MyApp)

bool MyApp::OnInit()
{
 MyFrame *frame=new MyFrame(wxT("Minimal wxWidgets App"));
 frame->Show(true);
 return true;
}

       BEGIN_EVENT_TABLE(MyFrame, wxFrame)
         EVT_MENU(wxID_EXIT, MyFrame::OnQuit)
       END_EVENT_TABLE()

Void MyFrame::OnQuit(wxCommandEvent& event)
{
  Close();
}
```

Obrázok 22. Príklad na zobrazenie okna v knižnici wxWidgets [3].

6.4 SPRACOVANIE DLL SÚBOROV

Načítavať funkcie z Dll súborov možno dvomi spôsobmi. Prvým spôsobom je importovať knižnicu pri kompilácii s tým, že sa k projektu pridá hlavičkový súbor Dll knižnice a jednotlivé funkcie sa volajú ako klasické funkcie. Jeho nevýhodou je, že Dll súbor musí byť k dispozícii pri kompilácii a linkovaní projektu. Pri pridaní

ďalšej Dll knižnice je treba projekt znovu skompilovať. Výhoda spočíva v tom, že funkcie sú volané priamo a nie je potrebný žiadny dodatočný kód [11].

Druhým spôsobom je dynamické volanie funkcií v Dll knižniciach. Pri tomto spôsobe nie je potrebné importovať dll knižnicu pri kompilácii, ale sa používajú funkcie, ktoré spúšťajú funkcie v dll knižniciach, pričom názvy dll súborov a názvy funkcií sú predávané ako parametre. Veľkou výhodou je to, že do projektu sa môžu pridávať funkcie z Dll knižníc počas prevádzky aplikácie bez nutnosti kompilovania. Je potom možné pridávať ľubovoľné funkcie v ľubovoľných súboroch s tým, že funkcie musia mať určitý počet parametrov definovaných už pri kompilácii. Prípadne, ak chceme použiť rôzne parametre, musíme ich poznať už pri kompilácii. Nevýhodou je nutnosť napísať kód navyše pre obsluhu funkcií, ktoré budú volané. Keďže volanie Dll funkcií prvým spôsobom je rovnaké ako klasických funkcií, nie je potrebné tento spôsob vysvetľovať. Stručne popíšem druhý spôsob volania Dll funkcií, ktoré bude aplikácia používať.

Keďže Dll súbory sa používajú iba v operačnom systéme Windows a wxWidgets je robená ako multiplatformná, triedy na spracovanie dll knižnic nie sú vo wxWidgets aplikované. Pre prácu s Dll funkciami je potrebné použiť funkcie z knižnice windows.h. Na obrázku 10 je časť kódu pre volanie Dll funkcií, na ktorom vysvetlím jednotlivé funkcie. Funkcie pre dynamicky volané Dll funkcie sú popísané v [11] a z tohto popisu som vychádzal. Najskôr sa pomocou typedef nadefinuje dátový typ ofunc1. Tento typ je funkcia s parametrami, ktorá bude z dll knižnice volaná. Potom nasleduje funkcia LoadLibrary s parametrom názov dll súboru. Táto funkcia otvorí Dll súbor. Pokiaľ v parametri nie je daná cesta, ale iba názov súboru, je Dll súbor hľadaný najskôr v pracovnom adresári aplikácie, potom v adresároch daných v premennej prostredia PATH. Ak je súbor nájdený, funkcia vracia pointer na súbor, pokiaľ nie je nájdený, funkcia vracia NULL. Ďalej nasleduje funkcia GetProcAddress majúca dva parametre. Prvým je pointer na Dll súbor, zistený v predchádzajúcej funkcii a druhým parametrom je názov funkcie v danom súbore, ktorú chceme volať. Funkcia vracia pointer na funkciu v Dll súbore, ak je funkcia v danom súbore nájdená a NULL pokiaľ sa funkcia v danom Dll súbore nenašla.

Potom sa spustí funkcia v dll súbore, nájdená funkciou GetProcAdress, s parametrami, aké boli definované. Nakoniec sa zavolá funkcia FreeLibrary, ktorá uvoľní volanú funkciu z pamäte.

