SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 752

OPTIMIRANJE ALGORITAMA OBRADE SLIKE VEKTORSKIM INSTRUKCIJAMA

Ozana Živković

Zagreb, lipanj 2009.

Sadržaj

1. Uvod

Obrada slike je područje u računarstvu koje se može smatrati veoma specifičnim u pogledu obrade podataka. Razlog tome jest spremanje informacija o slici u formatu koji svaki slikovni element reprezentira određenim vrijednostima, najčešće vrijednostima (intenzitetu) boja u tom slikovnom elementu. U obradi slike, svaki se slikovni element razmatra kao dio veće cjeline. To zahtjeva obradu svih slikovnih elemenata na nekom području, a ne pojedinačnog elementa. Takva se obrada slike sastoji od velikog broja operacija koje su međusobno veoma slične, pa i jednake. Upravo to zahtjeva veliku brzinu izvođenja jednostavnih operacija.

Ovakvi i slični problemi pojavljivali su se i u drugim područjima obrade informacija - puno podataka koje je potrebno obraditi na sličan način. Većina takvih problema može se jednostavno riješiti paralelnom obradom podataka. No, moderni procesori opće namjene zasnovani su na Von Neumannovom principu i obrađuju jedan podatak u jednoj jedinici vremena. Djelomično je razlog tome nemogućnost danjeg razvoja tehnologije u smjeru brzine i kreće se sa razvojem paralelne obrade podataka. Sve je to uvjetovalo početak razvoja vektorskih proširenja instrukcijskih arhitektura modernih procesora opće namjene.

Jedno od područja gdje su vektorske instrukcije ubrzale obradu podataka je upravo obrada slike. Njima je omogućeno procesiranje većeg broja elemenata slike jednom instrukcijom, što uvelike ubrzava ukupnu obradu. Često je riječ o situaciji kada se nad različitim elementima slike obavlja ista operacija i u tom slučaju postiže se drastično ubrzanje. Upravo su zato algoritmi zasnovani na vektorskim instrukcijama važna karika u modernoj obradi slike i zbog toga će biti razmatrani u ovom radu.

Rad je strukturiran kako slijedi. U poglavlju ***2*** analizirati ćemo koncepte relevantne za promatrano područje. Upoznati ćemo se sa teorijskim osnovama primijenjenih algoritama. Isto tako, analizirat ćemo tehnološku strana problema, odnosno korištena vektorska proširenja modernih procesora. Glavni dio rada jest implementacija dvaju algoritama obrade slike i njihova optimizacija vektorskim instrukcijama. U poglavlju ***4*** objasnit ćemo programsku realizaciju algoritma oduzimanja pozadine i analizirati

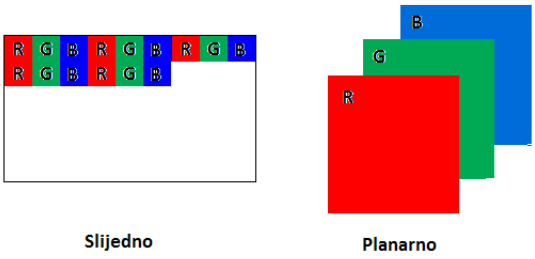
dobiveno ubrzanje. Implementacija algoritma konvolucije slike kao i njezina moguća poboljšanja bit će prikazana u poglavlju ***5***. Na kraju rada navest ćemo moguće, buduće implementacije algoritma konvolucije slike u cilju još većeg ubrzanja.

1.1. Obrada slike

Obrada slike najčešće podrazumijeva višestruko ponavljanje iste operacije nad svakim elementom odredišne slike. U pravilu, slike se sastoje od iznimno velikog broja slikovnih elemenata (eng. pixel). Često razmatramo tzv. sivu sliku čiji elementi predstavljaju razinu svjetline u odgovarajućem dijelu slike. Način spremanja sive slike je često u obliku dvodimenzionalne matrice elemenata veličine jednog okteta. Elementi su veličine jednog okteta iz razloga jer su vrijednost svakog elemenata u rasponu od 0 do 255.

Postoji niz različitih algoritama za obradu slike. Svi ti algoritmi određuju izlaznu sliku na temelju ulazne slike. Neki izračunavaju vrijednost elemenata izlazne slike na temelju odgovarajućeg elementa ulazne slike. Primjer takvog algoritma je algoritam oduzimanja pozadine. Drugi algoritmi izlazne slikovne elemente određuju u ovisnosti o susjednim slikovnim elementima odgovarajućeg elementa ulazne slike. Kao primjer takvog algoritma možemo navesti algoritam konvolucije slike.

Slika može biti zapisana u raznim formatima. Primjena pojedinog algoritma može izričito zahtijevati određeni format. Tako se algoritam konvolucije slike može izvoditi nad slikom zapisanom u RGB i u YUV formatu. Naime, svaki slikovni element sastoji se od tri komponente R, G i B. Radi se o crvenoj, zelenoj i plavoj boji, koje kombinacijom različitih omjera daju ostale boje. Kod RGB zapisa slike razlikuju se dva načina pohrane slikovnih elemenata u memoriju. Prilikom slijednog zapisa elementi slike spremaju se u memoriju u slijedećem obliku: „…RGBRGBRGBRGBRGBRGBRGBRGBRGB…“. Osim ovog formata slika može biti i planarno zapisana. Navedeni formati prikazani su na slici 2.1.



Slika 2.1. Slijedno i planarno pohranjivanje elemenata slike u memoriju

Algoritam mijenjanja svjetline može se izvoditi na slici u YUV formatu. YUV (YCrCb) prostor sastoji se od tri komponente: Y, U i V. Sama Y komponenta predstavlja svjetlinu. Dok su U i V kromatske komponente, odnosno boja. Razlog dijeljenja slike na dva dijela leži u tome što ljudsko oko posebno vidi svjetlinu, a posebno boju. Razlike u svjetlini lakše se zapažaju od razlika u boji, pa se može napraviti da se manje informacija zapisuje u boji, a više u svjetlini.

1.2. Opis algoritma oduzimanja pozadine

Na ulazu algoritma javljaju se dvije slike. Prvi okvir slike uzima se kao pozadina, dok drugi okvir predstavlja trenutnu sliku. Za detekciju pokreta potrebno je izračunati razliku vrijednosti slikovnih elemenata pozadine i trenutne slike. Za dalju izvedbu algoritma uzima se apsolutna vrijednost dobivene razlike. Zadaje se vrijednost praga, na temelju koje algoritam ispituje da li je došlo do pomaka. Prolazi se kroz sve slikovne elemente, a pomak se detektira ukoliko je apsolutna razlika između slikovnih elemenata pozadine i trenutne slike veća od zadane vrijednosti praga. U slučaju detektiranja pomaka za pojedini slikovni element, njegova vrijednost postavlja se na 255 i time postaje bijele boje. Ostali slikovni elementi kod kojih nije došlo do pomaka postavljaju se na vrijednost 0, odnosno postaju crne boje. Opisanim postupkom dobiva se binarna slika, na kojoj su promijenjeni elementi prikazani bijelom bojom dok su ostali elementi crne boje. Rezultati ovog algoritma prikazani su u odjeljku ***4.2.1***, a opisani temeljni princip rada može se vidjeti niže :

```
if (fabs(P[i,j] -  S[i,j]) > prag )  {
      P[i,j] = 255;
  } else {
      P[i,j] = 0;
}
```

Oduzimanje pozadine posebno je prikladno za nadzor prostora. Međutim, postoje situacije u kojima algoritam neće dati ispravne rezultate. Neki od razloga pogrešne detekcije navedeni su niže:

- postojanje titrajućih objekata u pozadini (kao što su monitor, drveće i sl.)
- promjene osvjetljenja
- sjene i refleksije (objekti mogu bacati sjene oko sebe koje se mogu pogrešno detektirati kao pokreti)
- prikrivanje (elementi promjene u trenutnoj slici mogu imati jednake vrijednosti intenziteta ili boje kao i pozadina, te se zbog toga ne mogu detektirati)

Da bi se riješili ti problemi potrebno je na određeni način modelirati pozadinu [15].

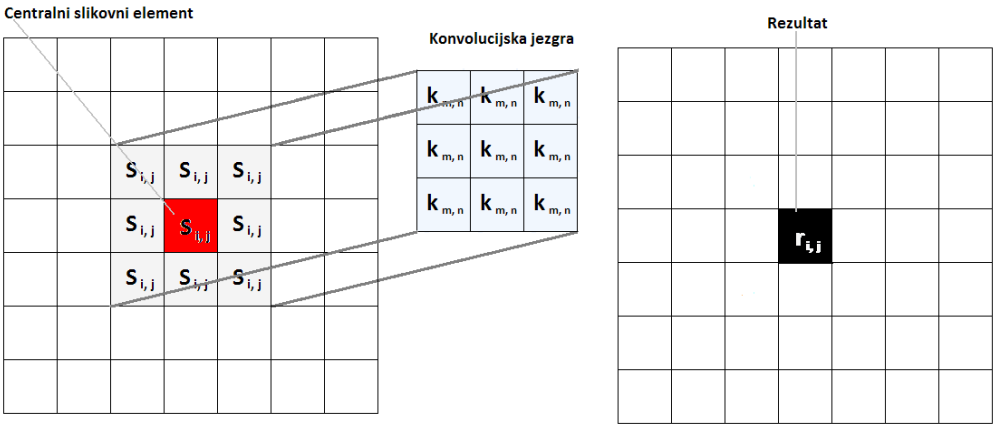
1.3. Konvolucija slike

Konvolucija slike je jedna od najvažnijih operacija u području obrade slike. Osnovna ideja dvodimenzionalne konvolucije slike je da se elementi izlazne slike uzimaju kao linearne kombinacije elemenata ulazne slike. To se naziva linearno filtriranje.

$$r[i,j] = s[i,j] * k[m,n] = \sum_m \sum_n s[i-m, j-n] k[m,n] \tag{2.2}$$

Vrijednost slikovnog elementa koji se u pojedinom trenutku smatra centralnim slikovnim elementom zajedno sa vrijednostima susjednih slikovnih elemenata, množi se

sa vrijednostima konvolucijske jezgre (eng. kernel). Zapravo radi se o matrici koja se „prevlači“ preko elemenata slike. Svaki element slike ispod matrice množi se sa odgovarajućom vrijednosti unutar matrice. Spomenuto množenje obavlja se po jednadžbi (2.2) i sumiraju se dobiveni rezultati. Trenutna vrijednost centralnog slikovnog elementa odredišne slike zamjenjuje se krajnjim rezultatom. Opisani postupak može se vidjeti na slici 2.3.



Slika 2.3. Dvodimenzionalna konvolucija

Dvodimenzionalna konvolucijska jezgra (filtar) širine *m* i visine *n* za svaki izlazni slikovni element zahtjeva ukupno *n·m* operacija množenja. Ako je korišteni filtar separabilan, postupak se uvelike ubrzava [6, 13].

1.3.1. Separabilni dvodimenzionalni filtri

Separabilni dvodimenzionalni filtri su posebna vrsta filtara, koji se mogu izraziti kao kompozicija dva jednodimenzionalna filtra.

U primjeru (2.4) pokazano je kako se konvolucijska jezgra može napisati kao produkt dva jednodimenzionalna vektora.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 \\ 2 \\ 1 \end{bmatrix} * \begin{bmatrix} -1 & 0 & 1 \end{bmatrix} \tag{2.4}$$

Može se pokazati da je konvolucija asocijativna, što znači da vrijedi:

$$f * (r * s) = (f * r) * s \tag{2.5}$$

To znači da će se isti rezultat dobit ako konvoluciju nad slikom (*f*) obavimo dva puta, jednom retčanim vektorom (*r*), a drugi put stupčanim vektorom (*s*) koristeći prethodno dobiveni rezultat *(f*r)* kao ulaz.

Na taj način, nad slikom obavljaju se dvije jednodimenzionalne konvolucije, jedna nad recima slike, a druga nad stupcima slike. Ovim načinom vrijeme obrade slike se znatno skraćuje. Za svaki izlazni element slike dimenzije *PxQ* prolaz po recima zahtijeva se oko *p·q·n* operacija množenja, gdje je *n* visina konvolucijske jezgre. Drugi prolaz po stupcima zahtijeva oko *p·q·m* operacija množenja (*m* je širina konvolucijske jezgre). Ukupno oko *p·q·(m+n)* operacija, dok je kod konvolucije istom dvodimenzionalnom konvolucijskom jezgrom za sliku istih dimenzija *PxQ* bilo potrebno oko *m·n·p·q* operacija. Omjer konvolucije dvodimenzionalnom konvolucijskom jezgrom i konvolucije separabilnim filtrom je sljedeći:

$$\frac{m \cdot n}{m + n} \tag{2.6}$$

Prema jednadžbi (2.6) može se izračunati kako korištenje separabilnih filtara ubrzava izvođenje 3.5 puta za konvolucijsku jezgru dimenzija *7x7*.

U nastavku rada razmatrat će se konvolucija sa konvolucijskim jezgrama kao kvadratnim matricama. Dimenzije jezgre bit će neparni brojevi radi jednoznačnog određivanja centralnog slikovnog elementa.

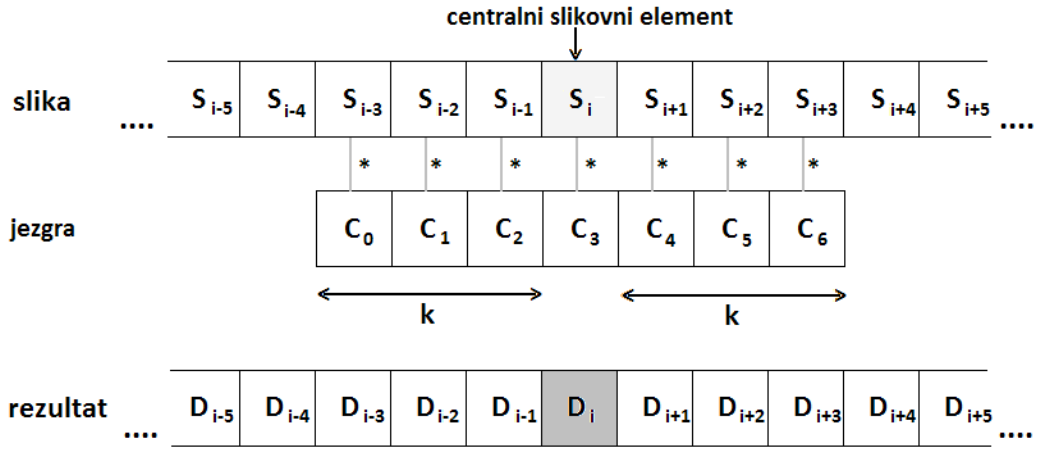
1.3.2. Konvolucija slike 1D konvolucijskom jezgrom

Konvolucija s retčanom ili stupčanom jezgrom se obavlja zasebno nad svakim retkom, odnosno stupcem slike. Svaki redak i stupac slike promatra se kao jednodimenzionalno polje (vidi sliku 2.8.). Isto tako, kao jednodimenzionalno polje ali sa neparnim brojem

elemenata, zadaje se i konvolucijska jezgra (eng. *kernel*). Za pojedini slikovni element, koji se u danom trenutku smatra centralnim u obzir se uzimaju i njegovi susjedni elementi. Znači, od centralnog slikovnog elementa na lijevu i desnu stranu u obzir se uzima broj elemenata jednak polovini duljine konvolucijske jezgre (na slici 2.8. prikazano kao ***k***). U idućem koraku ti elementi množe se sa vrijednostima odgovarajućih pozicija unutar jezgre. Dobiveni umnošci se sumiraju i zapisuju na poziciju centralnog slikovnog elementa odredišne slike prema formuli (2.7).

$$D[i] = \sum_{j=0}^{2k+1} s[i-k+j] * c[j] \tag{2.7}$$

Opisana zamjena provodi se za sve slikovne elemente unutar jednodimenzionalnog polja. Izuzetak su elementi na rubovima. Radi lakše implementacije, opisani postupak ne odnosi se na broj elemenata jednak (duljina_jezgre)/2 koji se nalaze na početku i samom kraju polja.



Slika 2.8. Jednodimenzionalna konvolucija

Za svaki slikovni element izvršit će se $2 \cdot k + 1$ operacija, gdje je ***k*** jednak polovini duljine zadane jezgre.

1.4. Vektorski instrukcijski podskupovi za arhitekturu x86

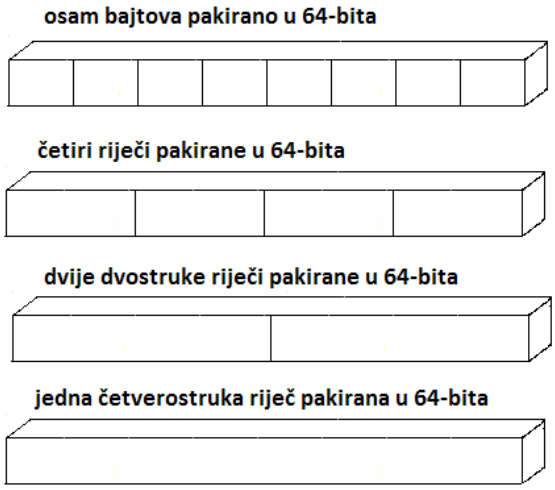
U ovom poglavlju kronološki se prikazuju proširenja instrukcijskog skupa arhitekture x86. Za svako proširenje navode se procesori u kojima se prvi puta pojavljuju. Isto tako, prikazuju se novi tipovi podataka koje uvode pojedina proširenja i ostale nove osobine koje omogućuju.

1.4.1. MMX

Glavna namjena korištenja proširenja MMX (eng. *MultiMedia eXtension)* je izvođenje vektorskih operacija nad cjelobrojnim tipovima podataka (eng. *integer*). Uvodi se novi tip podataka, 64-bitni vektor sa cjelobrojnim vrijednostima. Daje zadovoljavajuće ubrzanje i ne traži dodatne velike zahtjeve na arhitekturi. Razlikuju se četiri tipa MMX podataka [2].

- 64-bitni pakirani cjelobrojni bajt (eng. *64 bitni packed byte integer*)
  - sadrži osam 8-bitnih cjelobrojnih podataka
- 64-bitna pakirana cjelobrojna riječ (eng*. 64 bitni packed word integer*)
  - sadrži četiri 16-bitna cjelobrojna podatka
- 64-bitna pakirana cjelobrojna dvostruka riječ (eng. *64 bitni packed doubleword integer*)
  - sadrži dva 32-bitna cjelobrojna podatka
- 64-bitna pakirana cjelobrojna četverostruka riječ (eng. *64 bitni packed quadword integer*)
  - sadrži jedan 64-bitni cjelobrojni podatak

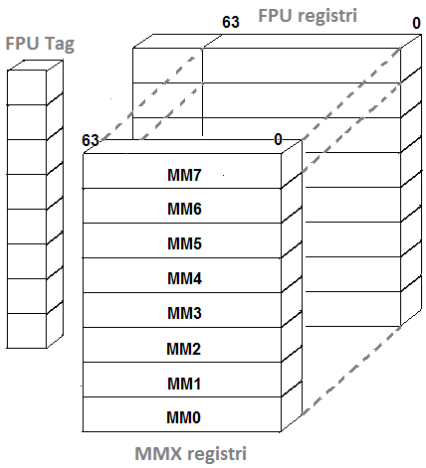
Kao što je prikazano na slici 2.9., 8-bitni, 16-bitni, 32-bitni i 64-bitni cjelobrojni podaci spremaju se u 64-bitne pakirane podatke. Prilikom obrade slike algoritmom oduzimanja pozadine koristit će se pakirani podatak s osam 8-bitnih cjelobrojnih vrijednosti.



Slika 2.9. MMX tipovi podataka

Proširenje MMX uvode u arhitekturu osam novih registara (od MM0 do MM7). Kako su registri MMX duljine 64 bita, ne mogu se pohraniti u registre opće namjene, duljine 32 bita. Umjesto toga, instrukcijska proširenja MMX za obavljanje matematičkih operacija koriste osam registara (FP) specijalne namjene za podatke sa pomičnim zarezom. Radi se o registrima od R0 do R7 (općenito Rx), svaki je duljine 80 bita i služi za pohranu podataka dvostruke preciznosti u *IEEE floating-point* standardu. Registri nisu dostupni direktno, već kao LIFO stog. Registri MMX izravno se preslikavaju na registre FP kako je prikazano na slici 2.10. Za razliku od registara Rx , registri MMX su fiksni, ne mogu se koristiti kao stog. Registar MM0 uvijek će se referencirati na FP registar R0. Stavljanjem novog podatka u registar MM0, ne dolazi do guranja prethodno pohranjenog podataka iz registra MM0 u registar MM1, već se jednostavno preko postojećeg podatka prepisuje novi. Kao što je već navedeno svaki od registara FP stoga duljine je 80 bita, te će prilikom MMX načina rada gornjih 16 bita biti neiskorišteno. Izvođenjem instrukcije MMX, registar FP Tag postaje okupiran. Ukoliko se u registrima FP nalaze podaci, potrebno je prije prelaska u MMX način rada spremiti stanje registara FP. Spremanjem stanja zapravo se odvajaju instrukcije nad podacima sa pomičnim zarezom u registrima FP od instrukcija MMX. Koristi se instrukcija ***FSAVE*** koja pohranjuje kompletno FPU okruženje zajedno sa podacima, odnosno kopira sve registre FP sa stoga u memoriju. Nakon što se izvede instrukcija MMX, izvođenjem instrukcije ***FRSTOR*** vraćaju se registri FP zajedno sa podacima u stanje koje je prethodilo spremaju, odnosno instrukciji ***FSAVE***. Kako bi se nastavilo s ispravnim

izvođenjem instrukcija nad podacima sa pomičnim zarezom, potrebno je nakon završetka rada s registrima MMX koristiti instrukciju ***EMMS***. Ona će pobrisati vrijednosti u FPU Tag registru upisane početkom izvođenja instrukcije MMX. Primjer njenog korištenja biti će prikazan prilikom implementacije algoritma oduzimanja pozadine instrukcijama MMX.