```
HINSTANCE hLib;
    typedef void (WINAPI*ofunc0)(unsigned char*,unsigned char*,unsigned
int,unsigned int);
ofunc0 funkcia;
 hLib=LoadLibrary(subor);

  if(hLib==NULL)
 {
  FreeLibrary((HMODULE)hLib);
  return -1; // Dll subor nenajdeny
 }

  funkcia=(ofunc0)GetProcAddress((HMODULE)hLib,Dfunkcia);
if(funkcia==NULL)
 {
 FreeLibrary((HMODULE)hLib);
return -2; // Dll funkcia nenajdena
  }

funkcia(obraz1->GetData(),obraz2->GetData(),(unsigned int) obraz1->sirka() ,
(unsigned int)obraz1->dlzka());

FreeLibrary((HMODULE)hLib);
```

Obrázok 23. Príklad funkcií pre dynamické volanie Dll funkcií.

6.5 POPIS JEDNOTLIVÝCH FUNKCIÍ A TRIED PROGRAMU

Aplikácia sa skladá z piatich zdrojových súborov gui_sseApp.cpp, gui_sseMain.cpp, adddll.cpp, dynamicdll.cpp a obrazcv.cpp.

Súbor gui_sseApp.cpp má jednu triedu gui_sseApp a tá ma jednu funkciu OnInit. Táto funkcia sa volá pri spustení programu a vykreslí hlavné okno aplikácie.

Súbor gui_sseMain.cpp má jednu triedu gui_sseMain, ktorá má funkcie na vykreslenie jednotlivých grafických prvkov do hlavného okna. Má tiež funkcie, ktoré sa volajú po nejakých udalostiach, ktoré sa generujú po zmene nejakého ovládacieho prvku okna (napríklad stlačenie tlačidla a pod). Má aj funkcie reagujúce na stlačenie niektorého ovládacieho prvku menu. Taktiež má funkciu VypocetCyklu, ktorá slúži na určenie počtu cyklov procesora za jednu milisekundu, na základe čoho sa bude počítať trvanie jednotlivých funkcií.

Súbor adddll.cpp má dve triedy a to adddll a FunkcData. Trieda adddll má funkcie na vykreslenie dialógového okna pre pridávanie funkcií z dll súborov a všetkých jeho ovládacích prvkov, ako aj funkcie, ktoré sa volajú pri jednotlivých udalostiach komponentov dialogového okna (stlačenie tlačidla a podobne). Funkcie v tejto triede načítavajú z dialógového okna meno dll súboru, názov funkcie, návratovú hodnotu a typ funkcie. Všetky tieto parametre sa zapisujú do registrov. Druhá trieda v tomto súbore je FunkcData obsahujúca funkcie, ktoré tieto štyri parametre z registrov načítajú a predávajú funkciam, ktoré dynamicky volajú jednotlivé funkcie v dll súboroch.

Súbor obrazcv.cpp má dve triedy a to ObrazFile a VyslednyObraz. Obidve využívajú funkcie knižnice OpenCv na spracovanie obrazov. Trieda ObrazFile spracováva vstupný obraz alebo obrazy. Má konštruktor, ktorého parametrom je názov spracovaného súboru s jeho cestou. Tento konštruktor načíta obraz pomocou funkcie knižnice OpenCv a získa z neho všetky potrebné dáta (šírku, výšku, data vo formáte uchar a podobne). Pri volaní deštruktoru tejto triedy sa tieto dáta uvoľnia z pamäte. Funkcie sirka,dlzka,velkost vracajú hodnoty šírky, výšky a veľkosti načítaného obrazu. Funkcia GetData vracia pointer na obrazové dáta, čím sa predáva funkciám v dll súboroch. Funkcia ZobrazSubor vytvorí súbor zobraz.bmp následne načítaný do hlavného okna.