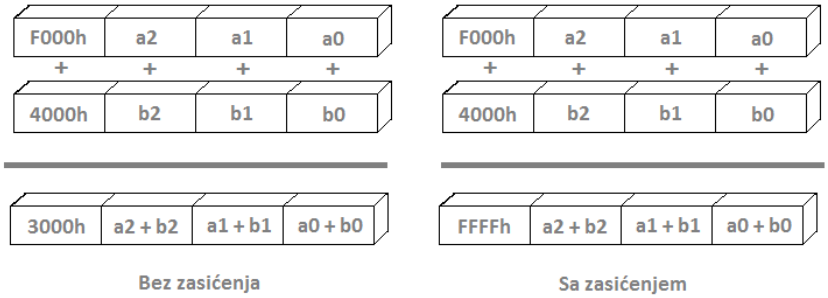
Slika 2.10. Izravno preslikavanje MMX registara u FP registre

Proširenje MMX uvodi novih 57 instrukcija. Instrukcije pokrivaju različita funkcionalna područja i prema načinu primjene dijele se u osam skupina:

- Logičke operacije
- Aritmetičke operacije
- Operacije usporedbe
- Operacije pretvorbe
- Operacije pomaka
- Prijenos podataka
- Upravljanje stanjem (***EMMS***)

Može se koristiti tzv. aritmetika sa zasićenjem (sa i bez predznaka) koja može značajno poboljšati performanse. U odjeljku ***4.1.1*** bit će prikazana jedna takva instrukcija za oduzimanje podataka bez predznaka. Isto tako može se koristiti i aritmetika bez zasićenja (vidi sliku 2.11.). U načinu rada bez zasićenja, ukoliko se rezultat operacije prelijeva iznad ili ispod graničnih vrijednosti kao krajnji rezultat uzimaju se samo najmanje značajni

bitovi, dok se preljev ili podljev ignorira. Kod zasićenja ako se dogodi preljev rezultat se postavlja na najveću dopuštenu vrijednost, a u slučaju podljeva na najmanju dopuštenu.

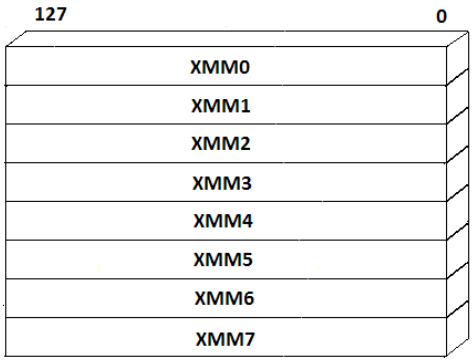


Slika 2.11. Zbrajanje bez zasićenja (lijevo) i sa zasićenjem (desno)

Korištenje instrukcija je vrlo jednostavno. Potrebno je pripremiti podatke, odnosno pretvoriti ih u željeni tip (eng. *packed byte*, *word*, *doubleword* ili *quadword*). Sljedeći korak je spremanje podataka u registre MMX. Zatim slijedi usporedno pribavljanje i izvođenje željene matematičke operacije, te usporedna pohrana rezultata na zadanu memorijsku lokaciju.

1.4.2. SSE

Uvođenje instrukcija MMX omogućilo je ubrzanje obrade cjelobrojnih podataka. Međutim za programe koji izvode operacije nad podacima sa pomičnim zarezom i dalje nema nikakvih promjena u brzini obrade. Ovaj problem rješavaju ekstenzije SSE (eng. *Streaming SIMD Extensions*) [2]. Radi se o generaciji SIMD proširenja implementiranih u procesor Pentium III. Omogućuju izvođenje vektorskih operacija nad podacima sa pomičnim zarezom i jednostrukom preciznošću. Uvodi se osam novih 128-bitnih registara (od XMM0 do XMM7), prikazanih na slici 2.12.



Slika 2.12. SSE registri

Uz ove registre uvodi se i dodatni 32-bitni kontrolni registar MXCSR, koji kontrolira stanja instrukcija SSE. Registri XMM ne referenciraju registre jedinice FPU, kao što je to slučaj kod proširenja MMX. Koriste se potpuno odvojeni 128-bitni registri. To omogućuje aplikacijama istovremeno izvođenje instrukcija iz podskupova SSE i FPU, odnosno ne-SIMD operacija nad podacima sa pomičnim zarezom. Osim dosad navedenih novosti, programeru se na korištenje pruža i preko 70 novih instrukcija za rad nad podacima sa pomičnim zarezom. U odnosu na instrukcije MMX, koje istovremeno obrađuju samo dva 32-bitna cjelobrojna podataka, ove instrukcije mogu istovremeno obraditi četiri 32-bitna ili dva 64-bitna podatka sa pomičnim zarezom. Iduća razlika je što sve instrukcije za aritmetičke operacije dolaze u dvije verzije. Instrukcije sa sufiksom PS izvode operacije slično kao i instrukcije MMX. Na primjer, ako se unutr 128-bitnog registra nalaze četiri 32-bitne vrijednosti, aritmetička operacija izvodit će se posebno za svaku 32-bitnu vrijednost. Isto tako, rezultat se pohranjuje kao 32-bitni podatak na odgovarajuću poziciju unutr 128-bitnog vektora. Druga verzija instrukcija SSE sa sufiksom SS ne izvode operacije nad svim 32-bitnim podacima unutar 128-bitnog vektora. Aritmetička operacija izvodi se samo na najniža 32-bita (*doubleword*). Rezultat operacije također se zapisuje na najniža 32-bita, dok se ostale tri 32-bitne vrijednosti prepisuju iz izvornog operanda. Potrebno je spomenuti kako se razlikuju instrukcije SSE za pristup podacima u memoriji. Ukoliko se radi o podacima poravnatim u memoriji na 16 bajtne lokacije, procesor će brže čitati podatke za pojedine operacije. Zbog toga je poravnanje podataka unutar memorije još jedan od uvjeta za što bržom obradom.

1.4.3. SSE2

Vektorska proširenje po prvi puta se pojavljuju u Intelovom procesoru Pentium IV. Nove instrukcije su po strukturi vrlo slične instrukcijama SSE. Glavna razlika, istovremeno i prednost novih instrukcija je to što omogućuju izvođenje instrukcija iz skupa SSE nad cjelobrojnim podacima. Uvodi se nekoliko novih tipova podataka [2]:

- 128-bitni pakirani podatak dvostruke preciznosti sa pomičnim zarezom (eng. 128-bit packed double-precision floating-point)
- 128-bitni pakirani cjelobrojni bajt (eng. 128-bit packed byte integers)
  - sadrži šesnaest 8-bitnih cjelobrojnih podataka
- 128-bitna pakirana cjelobrojna riječ (eng. 128-bit packed word integers)
  - sadrži osam 16-bitna cjelobrojnih podataka
- 128-bitna pakirana cjelobrojna dvostruka riječ (eng. 128-bit packed doubleword integers)
  - sadrži četiri 32-bitna cjelobrojna podatka
- 128-bitna pakirana cjelobrojna četverostruka riječ (eng. 128-bit packed quadword integers)
  - sadrži dva 64-bitna cjelobrojna podatka

Izvođenje instrukcija nad navedenim tipovima podataka nije moguće u registrima iz skupa MMX. Za ove tipove podataka koriste se registri XMM, a instrukcije mogu se izvoditi samo na procesorima koji podržavaju SSE2. Slično kako i kod SSE, svaki tip podatka ima svoju instrukciju za pojedinu matematičku operaciju. Koristeći ovu tehnologiju korisnik ima na raspolaganju ukupno 144 nove instrukcije. Moguće je izvoditi operacije nad podacima poravnatim u memoriji na 16 bajtne lokacije ili koristiti drugi, sporiji način kao što je pristupanje neporavnatoj memoriji. U odlomku o instrukcijama SSE objašnjeno je kako su registri XMM potpuno neovisni o FPU. Zbog toga aplikacije mogu istovremeno izvoditi instrukcije MMX i SSE2. Isto tako moguće je istovremeno izvoditi ne-SIMD operacije nad podacima sa pomičnim zarezom (instrukcije FPU) i instrukcije SSE2.

1.4.4. SSE3

Pentium IV je bio prvi procesor koji je podržavao SSE3 (eng. *Streaming SIMD Extension* 3), radi se o dodatnom proširenu instrukcija SSE2. Ove ekstenzije ne uvode nikakve nove tipove podataka kao što je to bio slučaj u prethodnim inačicama, već se dodaju samo nove instrukcije za još bržu obradu multimedijskih podataka. Pojavljuje oko 13 novih instrukcija koje donose nove prednosti na području [2]:

- Konverzije podataka s pomičnim zarezom u cjelobrojne vrijednosti
- Složene aritmetičke operacije (npr. horizontalne operacije)
- Video kodiranje
- Grafika
- Sinkronizacija dretvi

Većina ovih instrukcija koriste dobro znane registre XMM, izuzetak su instrukcije za sinkronizaciju dretvi i instrukcija za pretvorbu u cjelobrojni tip podatka. Novost kod aritmetičkih instrukcija su horizontalne operacije koje za izvođenje mogu koriste samo jedan registar. Jedna takva instrukcija je ***haddps***, pobliže objašnjena u odjeljku ***5.1.1.***

2. Korišteni razvojni alati

Za izvedbu algoritma korišteno je programsko okruženje *cvsh* (eng. *computer vision shell*), razvijeno na Zavodu za elektroniku, mikroelektroniku, računalne i inteligentne sustave (ZEMRIS). Radi se o ljusci sličnoj korisničkoj ljusci operacijskih sustava UNIX. Ona omogućuje relativno jednostavno eksperimentiranje algoritmima računalnog vida. Zamišljena je tako da se korisnički postupci prevedu zajedno s ljuskom u jednu izvršnu datoteku. Također je vrlo jednostavno dodavanje novih algoritama, jer se komponente napisane u skladu sa konvencijama automatski povezuju s ostatkom aplikacije. Razvijene su paralelne komponente za operacijske sustave Windows i Unix. Komponente za operacijski sustav Windows imaju sufiks _w32, a komponente za Unix _unix. Ljuska zahtjeva vanjsku biblioteku *boost*. Za pribavljanje slike iz datoteka koje koriste noviji algoritam kompresije potrebno je koristiti Windows Media Format SDK.

U ovom radu, za pisanje i prevođenje koda korišteno je okruženje Eclipse IDE. Kao što će biti prikazano u nastavku, cvsh se prevodi pomoću prevoditelja GCC (eng. *GNU Compiler Collection)*. U cilju kontrole vremena prevođenja ovaj prevoditelj pruža nekoliko razina optimizacija i individualne opcije za određene vrste optimizacije. Razlikuju se tri općenite razine optimizacije (O1, O2 i O3).

Unutar koda pisanog u programskom jeziku C++ mogu se umetnuti dijelovi asemblerskog koda (eng. *Inline assembler*). Ovakav način umetanja koda pisanog u nižem programskom jeziku omogućit će nam korištenje instrukcija MMX i SSE. Prevoditelji iz porodice GCC koriste ključnu riječ ***asm*** za označavanje dijelova koda pisanih u asemblerskom jeziku. Sintaksa tih dijelova je sljedeća:

```
__asm__ volatile ("instrukcija"
                      : izlazna_vrijednost
                      : ulazna_vrijednost
                      : korišteni_registri
    );
```

Nakon ključne riječi ***asm*** koja označava da slijedi dio koda pisan u asembleru, navodi se modifikator ***volatile***. Njime se daje uputa prevoditelju *GCC* da taj dio koda napisan u asembleru ne optimizira.

Za referenciranje memorijskih lokacija ulaznih i izlaznih vrijednosti koristi se konstanta ***m***. Ona omogućava izravno korištenje memorijskih lokacija ili varijabli programskog jezika C ili C++. Ispred konstante može se dodati modifikator kojim se daje uputa prevoditelju kako se koristi pojedine vrijednosti. Modifikator može biti:

- **+** (***"+m"***) - operand se može čitati i/ili spremati
- **=** (***"=m")*** - operand se može samo spremati
- (***"m"***) - operand se može samo čitati

3. Primjena vektorskih instrukcija u oduzimanju pozadine

3.1. Programska izvedba

Kao što govori naslov, u ovom poglavlju, bit će prikazana programska izvedba algoritma oduzimanja pozadine. U odjeljku ***4.1.1*** opisano je optimiranje algoritma instrukcijama MMX. Na samom kraju prikazana je programska izvedba algoritma bez korištenja vektorskih instrukcija. Na temelju nje, provest će se analiza rezultata dobivenog instrukcijama MMX.

U odjeljku ***4.1.2*** prikazana je integracija algoritma oduzimanja pozadine u programsko okruženje opisano ranije. Prikazan je način na koji se dohvaća niz slika i kako se pokreće algoritam.

3.1.1. Implementacija algoritma oduzimanja pozadine

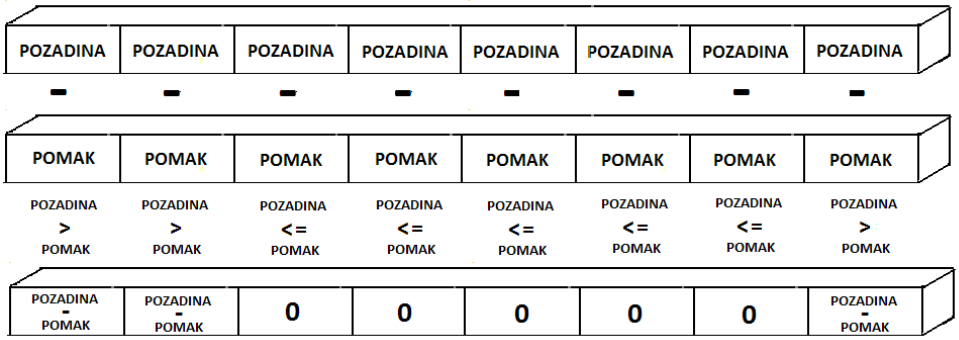
Elementi ulazne i izlazne slike su duljine jednog bajta, te će se prilikom obrade slike kao tip podatka koristiti nepredznačeni podatak (eng. unsigned char) raspona vrijednosti od 0 do 255. Na samom početku ovog rada prikazani su 64-bitni registri MMX i svi načini na koje se mogu popuniti podacima ovisno o duljini podatka. Kako su slikovni elementi duljine 8 bita unutar jednog 64-bitnog registra MMX (eng. 64 bitni packed byte integer) spremat će se osam slikovnih elemenata. Zadani prag potrebno je također spremiti u neki od registara MMX, kako bi kasnije bilo moguće istovremeno usporediti svih osam razlika slikovnih elemenata sa pragom. Zbog toga je potrebno osam puta kopirati prag u neko polje. Takvo polje sprema se u jedan od MMX registara. Poziva se funkcija *Spremi Prag_MMX* koja izgleda ovako:

```
#define SpremiPrag_MMX(prag) __asm__ volatile(
                                    "\n\t movq %0, %%mm7\n"
                                          :
```

```
                                        : "m"(prag)
                                        : "%mm7"
                                      );
```

Tijelo funkcije *SpremiPrag_MMX* odgovara prethodno prikazanoj sintaksi u poglavlju ***3***. Kao ulazna vrijednost referencira se memorijska lokacija na koju pokazuje varijabla *prag*. Koristi se ***"m"***, što znači da se čita vrijednost varijable *prag* i ista se pomoću instrukcije ***movq*** sprema u registar MM7.

Nakon spremanja praga u registar MM7 potrebno je u MMX registre spremiti sliku pozadine i sliku koja se uspoređuje sa pozadinom (trenutna slika) kako bi se utvrdilo je li došlo do pomaka. Unutar svakog retka grupira se po osam slikovnih elemenata koji se učitavaju u MMX registar. Za spremanje u registre MMX koristi se instrukcija ***movq*** (eng. Move Quadword). U poglavlju ***2.2.*** objašnjeno je da se rad algoritma temelji na apsolutnoj razlici pojedinih slikovnih elemenata pozadine i trenutne slike. Zbog toga se pozadina sprema u dva MMX registra (MM0 i MM2) i na taj se način sprečava gubitak vrijednosti kod izračunavanja i spremanja rezultata razlike. Prvo se od slikovnih elemenata pozadine koje se nalaze unutar registra MM0 oduzimaju slikovni elementi trenutne slike i dobiveni rezultat sprema se natrag u registar MM0. Zahvaljujući prethodnom spremanju pozadine u dva registra MMX ne dolazi do gubitka vrijednosti slikovnih elemenata pozadine, već se one nalaze u registru MM2. Oduzimaju se od slikovnih elemenata trenutne slike, dobiveni rezultat pohranjuje se umjesto elemenata trenutne slike unutar registra MM1. Za izračunavanje razlike slikovnih elemenata koristi se instrukcija ***psubusb*** (eng. Packed Subtract Unsigned with Saturation Support Packed Byte) U odjeljku ***2.5.1*** je spomenuto kako postoje instrukcije sa zasićenjem. Ovdje se radi o jednoj takvoj instrukciji. Ako je rezultat oduzimanja negativan, postavlja se na nulu. Na slici 4.1. demonstriran je njezin rad.



Slika 4.1. Instrukcija psubusb

Nakon oduzimanja vrijednosti slikovnih elemenata pozadine od trenutne slike i obrnuto, koristi se instrukcija ***por*** (eng. *Packed Data Bitwise OR*) Ova instrukcija vrši operaciju *logičkog ili* nad 8-bitnim podacima unutar 64-bitnog registra MMX. Uspoređuje 8-bitni podatak prethodno dobivene razlike pohranjene u registar MM0 s 8-bitnim podatkom na istoj poziciji registra MM1. Jedan od ova dva podatka uvijek će biti nula. U slučaju da nije došlo do pomaka tada oba podatka iznose nula. Idući korak je kreiranje maske. Maska se kreira instrukcijom ***pcmpgtb*** (eng. *Packed Compare for Greater Than Supports Packed Byte*) na temelju rezultata instrukcije ***por*** unutar registra MM0 i praga spremljenog u registar MM7. Ako su 8-bitni podaci koji su pohranjeni kao rezultati instrukcije ***por*** unutar 64-bitnog registra MM0 veći od praga, na odgovarajućoj poziciji svih osam bitova postavlja se u jedinice (upisuje se vrijednost 255). Može se zaključiti da je detektiran pomak za pojedini slikovni element i istom se pridjeljuje bijela boja. U slučaju da rezultat instrukcije ***por*** nije veći od praga svih se osam bitova na odgovarajućoj poziciji unutar MMX registra postavlja u nulu. Znači da do pomaka nije došlo ili je razlika slikovnih elemenata manja od zadane vrijednosti praga, te se ne uzima u obzir. Dobiveni rezultat, zapravo maska zapisuje se na željenu memorijsku lokaciju. Na kraju asemblerskog koda potrebno je koristiti instrukciju ***emms*** (eng. *Empty MMX Technology State*) koja označava kraj rada s instrukcijama MMX i omogućuje daljnje ispravno izvođenje programa. Opisani dio napisan u asemblerskom jeziku prikazan je niže:

```
__asm__ (
       // instrukcija                  # komentar
      "\n\t movq %2,%%mm1             \t# trenutna -> mm1"
```

```
        "\n\t movq %1,%%mm0             \t# pozadina -> mm0"
        "\n\t movq %%mm0,%%mm2          \t# pozadina-> mm2 (kopija u mm2)"
        "\n\t psubusb %%mm1, %%mm0      \t# razlika1 = pozadina - trenutna"
        "\n\t psubusb %%mm2, %%mm1      \t# razlika2 = trenutna - pozadina"
        "\n\t por %%mm1, %%mm0          \t# razlika1 ili razlika2"
        "\n\n pcmpgtb %%mm7, %%mm0      \t# kreiraj masku u ovisnosti o pragu u
                                            mm7"
        "\n\t movq %%mm0,%0             \t# vrati masku"
              : "=m" (mmxRezultat[index])      // %0 rezultat
              : "m"  (mmxPozadina[index])      // %1 slika pozadine
              , "m"  (mmxPomak[index])  // %2 trenutna slika
              : "%mm0", "%mm1", "%mm2", "%mm7"
              );
 __asm__("emms" : : );
```

Za potrebe analize dobivenog ubrzanja obrade slike pomoću instrukcija MMX, algoritam oduzimanja pozadine implementiran je i u programskom jeziku C bez umetnutih dijelova koda pisanih u asembleru. Radi se o jednostavnoj funkciji prikazanoj niže:

```
extern "C" {
       void MotionDetect_C( unsigned char *pozadina, unsigned char *pomak,
                            unsigned char *rezultat, long visina, long sirina,
                            unsigned char prag)
       {
              int i;
              long velicina=visina*sirina;

              for(i=0; i<velicina; i++)

              rezultat[i] = ((abs((int)(pozadina[i])-(int)(pomak[i])) > prag) ?
                              255 : 0);
       }
}
```

3.1.2. Integracija algoritma u ljusku cvsh

Algoritam oduzimanja pozadine nalazi se unutar stabla izvornog koda ljuske, točnije unutar direktorija *cvsh_src/ext/ozana*. Definiran je u okviru razreda *alg_MotionDetectMMX*. Svaki put kad korisnik zatraži obradu slike virtualnim pozivom metode *process()* poziva se korisnički algoritam. Metoda je deklarirana unutar razreda *alg_MotionDetectMMX* kao što je prikazano niže:

```
virtual void process(
       const img_vectorAbstract& src,
       const win_event_vectorAbstract& events,
       int msDayUtc);
```

Na samom početku metode *process()* inicijalizira se pozadina po potrebi, a sam poziv izgleda ovako:

```
void alg_MotionDetectMMX::process(
       const img_vectorAbstract& src,
       const win_event_vectorAbstract&,
       int
   )
```

Važno je napomenuti da korištenje funkcije *img_access_copy* ulaznu sliku pretvara u format izlazne slike, što omogućava izvođenje algoritma za niz slika u boji i niz sivih slika sa jednom komponentom boje.