Výška a šírka tohto obrazu sa vypočíta z veľkosti hlavného okna, aby sa zobrazený obraz prispôsoboval zmene veľkosti hlavného okna. Trieda VyslednyObraz spracuvá dáta vracajúce funkcie a vytvára z nich obrazový súbor. Má konštruktor s dvomi parametrami, šírkou a výškou výsledného obrazu. Tieto sa určujú tak, že pokiaľ je vstupný obraz iba jeden, používajú sa jeho rozmery. Ak sú obrazy dva, berú sa veľkosti menšieho obrazu. Funkcia GetData, predáva pointer na výstupné dáta, ktorý sa predáva ako parameter funkciám v dll súboroch. Funkcia ZobrazVysledok vytvorí z výstupných dat výstupný obrazový súbor a jeho rozmery sa počítajú podľa veľkosti hlavného okna. Tento súbor sa potom zobrazuje v hlavnom okne. Funkcia UlozObraz vytvorí z výstupných dat obrazový súbor v takej veľkosti, ako mal vstupný obrazový súbor. Táto funkcia má jeden parameter, a to je cesta, kde bude uložený tento obraz. Pokiaľ je tento parameter prázdny, vytvorí tento súbor v pracovnom adresári aplikácie s názvom vysledok.bmp.

Súbor dynamic.dll.cpp má jednu triedu Dllfunkcia slúžiacu na dynamické volanie dll funkcií. Konštruktor tejto triedy má jeden parameter, a tým je počet cyklov procesora za mikrosekundu. Je to z toho dôvodu, aby sme mohli merať časy jednotlivých funkcií volaných dynamicky. V tejto triede sú tri funkcie. Prvou je funkcia Nacitaniefunkcie, kontrolujúca, či sú načítané obrazové súbory podľa typu jednotlivej funkcie a ukladá si cesty k jednotlivým súborom. Druhou funkciou je Start. Táto načíta vybranú funkciu z dll súboru podľa zadaných parametrov, funkciu zavolá a spočíta čas trvania danej funkcie. Treťou funkciou je TrvanieFce vracajúca vypočítanú hodnotu trvanie danej funkcie.

Súbor paramet.cpp slúži na vykreslenie okna pre zadávanie hodnoty prahu. Má jednu triedu *paramet,*ktorej konštruktor inicializuje a vzkreslí dialógové okno pre zadávanie hodnoty prahu. Funkcia tejto triedy *ParamNavr* vracia hodnotu prahu zadanú do dialógového okna. Táto funkcia je typu *unsigned char.*

Súbor paramet2.cpp má jednu triedu *paramet2*, ktorá vykresľuje dialógové okno na zadanie šiestich parametrov: hodnoty prahu, štyri údaje na zadávanie hraníc oblasti a jeden parameter na zadávanie minimálnej veľkosti objektu alebo priemerného počtu pixelov v jednom objekte. Na načítanie týchto parametrov z okna dialógu slúžia štyri funkcie. Funkcia *ParamNavr* vracia hodnotu prahu vo formáte

unsigned char. Funkcia *Dx1_Navr* vracia hodnotu prvej hranice v x-ovej osi vo formáte *unsigned int*. Funkcia *Dx2_Navr* vracia hodnotu druhej hranice v x-ovej osi vo formáte *unsigned int*. Funkcia *Dy1_Navr* vracia hodnotu prvej hranice v y-ovej osi. Funkcia *Dy2_Navr* vracia hodnotu druhej hranice v y-ovej osi. Funkcia *ObjemNavr* vracia hodnotu priemerného objektu alebo minimálnej veľkosti objektu.

Súbor cpu_info.cpp slúži na načítanie systémových informácii. Tento súbor bude bližšie vysvetlený v nasledujúcom odstavci.

6.6 NAČÍTANIE SYSTÉMOVÝCH INFORMÁCIÍ.

Táto aplikácia slúži na testovanie rôznych funkcií s použitím SSE inštrukcií. Je potrebné zistiť, či počítač, na ktorom aplikácia beží, podporuje SSE inštrukcie.Takisto zisťujeme aj informácie o procesore, aby sa dali výsledky porovnávať na rôznych počítačoch, a aby z nich bolo jasné, pri akých podmienkach meranie prebiehalo. Pre načítanie systémových informácii som vytvoril triedu CPU_Info, ktorá je umestnená v súbore cpu_info.cpp /h.

Táto trieda má 4 funkcie prístupné z iných tried, a pomocou ktorých sa odovzdávajú aplikácii systémové informácie.