3.1.3. Pokretanje algoritma

Za pokretanje algoritma potrebno je upisati parametre naredbenog retka u razvojnom okruženju. Naredba se sastoji od sljedećih dijelova:

-sf – putanja do izvorne datoteke slike

Primjer: *-sf="C:\...\Sekvence\film1.avi"*

-i – specificira se interaktivni rad, može se zadati i početna naredba.

-a – definira se algoritam koji se koristi za obradu slike

Primjer*: -a=MotionDetectMMX*

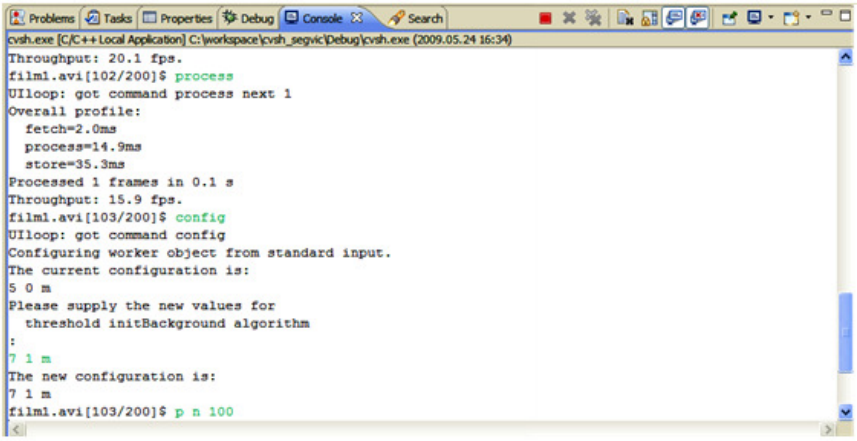
- ***c*** – definira se konfiguracija

Kao što je spomenuto nakon izvođenja početne naredbe koja ne mora biti zadana unutar opcije ***–i*** daljnje upravljanje prepušta se korisniku. Korisnik putem tekstualnog sučelja unosi željene naredbe. Izmjenu parametra omogućuje naredba ***config***, radi bržeg pozivanja može se koristiti samo ***c***. Potrebno je unijeti nove vrijednosti praga, inicijalizirati pozadinu i odabrati između obrade instrukcijama MMX ili u samom C-u:

m – označava da će se sljedeći niz slika obraditi instrukcijama MMX

c – za obradu niza slika poziva se algoritam pisan u C kodu (nema optimizacije)

Za prikaz i obradu slijeda ulaznih slika koriste se naredbe: ***process*** ili kraće ***p*** i ***show*** (***s***). Naredbom ***process next*** (***p n***) može se odabrati i broj slika koji se šalje na obradu. Isto tako mogu se koristiti naredbe: ***process previous*** (***p p***) i ***process this*** (***p t***). Na primjeru niže prikazano je korištenje nekih naredbi.



Slika 4.2. Pokretanje algoritma Motion Detect

3.2. Eksperimentalni rezultati

3.2.1. Primjer rada algoritma

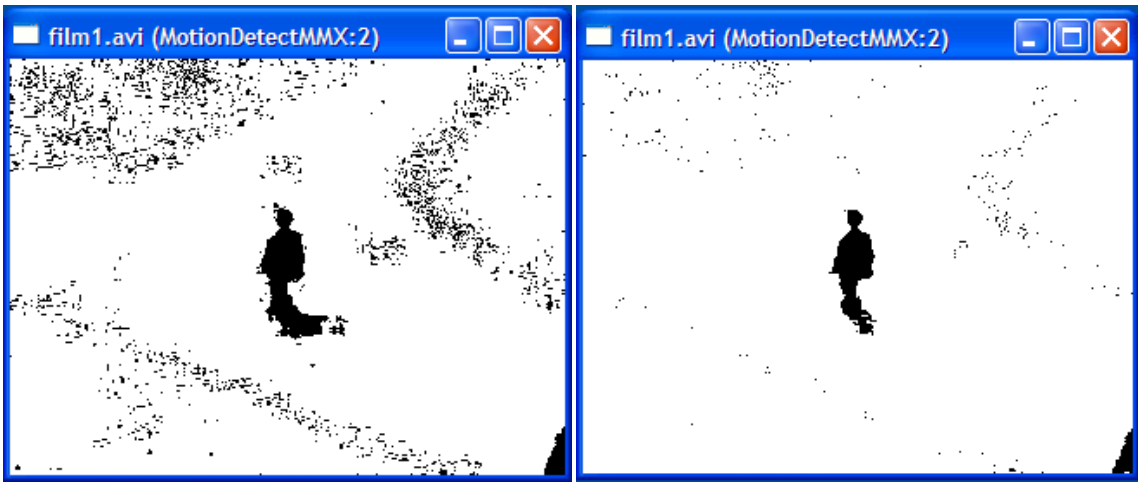
U ovom poglavlju bit će prikazan rad algoritma oduzimanja pozadine. Nakon pokretanja algoritma naredbom ***process*** (***p***) dobivaju se tri slike prikazane niže. Slika 4.3. predstavlja pozadinu s kojom će se uspoređivati sve sljedeće slike za koje se zatraži obrada. Kako bi se pokazalo da rezultati zaista ovise o pragu, u nastavku dani su rezultati dviju zasebnih obrada koje se jedino razlikuju za zadanu vrijednost praga.

Naredbom ***config*** (***c***) unese se željeni parametri. U svim eksperimentiranjima kao pozadinska slika inicijalizira se nulta slika iz slijeda i odabire se algoritam implementiran instrukcijama MMX. Za prvu obradu vrijednost parametara praga iznosi 7, a kod druge obrade ona je gotovo dvostruko veća, tj. iznosi 15.

Slika 4.3. Pozadina

Naredbom ***process next 100*** (***p n 100***) zatražena je obrada idućih sto slika. Na slici 4.4. prikazano je stanje nakon obrade.

a)



b) c)

Slika 4.4. a) pomak, b) rezultat za prag 7, c) rezultat za prag 15

Kao što je rečeno kod opisa algoritma, slikovni elementi nakon obrade poprimaju isključivo vrijednosti 255 ili 0. Upravo se to vidi na slikama 4.4. b) i 4.4. c) gdje su slikovni elementi za koje je detektiran pomak bijele, a ostali crne boje. Povećanjem praga bijelu boju poprimit će slikovni elementi na mjestima gdje su veće razlike, kao što je ovdje kretanje čovjeka. Algoritam implementiran u C-u bez instrukcija MMX daje identične rezultate kao što su slike 4.4. b) i 4.4. c), međutim potrebno mu je duže vrijeme obrade.

3.2.2. Analiza ubrzanja obrade slike

Dok se u verziji algoritma pisanog u jeziku C svaka instrukcija istovremeno izvodi samo nad jednim elementom, kod verzije sa tehnologijom MMX ta ista instrukcija izvodi se istovremeno nad osam slikovnih elemenata. Zbog toga se očekuje oko osam puta brže izvođenje algoritma.

Kako bi se moglo utvrditi stvarno ubrzanje, za potrebe testiranja isključena je optimizacija koju pruža prevoditelj GCC (postavljena je opcija -O0). Testiranje je provedeno na slici rezolucije 640x480. Vrijeme se mjeri kroz 10000 poziva algoritma. Prosječno vrijeme dobiveno kroz pet mjerenja za obje verzije algoritma prikazano je niže:

Tablica 1. Prosječno ubrzanje instrukcijama MMX bez optimizacije prevoditelja

	MMX	C	UBRZANJE
PROSJEČNO	2.68 s	26.64 s	9.94

Dobiveni omjer radi lakše usporedbe zaokružen je na dvije decimale. Kao što je prikazano, algoritam implementiran instrukcijama MMX izvodi se skoro deset puta brže od algoritma napisanog u samom C kodu. Programski jezik C koristi mnogo više opsežnije grananje u odnosu na MMX instrukcije te ga to još dodatno usporava. To je jedan od razloga zbog čega je ubrzanje skoro deset, a ne očekivanih osam puta.

Tablica 2. prikazuje prosječno ubrzanje dobiveno kroz pet mjerenja uz uključenu optimizaciju koju pruža prevoditelj GCC (postavljena je opcija -O3). Obavlja se testiranje za istu sliku rezolucije 640x480 i sa svim ostalim identičnim parametrima. Vrijeme potrebno za obradu slike smanjilo se oko tri puta. Ubrzanje je relativno ostalo isto, nešto malo manje u odnosu na rezultat tablice 1.

Tablica 2. Prosječno ubrzanje instrukcijama MMX uz optimizaciju prevoditelja

	MMX	C	UBRZANJE
PROSJEČNO	0.82 s	7.44 s	9.10

4. Primjena vektorskih instrukcija u implementaciji konvolucije

4.1. Programska izvedba

4.1.1. Implementacija jednodimenzionalne konvolucije

Jednodimenzionalna konvolucija opisana u odjeljku ***2.3.2*** implementirana je korištenjem proširenja SSE3. Na ulazu nalazi se slika bajtova, a na izlazu slika s 32-bitnim podacima u notaciji pomičnog zareza. Slikovni elementi pretvaraju se u zapis sa pomičnim zarezom duljine 32 bita. Unutar jednog registra XMM veličine 128 bita spremljena su četiri takva slikovna elementa, koja će se istovremeno obrađivati. Korištenje zapisa sa pomičnim zarezom ima svoje nedostatke. Uglavnom je to što se smanjuje paralelna obrada podataka. Kod algoritma oduzimanja pozadine implementiranog instrukcijama MMX, istovremeno se obrađivalo osam slikovnih podataka, dok je to ovdje upola manje.

Za implementaciju jednodimenzionalne konvolucije instrukcijama SSE koristi se umetnuti dijelovi koda pisani u asemblerskom jeziku identične sintakse kao i kod ekstenzija MMX. Detaljni opis sintakse može se vidjeti u poglavlju ***3***. Slikovni elementi spremaju se u polje, neovisno da li se radi o elementima stupca ili retka. Polovina duljine konvolucijske jezgre određuje broj elemenata sa svake strane polja čije vrijednosti se neće mijenjati.

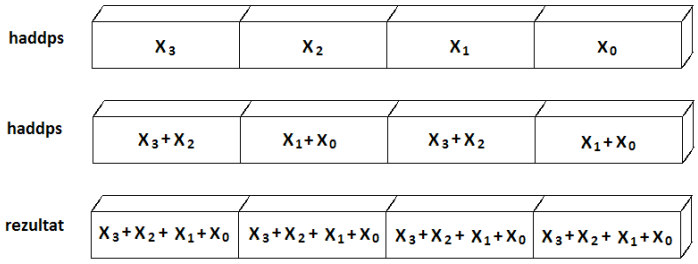
Problemu implementacije jednodimenzionalne konvolucije može se pristupiti s dvije strane. U skladu s tim nastale su potpuno različite verzije implementacije, detaljno opisane u nastavku. U prvoj verziji implementacije prolazi se slijedno po slikovnim elementima i za svaki od njih, koji se u danom trenutku smatra centralnim, paralelno se izvode koraci konvolucije. Svakim prolaskom dobiva se rezultat konvolucije samo za trenutno promatrani centralni slikovni element. Isto tako, sam princip rada druge verzije je gotovo isti. U odjeljku ***5.2.2*** prikazani su rezultati dobiveni implementacijom ovih dviju verzija. Tamo se može vidjeti kako prva verzija implementacije uključivanjem optimizacije prevoditelja *GCC* pokazuje minorno ubrzanje. Pretpostavlja se da najviše vremena odlazi

na pristupanje memoriji. Odnosno, prilikom spremanja u memoriju i učitavanja iz memorije rezultata svake od iteracija. S ciljem rješavanja ovog problema nastaje druga verzija implementacije. Za razliku od prve implementacije međurezultat svake od iteracija sprema se u posebni registar XMM i na taj način izbjegava se stalno pristupanje memoriji. Princip rada treće verzije uvelike se razlikuje se od prethodnih verzija. Ovdje se unutar procesora paralelno izvodi isti korak konvolucije nad različitim podacima. Najme, četiri različita slikovna elementa pohranjena unutar jednog registra XMM istovremeno se uzimaju kao centralni elementi i nad njima se izvodi jedna vektorska instrukcija.

4.1.1.1. Prva verzija implementacije

Prvi korak je spremanje dijela konvolucijske jezgre u registar XMM1. U jedan registar XMM moguće je spremiti samo četiri 32-bitne vrijednosti. Isto tako, za pojedini slikovni element koji se u danom trenutku smatra centralnim u registar XMM2 spremaju se po četiri 32-bitna elementa. Prolaskom kroz vanjsku for petlju slikovni elementi izvorne slike spremaju se počevši od pozicije lijevog elementa udaljenog *(duljina_jezgre)/2 od* centralnog. Na samom početku tijela ove for petlje, varijabla *rez* postavlja se na nulu. Slijedi unutarnja for petlja, također napisana u u višem programskom jeziku. Jedini razlog stvaranja te for petlje proizlazi iz toga što duljina jezgre može biti veća od duljine registra XMM. U jedan registar XMM stanu četiri podatka sa pomičnim zarezom, dok se jezgra može sastojati od 3, 5, 7, 9 i više elemenata. To povlači potrebu za višestrukim prolascima po elementima jezgre. Broj takvih prolazaka jednak je (*duljina_jezgre)/4 +1*. Tijelo ove for petlje predstavlja asemblerski kod. Unutar njega prvo se u registar XMM1 spremaju četiri 32-bitna elementa konvolucijske jezgre. Zatim slijedi spremanje četiri slikovna elementa u registar XMM2. Usporedno množenje slikovnih elemenata spremljenih unutar registra XMM2 sa vrijednostima jezgre na odgovarajućim pozicijama unutar registra XMM1 omogućuje instrukcija ***mulps*** (eng. Multiply Parallel Scalars). Rezultat množenja za pojedini element prepisuje se preko njegove vrijednosti na odgovarajućoj poziciji unutar registra XMM2. Pomoću instrukcije ***haddps*** (eng. *Packed Single-FP Horizontal Add*) dobiveni umnošci horizontalno se zbrajaju. Instrukcija obavlja zbrajanje parova susjednih elemenata vektorskog registra. Rezultat ove instrukcije sprema se u isti registar XMM2 (vidi sliku 5.1.). U idućem koraku instrukcija ***movss*** (eng. *Move Single Scalar*) s memorijske

lokacije učitava 32-bitnu sumu prijašnjih iteracija u registar XMM3. Pomoću instrukcije ***addss*** (eng. *Add Single Scalar*) toj se sumi pribraja nižih 32 bita rezultata instrukcije ***haddps*** iz registra XMM2. Ponovno se koristi instrukcija ***movss*** koja niža 32 bita konačnog rezultata pohranjuje na željenu memorijsku lokaciju. Valja primijetiti da tijekom izvođenja ovih instrukcija nema usporedne obrade podataka. U slučaju da postoji još koji prolazak kroz petlju opisani postupak se ponavlja, te se na kraju konačan rezultat konvolucije zapisuje na poziciju centralnog slikovnog elementa odredišne slike.



Slika 5.1. Instrukcija haddps

Opisani dio napisan u asemblerskom jeziku može se vidjeti niže:

```
for(i=k; i<(sirina-k); i++){
       rez = 0.0;
       for(j=0; j<=velicina_jezgra/4; j++){
              __asm__ (
                  // instrukcija                 # komentar
              "\n\t .align 16                        "
              "\n\t movupd  %1,%%xmm1          \t# jezgra -> xmm1
              "\n\t movupd %2,%%xmm2           \t# slikovni_ele -> xmm2"
              "\n\t mulps %%xmm1,%%xmm2        \t# slikovni_ele*jezgra->xmm2"
              "\n\t haddps  %%xmm2,%%xmm2      \t# horizonatalno zbraja"
              "\n\t haddps  %%xmm2,%%xmm2      \t# horizonatalno zbraja "
              "\n\t movss %0, %%xmm3           \t# rez iteracija ->xmm3"
              "\n\t addss %%xmm2, %%xmm3       \t# niža 32-bita xmm2 +rez->xmm3"
              "\n\t movss  %%xmm3,%0           \t# niža 32-bita xmm3->rez"
                : "+m" (rez)                     // %0
                : "m"  (jezgra[4*j])             // %1
                , "m"  (redak_sl[i-k+4*j])       // %2
```

```
                : "%xmm1", "%xmm2", "%xmm3"
            );
        }
        redak_rezultat[i] = rez;
}
```

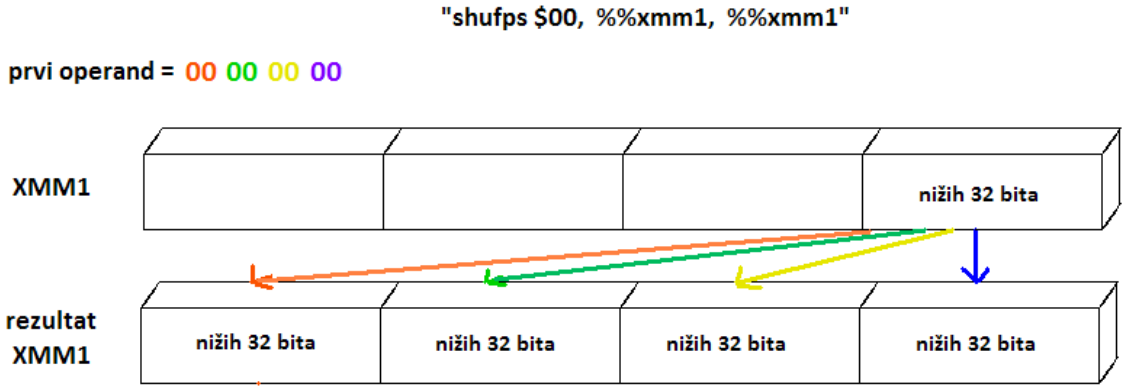
4.1.1.2. Druga verzija implementacije

Sljedeće ćemo analizirati na koji način se može izbjeći stalno pristupanje memoriji. Princip rada jednak je prethodno opisanom. Također se koristi vanjska for petlja napisana u višem programskom jeziku kojom se prolazi po svim elementima slike, izuzetak su elementi na rubovima slike. Na samom početku tijela for petlje, varijabla *rez* postavlja se na nulu. Unutar koda napisanog u asembleru, instrukcijom ***movss*** ista se sprema u registar XMM3. Slijedi unutarnja for petlja. Kao što je objašnjeno, broj poziva ove petlje unutar koje se pohranjuju nove vrijednosti u XMM registre ovisi o duljini konvolucijske jezgre i jednak je (*duljina_jezgre)/4+1*. U registar XMM1 spremaju četiri 32-bitne vrijednosti jezgre, dok se u registar XMM2 spremaju četiri slikovna elementa. Instrukcijom ***mulps*** množe se vrijednosti slikovnih elemenata unutar registra XMM2 sa vrijednostima jezgre na odgovarajućim pozicijama registra XMM1. Rezultati množenja spremaju se natrag na odgovarajuće pozicije registra XMM2. Dobiveni umnošci horizontalno se zbrajaju instrukcijom ***haddps*** unutar registra XMM2. Princip rada ove instrukcije opisan je ranije. Slijedi dio koji predstavlja osnovnu razliku. Nižih 32-bita rezultata horizontalnog zbrajanja iz registra XMM2, instrukcijom ***addss*** pribraja se vrijednosti pohranjenoj u registru XMM3. Ukoliko postoji još koji prolazak kroz unutrašnju petlju opisani postupak se ponavlja, a rezultat trenutne iteracije pribrajat će se vrijednosti prethodne iteracije spremljene u registru XMM3. Nakon zadnjeg prolaska kroz unutrašnju for petlju konačan rezultat konvolucije nalazi se u registru XMM3. Nakon unutrašnje for petlje slijedi dio asemblerskog koda gdje se rezultat konvolucije iz registra XMM3 sprema na poziciju centralnog slikovnog elementa odredišne slike. Vanjskom for petljom nastavlja se prolazak kroz slikovne elemente i ponavlja se opisani postupak izračunavanja konvolucije. Objašnjen izvorni kod može se vidjeti niže:

```
for(i=k; i<(sirina-k); i++) {
	rez = 0.0;
	__asm__  (
		// instrukcija		# komentar
		"\n\t .align 16		"
		"\n\t movss %0, %%xmm3	\t# rez ->xmm3"
			:
			: "m" (rez)	// %0
			: "%xmm3"
		);
	for(j=0; j<=velicina_jezgra/4; j++) {
		__asm__  (
			// instrukcija		# komentar
			"\n\t .align 16		"
			 "\n\t movupd  %0,%%xmm1	\t# jezgra -> xmm1"
			 "\n\t movupd %1,%%xmm2	\t# slikovni_ele -> xmm2"
			"\n\t mulps %%xmm1,%%xmm2	\t# slikovni_ele*jezgra->xmm2"
			"\n\t haddps  %%xmm2,%%xmm2	\t# horizonatalno zbraja"
			"\n\t haddps  %%xmm2,%%xmm2	\t# horizonatalno zbraja"
			"\n\t addss %%xmm2, %%xmm3	\t# xmm2 + xmm3->xmm3"
				:
				: "m"  (jezgra[4*j])	// %0
				, "m"  (redak_sl[i-k+4*j])	// %1
				: "%xmm1", "%xmm2", "%xmm3"
			);
	       }
		__asm__  (
			// instrukcija			# komentar
			 "\n\t .align 16		"
			"\n\t movss  %%xmm3,%0		\t# xmm3->rez"
				: "=m" (redak_rezultat[i])	// %0
				:
				: "%xmm3"
			);
}
```

4.1.1.3. Treća verzija implementacije

I u ovoj verziji implementacije vanjska for petlja pisana je u višem programskom jeziku. Vanjskom petljom prolazi se po svim elementima, počevši od elementa udaljenog za (*duljina_jezgre/2*) od početnog. U svakom sljedećem pozivu pomičemo se za četiri memorijske lokacije. Na taj način grupiraju se po četiri slikovna elementa koja će se učitati u jedan registar XMM i nad kojim će se paralelno izvoditi koraci konvolucije. Slijedi dio koda napisan u asemblerskom jeziku. Unutar njega vrijednost u registru XMM3 postavlja se na nulu korištenjem instrukcije ***pxor*** (eng. *Packed Data Bitwise XOR*). Kao što je spomenuto u prvom prolazu vanjske for petlje kreće se od elementa udaljenog za vrijednost (*duljina_jezgre/2*) od početnog. Slijedi unutarnja petlja pisana u višem programskom jeziku. Ovom petljom prolazi se po svim elementima konvolucijske jezgre. Tijelo petlje predstavlja dio koda napisan u asembleru. U svakom prolazu instrukcijom ***movss*** sprema se jedna vrijednost konvolucijske jezgre u nižih 32 bita registra XMM1. Isto tako, instrukcijom ***movups*** (eng. *Move Unaligned Packed Single-Precision FP Values*) u registar XMM2 spremaju se četiri slikovna elementa počevši od memorijske lokacije pohranjene u varijabli *i0*. Potrebno je vrijednost jednog elementa konvolucijske jezgre pohranjenog unutar 32 niža bita registra XMM1 postaviti i na ostale tri 32-bitne skupine unutar registra XMM1. To nam omogućava instrukcija ***shufps*** (eng *Shuffle Packed Single-Precision FP Values*). Prvi operand ove instrukcije je 8-bitni i u ovom slučaju iznosi $00. Bitovi 0 i 1 ovog operanda odabiru koja će vrijednost iz odredišnog registra XMM1 biti zapisana na nižih 32-bita istog registra. Bitovi 2 i 3 odabiru vrijednost koja će biti pohranjena unutar iduće 32-bitne skupine odredišnog registra XMM1 itd. Na slici 5.2. demonstriran je rad funkcije ***shufps*** .