Prvou funkciou je funkcia *NacitajProcesor()*.Táto vracia meno procesora ako textový reťazec wxString. Druhou funkciou je funkcia *MMX_support(),* ktorá vracia 1 ak procesor podporuje technológiu MMX a 0 ak ju nepodporuje. Treťou funkciou je funkcia *SSE_support()*. Funkcia vracia 1, ak procesor podporuje SSE inštrukcie a 0, ak tieto inštrukcie nepodporuje. Štvrtou funkciou je funkcia *SSE2_support()*. Táto vracia 1, ak procesor podporuje inštrukčnú sadu SSE2 a 0, ak ju nepodporuje.

Zisťovanie systémových informácií sa uskutoční pri inicializácii konštruktora tejto triedy. Informácie uvoľní procesor potom, ako sa zapíše do registra *eax* hodnota 0 alebo 1. Podľa zapísanej hodnoty vyčítame z ostatných registrov (*eax,ebx,ecx,edx)* parametre, ktoré potrebujeme na systémové informácie. Na zápis a načítanie týchto registrov je potrebné použiť funkciu *asm()* umožňúcu zadávať príkazy v assembleri. Keďže funkcia *asm* je dostatočne popísaná v predchádzajúcich kapitolách, nie je potrebné bližšie vysvetľovať jej význam.

Načítanie výrobcu procesora prebieha následovne. Najskôr zapíšeme do registru *eax* hodnotu 0 a načítame hodnoty registrov *eax,ebx,ecx,edx*. Tieto registre sú 32 bitové, a teda každý obsahuje 4 osembitové hodnoty ASCII kódu, ktoré sa spoja a vytvoria reťazec ktorý udáva jedinečný názov výrobcu procesora. Napríklad názov pre Intel je *GenuineIntel*.

Ďalšie informácie dostaneme po zapísani hodnoty 1 do registru *eax*. Potom načítame hodnoty registrov *eax* a *edx*, z ktorých získame 2 32-bitové hodnoty. Z nich zistíme rodinu procesora a jeho model, ďalej informácie o podpore MMX,SSE,SSE2 a iných funkcií. Dekódovanie týchto hodnôt je popísané v manuáloch jednotlivých výrobcov procesora. Na základe týchto informácií je možné presne zistiť typ procesora a podporu jednotlivých technológii. V tejto aplikácii som použil informácie k najpoužívanejším procesorom Intel a AMD, ktoré je možné bezplatne stiahnuť zo stránok týchto výrobcov. V prípade potreby rozšírenia je možné tieto informácie doplniť.

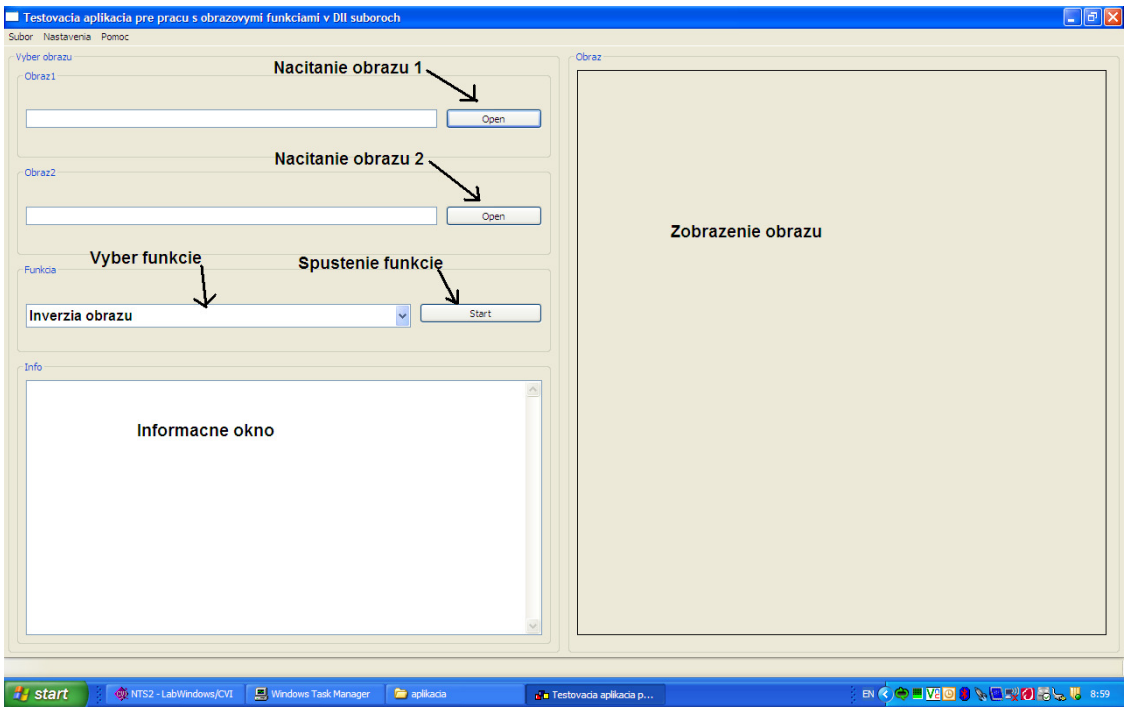
Jedinou systémovou informáciou, ktorá nie je zisťovaná v tejto triede, je informácia o rýchlosti procesora. Táto informácia sa získava pomocou funkcií na načítanie taktov procesora v súbore *rdsc.h*, rovnako ako pre funkciu na meranie času jednotlivých inštrukcií, ktoré boli popísané v kapitole 3.3.