Slika 5.2. Instrukcija shufps

Nakon pohrane iste vrijednosti konvolucijske jezgre na sve četiri pozicije ([127-96],[95-64],[63-32],[31-0]) registra XMM1, vrši se operacija množenja. Instrukcijom ***mulps*** paralelno se množe sva četiri slikovna elementa unutar registra XMM2 s istim vrijednostima konvolucijske jezgre unutar registra XMM1. Rezultat množenja sprema se natrag na odgovarajuću poziciju unutar registar XMM2. Za svaki od četiri slikovna elementa na odgovarajućim pozicijama registra XMM3 nalazi se zbroj rezultata množenja prijašnjih iteracija. Instrukcijom ***addps*** (eng. *Add Parallel Scalars*) rezultat trenutne iteracije pribraja se vrijednostima prethodnih unutar registra XMM3. Nakon izlaska iz unutrašnje for petlje u registru XMM3 nalaze se rezultati konvolucije za sva četiri slikovna elementa. Slijedi asemblerski dio gdje se instrukcijom ***movups*** rezultati konvolucija spremaju natrag u memoriju. Objašnjen izvorni kod može se vidjeti niže:

```
for(int i=k; i<sirina-k; i+=4)
  {
    __asm__ (
      "\n\t pxor %%xmm3, %%xmm3 \t# 0 ->xmm3"
          : :: //"%xmm3"
       );

    int i0=i-k;
    for(int j=0; j<velicina_jezgra; ++j) {

      __asm__ (
        "\n\t movss  %0, %%xmm1"
```

```
        "\n\t shufps $00, %%xmm1, %%xmm1"
        "\n\t movups %1,  %%xmm2"
        "\n\t mulps %%xmm1,%%xmm2"
         "\n\t addps %%xmm2,%%xmm3"
           :
           : "m"  (jezgra[j])         // %0
           , "m"  (redak_sl[i0+j])  // %1
           : //"%xmm1", "%xmm2", "%xmm3"
         );
    }
    __asm__ (
      "\n\t movups  %%xmm3,%0"
        : "=m" (redak_rezultat[i])    // %0
        :
        : //"%xmm3"
      );
  }
```

Na samom početku asemblerskog koda navodi se instrukcija ***.align 16*** koja vrši poravnavanje podataka na 16 bajtne memorijske lokacije i time ubrzava pristup podacima u memoriji. U odjeljku ***2.4.2*** navedeno je kako registri XMM ne referenciraju registre stoga FPU, već se radi o potpuno odvojenim 128-bitnim registrima. Zbog toga se ovdje na kraju asemblerskog koda ne koristi instrukcija ***emms***.

Za potrebe testiranja, jednodimenzionalna konvolucija implementirana je i u programskom jeziku C. Kod jednostavne funkcije koja obavlja konvoluciju prikazan je niže:

```
extern "C" {
       void C_kon_slika_jezgra(float *jezgra, int velicina_jezgra,
                                float *redak_sl, long sirina,
                                float *redak_rezultat) {
              int k = (velicina_jezgra-1)/2;
              int j,i;
               float rez
       for(i=k; i<(sirina-k); i++){
                     rez=0.0;
                     for(j=0; j<(2*k)+1; j++){
```

```
                        rez += redak_sl[i-k+j]*jezgra[j];
                    }

                    redak_rezultat[i]=rez;
                }
            }
}
```

4.1.2. Pokretanje algoritma konvolucije

Kao i kod pokretanja algoritma oduzimanje pozadine potrebno je upisati parametre naredbenog retka:

-sf – putanja do izvorne slike

Primjer: -sf="C:\...\slika.bmp"

-i – specificira se interaktivni rad

-a – ovdje se definira algoritam koji se poziva

Primjer: -a=convolveSSE

Putem korisničkog sučelja unosi se parametri za: duljinu jednodimenzionalne konvolucijske jezgre, njezine vrijednosti, te se odabire :

s – označava da će se slika obraditi instrukcijama SSE

c – za obradu slike poziva se algoritam pisan u C kodu

4.2. Eksperimentalni rezultati

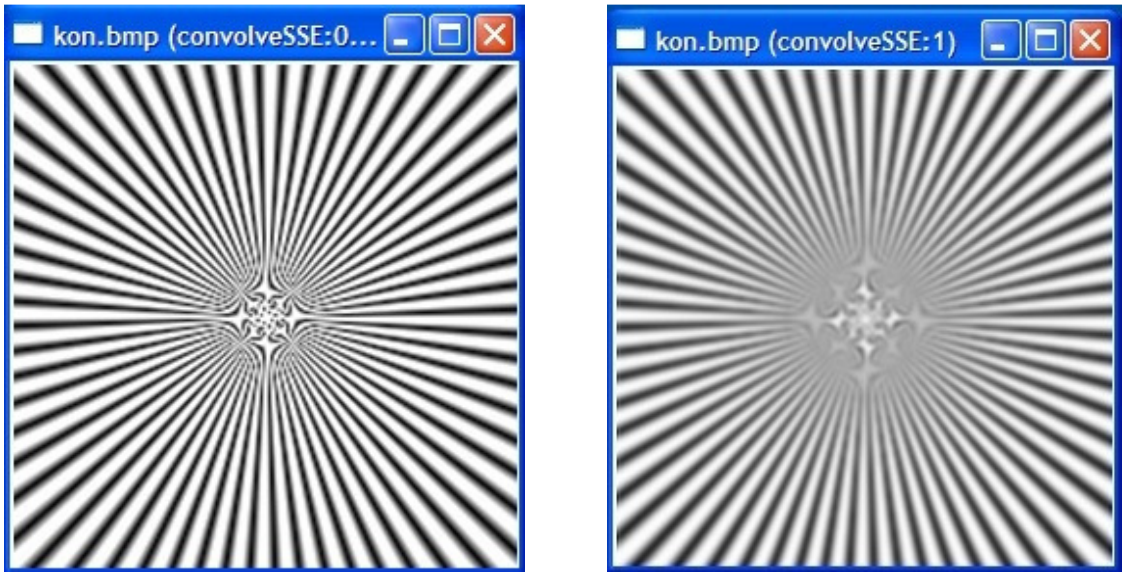
U nastavku ovog poglavlja na nekoliko primjera bit će prikazani rezultati jednodimenzionalne konvolucije. Obrađivat će se samo sive slike sa jednom komponentom boje. Dobiveni rezultati ovise o duljini konvolucijske jezgre i njezinim vrijednostima. Prilikom unosa korisnik mora paziti da zbroj svih vrijednosti konvolucijske jezgre bude točno jedan. Nad svakom ulaznom slikom izvršit će se jednodimenzionalna

konvolucija implementirana i u programskom jeziku C i instrukcijama SSE. Analizirat će se izmjereno vrijeme tijekom ove dvije obrade.

4.2.1. Primjeri konvolucije

Primjer 1.

Slika 5.2. a) prikazuje ulaznu sliku dimenzija 192x 192. Nakon obrade instrukcijama SSE dobiva se identična slika kao i kod obrade programskim jezikom C. Zbog toga je kao rezultat jednodimenzionalne konvolucije prikazana samo jedna slika, u ovom slučaju nakon obrade instrukcijama SSE. Slika 5.2.b) prikazuje rezultat konvolucije Gaussovim filtrom. Unosi se konvolucijska jezgra dimenzije *1x7* sa sljedećim vrijednostima: *0.015625, 0.09375, 0. 234375, 0. 3125, 0. 234375, 0.09375, 0.015625.*

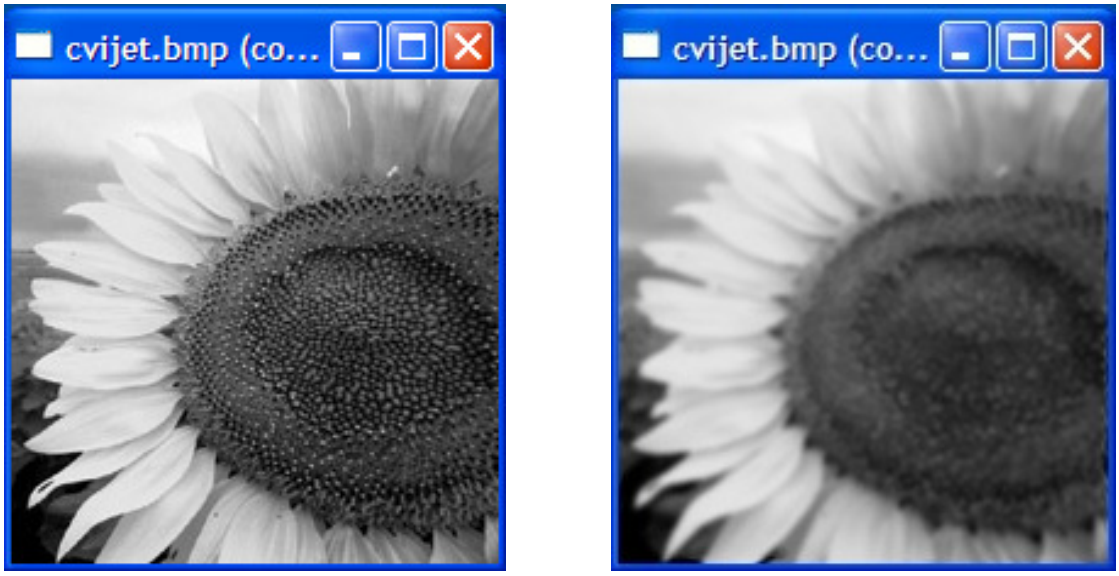


a) b)

5.2. a) ulazna slika, b) rezultat

Primjer 2.