6.7 POPIS APLIKÁCIE

Po spustení aplikácie sa zobrazí hlavný panel ako na obrázku 25. Aplikácia má v ľavej časti tlačidlá na načítanie obrazových súborov. Je možné pracovať s dvoma obrazmi podľa typu funkcie, ktorá je vybraná. Pokiaľ vybraná funkcia pracuje len s jedným obrazom, vyberá sa obraz z políčka Obraz 1. Načítanie obrazového súboru na prevedie stlačením tlaćidla Open v príslušnom obrazovom poli. Po stlačení sa zobrazí dialógové okno pre výber obrazového súboru.

Je možné vybrať súbory typu: bmp,jpg a jpeg, ako aj vyberať šedotónové aj farebné obrazy, farebné obrazy sa automaticky transformujú na šedotónové. Po výbere sa v pravej časti zobrazí vybraný obraz a v informačnom okne informácie o vybranom obraze (počet pixelov, šírka, výška, názov súboru a cesta). Zobrazený obraz v pravej

časti je transformovaný podľa aktuálnej veľkosti hlavného okna. Táto transformácia je iba pre informačné zobrazenie, na vybraný obraz nemá vplyv. Taktiež funkcie načítavajú originálny netransformovaný obraz. Výber funkcií sa vyberá pomocou Combo boxu Funkcia. Pokiaľ nie je do programu pridaná žiadna funkcia z nejakého dll súboru, sú ponúknuté iba dve funkcie InverziaObrazu a SSE_InverziaObrazu nalinkované staticky.

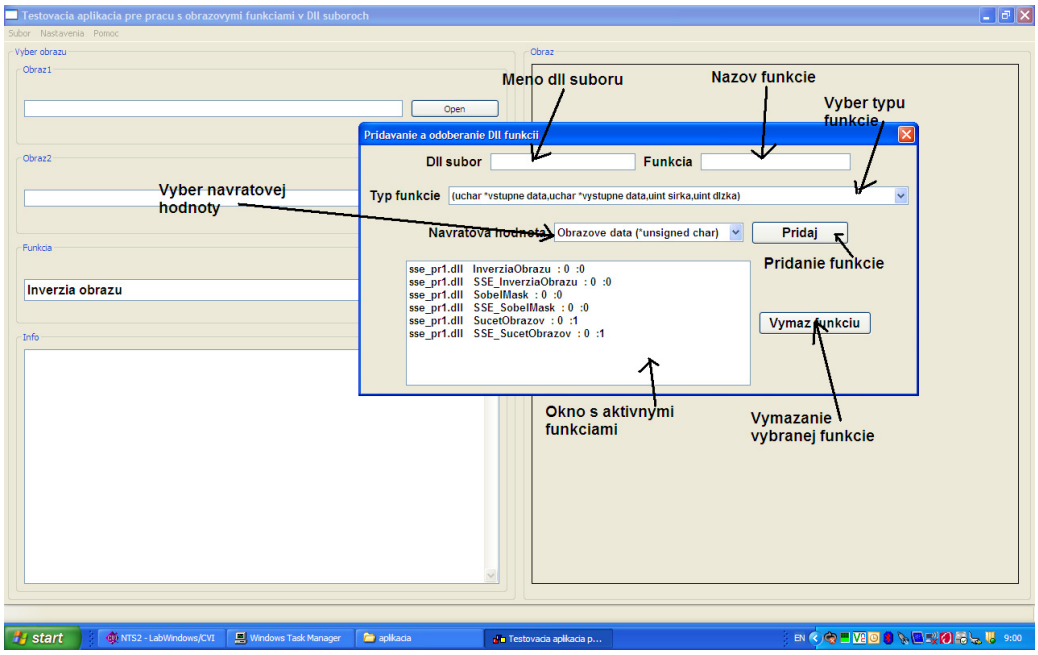


Obrázok 24. Zobrazenie hlavného okna aplikácie

Vybranú funkciu spustíme tlačidlom Start a výsledný obraz je zobrazený v okne Obraz. Tento obraz je taktiež trasformovaný podľa aktuálnej hodnoty hlavného okna.

Táto transformácia nemá vplyv na výsledný obraz, ktorý je uložený, pokiaľ nie je vybraný iný adresár a názov v pracovnom adresári aplikácie s názvom vysledok.bmp. Po vykonaní funkcie sa taktiež zobrazí v informačnom okne čas vykonania funkcie v mikrosekundách.

Aplikácia má tri hlavné menu. Prvé je Subor s jedným menu nazvaným ukončenie aplikácie. V menu pomoc je submenu nápoveda, kde sa zobrazí nápoveda k aplikacii a submenu O programe, kde sa zobrazia informácie o programe. V menu nastavenia je submenu Pridanie funkcie, po ktorého stlačení sa zobrazí okno na pridávanie a odoberanie funkcií. Tvar okna je zobrazený na obrázku 26.



Obrázok 25. Zobrazenie okna na pridávanie a odoberanie Dll funkcií

Okno na pridávanie funkcií slúži na pridávanie jednotlivých Dll funkcií do aplikácie. Do políčka Dll subor sa napíše meno Dll súboru aj s príponou dll. Pokiaľ tam nie je napísaná cesta, tak aplikácia hľadá dll súbor v pracovnom adresári aplikácie a potom v adresároch pridaných do premennej prostredia PATH. Do políčka Funkcia sa napíše meno funkcie ktorú chceme pridať, bez zátvoriek a parametrov.

V políčku Návratová funkcia si vyberieme typ návratovej funkcie. Zatiaľ je aplikovaná návratová funkcia Obrazové dáta vo formáte **unsigned char* alebo *textový súbor*. Podľa výberu návratovej funkcie sú ponúknuté typy funkcie, ktoré si vyberieme. Sú to parametre pridávanej funkcie. Pokiaľ je ako návratová hodnota

Obrazové dáta *unsigned char, sú ponúknute typy: *(*unsigned char vstupný obraz, *unsigned char výstupný obraz, uint šírka, uint výška*) , *(*unsigned char vstupný obraz1, *unsigned char vstupný obraz 2, *unsigned char výstupný obraz, uint šírka, uint výška), (*unsigned char vstupný obraz, *unsigned char výstupný obraz, uint šírka, uint výška,1 parameter uchar*) a *(*unsigned char vstupný obraz, *unsigned char výstupný obraz, uint šírka, uint výška,1 parameter uchar,5 parametrov uint*) Prvý typ sa používa pre funkcie spracúvajúce jeden vstupný obraz, druhý pre funkcie spracúvajúce dva vstupné obrazy. Tretí typ sa používa pokiaľ sa musí zadať pri spúšťaní funkcie jeden vstupný parameter. Tento typ sa používa pri funkcii Prahovanie, kde sa zadáva ako vstupný parameter hodnota prahu. Štvrtý typ je použitý pri funkciach, ktoré majú 6 vstupných parametrov. Jeden parameter je typu uchar a 5 parametrov je typu uint. Tento typ som použil pri funkciách na vyhľadávanie objektov v obraze, kde uchar parameter bol hodnota prahu, a uint parametre boli hranicami vyhľadávaných oblasti v obraze. Pri posledných dvoch funkciách sa po stlačení tlačidla Start zobrazí okno, kde je možné zadať jednotlivé parametre. Aplikácia zachová ako predvolené parametre posledné používane, teda nie je potrebné pri každom spustení funkcie parametre nanovo zadávať. Pokiaľ je návratová hodnota Textový súbor, sú ponúknuté možnosti: *(*unsigned char vstupný obraz, *char cesta výstupného súboru, uint šírka, uint výška)* a *(*unsigned char vstupný obraz 1, *unsigned char vstupný obraz 2, *char cesta výstupného súboru, uint šírka, uint výška).* Vybraná funkcia s parametrami sa pridá do aplikácie stlačením tlačidla Pridaj. Pokiaľ chceme danú funkciu odobrať z programu, označíme ju kliknutím na ňu v okne s aktívnymi funkciami a stlačením tlačidla Vymaž funkciu. Aby sa zmeny prejavili, je potrebné ukončiť aplikáciu a po jej opätovnom spustení sa medzi ponukami na funkcie zobrazia pridané funkcie, respektíve, odstránené funkcie sa nezobrazia.

7. ZÁVER

Cieľom tejto diplomovej práce bolo otestovať použitie SSE inštrukcií na funkcie pre spracovanie obrazu a ukázať na praktickej aplikácii ich výhody.

Vybranou úlohou bolo detekovanie počtu svietiacich LED na LED paneli výkonového zdroja. Túto úlohu som skúšal riešiť dvomi metódami. Obidve metódy som vytvoril pomocou klasických funkcií s použitím jazyka C. Následne som tieto metódy naprogramoval s využitím algoritmov s SSE inštrukciami. Cieľom bolo porovnávať čas na spracovanie týchto funkcií a tým ukázať hlavnú výhodu SSE inštrukcií, jako aj zrýchlenie výpočtových algoritmov. Algoritmy som testoval na 34 testovacích snímkach, ktoré pokrývali všetky režimy a boli vyhotovené pri rôznych podmienkach.

Prvou metódou bolo počítanie svietiacich LED pomocou spočítania celkového počtu pixelov a vydelením priemerného počtu pixelov na jednu svietiacu LED. Táto metóda bola rýchla, avšak dosahovala iba 50% úspešnosť správnej detekcie. Dôvodom bolo to, že snímky boli nasnímané pri rôznych svetelných podmienkach, kde veľkosť plochy svietiacich LED nebola u všetkých LED rovnaká. Táto metóda by sa dala použiť pri menšom počte detekovaných LED, kde by sa eliminovala chyba spôsobená rôznymi veľkosťami svietiacich objektov.

Funkcia vytvorená pomocou klasických funkcií mala priemerný čas spracovania 2,226 ms. Optimalizovaná funkcia s SSE inštrukciami mala priemerný čas spracovania 0,5132ms. Z týchto výsledkov vidno, že optimalizovaná funkcia s SSE inštrukciami dosahovala asi štyrikrát kratší čas spracovania ako neoptimalizovaná funkcia.

Druhou metódou bolo určenie počtu svietiacich LED pomocou označenia svietiacich objektov maskou štvorokolia. Uvedená metóda bola pomalšia ako predchádzajúca, ale dosiahla 100% úspešnosť správnej detekcie počtu svietiacich objektov. Táto metóda zabezpečuje vysokú spoľahlivosť pri použití v tejto praktickej aplikácii.

Primerný čas spracovania funkcie vytvorenej klasickými algoritmami bol nameraný 124,84ms. Funkcia optimalizovaná pomocou SSE inštrukcií mala

priemerný čas spracovania 58,99 ms. Z nameraných údajov vidno, že aj v tomto prípade sa prejavila rýchlosť spracovania pomocou SSE inštrukcií, kde optimalizovaná funkcia bola dvakrát rýchlejšia.

Ďalšou úlohou bolo porovnanie rýchlosti funkcií kompilovaných rôznymi kompilátormi. Na porovnanie som vybral dve funkcie Prahovanie a LedTestPriemer. Porovnával som kompilátory gcc a lcc. Pri prvej funkcii Prahovanie, skompilovanej pomocou kompilátora gcc, som nameral priemerný čas spracovania 1,98ms a pri tej istej funkcii skompilovanej pomocou kompilátora lcc, som nameral čas spracovania 2,31ms. Pri druhej funkcií LedTestPriemer som nameral čas spracovania 2,226 ms na funkcii kompilovanej pomocou kompilátora gcc a čas spracovania 2,449 na funkcii kompilovanej pomocou kompilátora lcc. Z obidvoch porovnávaní je zrejmé, že kompilátor gcc je rýchlejší ako kompilátor lcc. Vo všetkých funkciách používaných v tejto práci som použil kompilátor gcc.

Pri tejto práci som vytvoril ďalšie 3 funkcie pre spracovanie obrazu. Sú to funkcie na súčet obrazov, inverziu obrazu a na detekciu hrán Sobelovým operátorom. Cieľom bolo porovnať rýchlosť spracovania obrazu funkcií s SSE inštrukciami voči klasickým. Pracoval som s dvomi obrazmi 320 na 300 pixelov. Pri súčte dvoch obrazov bez použitia SSE inštrukcií som nameral čas 443 us. Pri rovnakej operácii s použitím SSE inštrukcií som nameral čas 219 us. Pri inverzii obrazu bez použitia SSE som nameral čas 309 us. Pri súčte obrazu s SSE inštrukciami som nameral čas 159 us. Pri poslednej funkcii detekcia hrán maskou so Sobelovymi operatoromi bez použitia SSE inštrukcií, som nameral čas 14,374 ms. Pri rovnakej operácii s použitím SSE inštrukcií, som nameral čas 4,942 ms. Z uvedených meraní je zrejmé, že použitie SSE inštrukcií môže výrazne skrátiť čas výpočtu. Najväčšia úspora času bola pri detekcii hrán, kde použitie SSE inštrukcií zrýchlilo výpočet trikrát. Táto úspora času určite vynahradí relatívnu zložitosť kódu s SSE.

Ďalším cieľom projektu bolo vytvoriť užívateľské grafické rozhranie umožňujúce prácu s SSE funkciami a zobrazovanie výsledku. Aplikáciu som vytvoril pre prostredie OS Windows pomocou knižnice wxWidgets. Aplikácia umožňuje načítavať obrazové súbory, zobrazovať výsledky a počíta časy jednotlivých funkcií. Ďalej umožňuje dynamicky pridávať funkcie v dll súboroch bez nutnosti ďalšej

kompilácie. To umožňuje vyvíjať ďalšie funkcie s použitím SSE inštrukcií a ich rýchle otestovanie.

V tejto diplomovej práci som dokázal, že použitie algoritmov s SSE inštrukciami výrazne skracuje dĺžku výpočtu rôznych funkcií na spracovanie obrazu. Získaný čas má veľký prínos na praktické použitie počítačového videnia. Zrýchlenie výpočtu umožňuje zvýšiť produktivitu vo výrobných procesoch, ktoré maju aplikácie založené na počítačovom videní.

8. LITERATÚRA

[1]. Zpracování obrazu a algoritmy c C# , Michal Dobeš

[2]. Moderní počítačová grafika, Jiří Žára, Bedřich Beneš, Jiří Sochr, Petr Felkel

[3]. Cross-Platform GUI Programming with wxWidgets, Julian Smart, Kevin Hock, Stefan Csomor.

[4]. Open Computer Vision Library, http://sourceforge.net/projects/opencvlibrary/

[5]. Počítačové vidění, Milan Šonka, Václav Hlaváč, Grada 1992.

[6]. Time stamp counter, http://www.mcs.anl.gov/~kazutomo/rdtsc.html

[7]. Počítačové vidění, Ing.Karel Horák,Ph.D, Ing.Ilona Kalová,Ph.D, Ing.Petr Petyovský, Ing.Miloslav Richter,Ph.D.

[8]. Intel, http://www.intel.com/

[9]. Reference SSE , http://www.rz.uni-karlsruhe.de/rz/docs/VTune/reference/

[10].WxWidgets , http://www.wxwidgets.org/

[11]. Dynamic-Link Function,
http://msdn.microsoft.com/en-us/library/ms682599(VS.85).aspx