U ovom primjeru kao ulazna slika koristi se slika 5.3. a), dok je na slici 5.3. b) prikazan rezultat konvolucije Gaussovim filtrom dimenzija *1x7*, istih vrijednosti kao i u prethodnom primjeru.

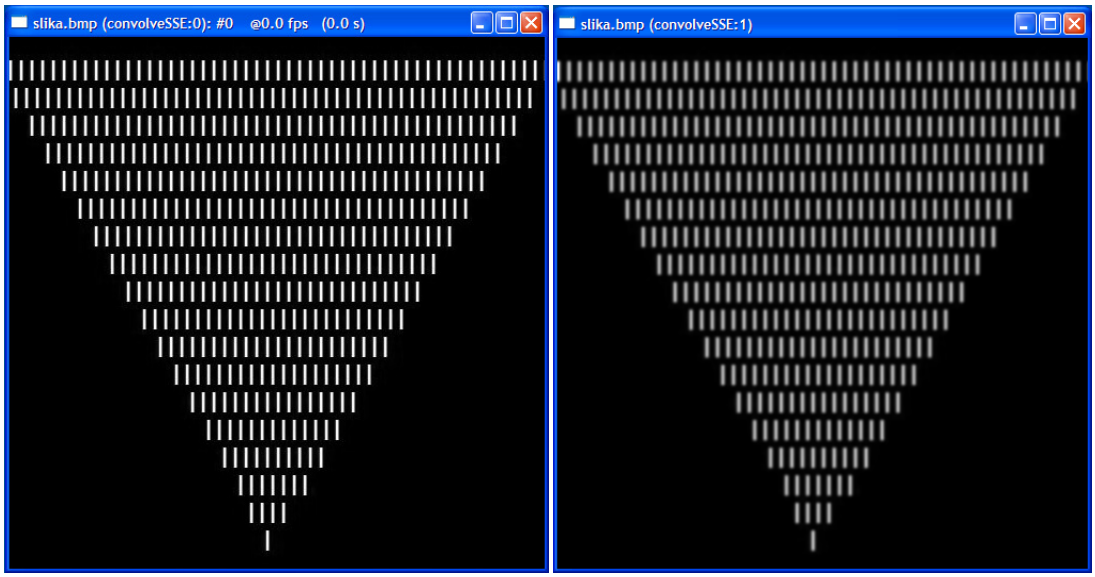


a) b)

5.3. a) ulazna slika, b) rezultat

Primjer 3.

Algoritam konvolucije izvodi se nad ulaznom slikom 5.4.a). Kao parametar naredbenog retka zadaje se dimenzija jezgre *1×7*, identičnih vrijednosti kao i u prethodnim primjerima. Radi se o slici dimenzija 500x500. Odnosno broj slikovnih elemenata retka jednak je broju elemenata stupca. Zbog toga će vrijeme potrebno za izvođenje konvolucije nad stupcem biti jednako vremenu izvođenja nad retkom. Kako je jezgra dimenzija *1x7*, sa svake strane slike prilikom konvolucije zanemarit će se se tri slikovna elementa. Što znači da se unutar svakog retka, odnosno stupca nalazi točno 494 slikovnih elemenata čije vrijednosti će se zamijeniti rezultatom konvolucije. Rezultat konvolucije prikazan je na slici 5.4 b).



a) b)

Slika 5. 4. a) ulazna slika, b) rezultat

4.2.2. Analiza ubrzanja obrade slike

Testiranje algoritma jednodimenzionalne konvolucije za sve tri verzije implementacije izvodi se nad jednodimenzionalnim poljem s ukupno 100 000 elemenata. Zadana je dimenzija jezgre *1×7* i testiranje algoritma obavlja se kroz 10 000 poziva. U nastavku prikazani su rezultati za svaku verziju implementacije, kao i njihova usporedba sa rezultatima C verzije.

4.2.2.1. Analiza rezultata prve verzije implementacije

Prilikom ovog testiranja isključena je optimizacija koju pruža prevoditelj GCC (opcija –O0) kako bi se mogli vidjeti stvarno ubrzanje tehnologije SSE. Prosječno vrijeme dobiveno kroz pet mjerenja prikazano je u tablici 3.

Tablica 3. Prosječno ubrzanje instrukcijama SSE bez optimizacije prevoditelja

	SSE	C
PROSJEČNO	32.82 s	54.27 s

Iz prikazanog rezultata može se vidjeti kako se algoritam implementiram instrukcijama SSE izvodi brže od algoritma implementiranog u C-u. Dobiveno ubrzanje algoritma iznosi oko 40%.

U tablici 4. prikazano je prosječno vrijeme dobiveno kroz pet mjerenja uz uključenu optimizaciju prevoditelja.

Tablica 4. Prosječno vrijeme potrebno za konvoluciju instrukcijama SSE uz optimizaciju prevoditelja

	SSE	C
PROSJEČNO	20.48	11.95

Kao što se vidi uz uključenu optimizaciju (opcija –O3) ubrzanja nema. Pretpostavljalo se da pristupanje memoriji uvelike usporava algoritam. U nastavku prikazani su rezultati dobiveni rješavanjem nepotrebnog pristupanja memoriji.

4.2.2.2. Analiza rezultata druge verzije implementacije

Prosječno vrijeme dobiveno kroz pet mjerenja uz isključenu optimizaciju prikazano je u tablici 5. niže.

Tablica 5. Prosječno vrijeme ubrzanja instrukcijama SSE bez optimizacije prevoditelja

	SSE	C
PROSJEČNO	27.8 s	54.27 s

Uz isključenu optimizaciju prevoditelja *GCC,* dobiva se gotovo identičan rezultat kao i kod neprestanog pohranjivanja u memoriju svakog od međurezultata. Uključivanjem optimizacije (opcija –O3) prosječno vrijeme iznosi:

Tablica 6. Prosječno vrijeme potrebno za konvoluciju instrukcijama SSE uz optimizaciju prevoditelja

	SSE	C
PROSJEČNO	15.55 s	11.95 s

Rješavanje pristupa memoriji, suprotno od očekivanja značajno ne ubrzava algoritam. Isto tako, možemo zaključiti da uz uključene optimizacije ubrzanja uopće nema. Razlog vjerojatno leži u korištenju instrukcije SSE3, ***haddps***. Izvođenjem ove instrukcije gubi se mnogo vremena. Vjerojatno i činjenica da se istovremeno obrađuju samo četiri podatka, a i to što se izvode operacije nad podacima sa pomičnim zarezom zasigurno ima utjecaja. Također, još jedan od razloga je i korištenje implementacije slične onoj u algoritmu oduzimanja pozadine.

4.2.2.3. Analiza rezultata treće verzije implementacije

U nastavku je prikazano prosječno vrijeme kroz pet mjerenja uz isključenu optimizaciju.

Tablica 7. Prosječno vrijeme potrebno za konvoluciju instrukcijama SSE bez optimizacije prevoditelja

	SSE	C
PROSJEČNO	14.61 s	54.27 s

Testiranje uz uključenu optimizaciju (opcija –O3) pokazalo je iduće prosječno vrijeme:

Tablica 8. Prosječno vrijeme potrebno za konvoluciju instrukcijama SSE uz optimizaciju prevoditelja

	SSE	C
PROSJEČNO	7.35 s	11.95 s

Na temelju dobivenih rezultata možemo zaključiti da paralelno izvođenje konvolucije nad četiri slikovna elementa daje gotovo dvostruko brže izvođenje algoritma.

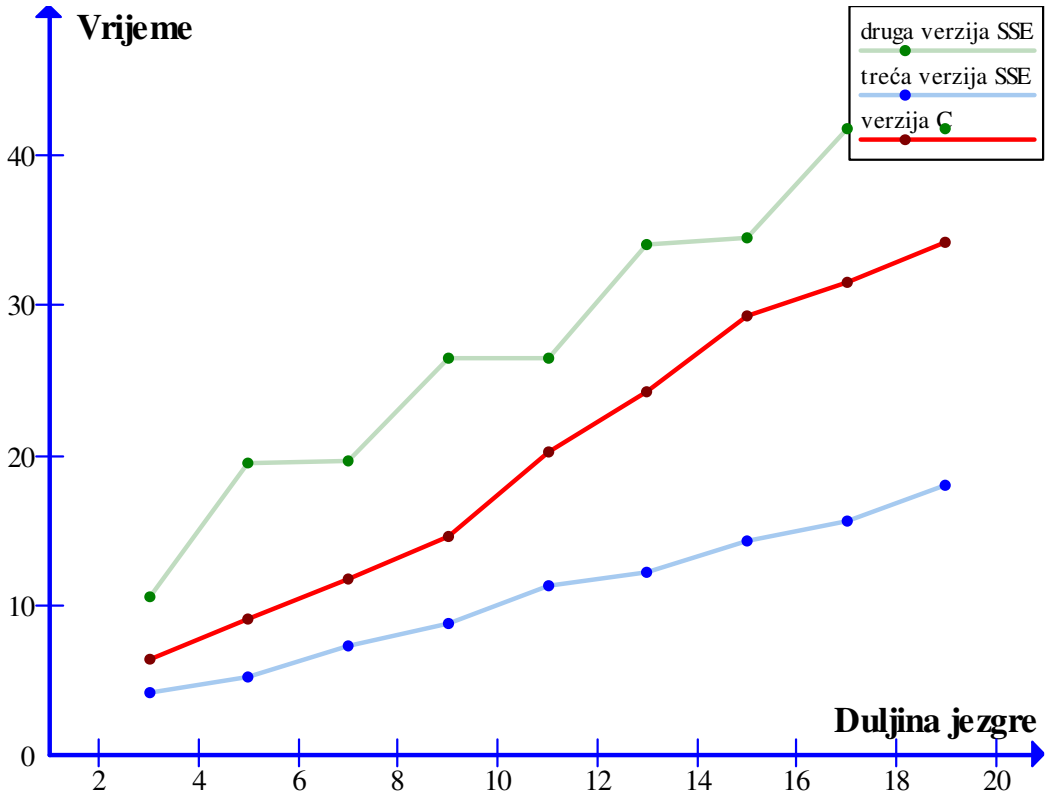
Postavlja se pitanje, zbog čega je ova verzija implementacije toliko brža od prethodnih dviju verzija. Razlog toga se može otkriti usporedbom asemblerskog koda obje implementacije. U [14] se može naći usporedba latencija različitih instrukcija koje koriste moderni Intel i AMD procesori. Potrebno je definirati latenciju. Latencija predstavlja broj taktova jezgre procesora između početka izvođenja promatrane instrukcije i trenutka kada je sljedeća instrukcija u nizu spremna za izvođenje [14]. Analiza latencija u literaturi je provedena za različite modele procesora. Kako su implementacije testirane na Core 2 Duo procesoru, vrijednosti korištene u ovoj usporedbi odnose se na 65 nanometarskih verzija Core 2 Duo procesora.

Tablica 9. Usporedba latencija instrukcija SSE

PRVA I DRUGA VERZIJA SSE		TREĆA VERZIJA SSE	
Instrukcija	Latencija	Instrukcija	Latencija
MOVUPD	3 – 4	MOVSS	3
MOVUPD	3 – 4	SHUFPS	3
MULPS	4	MOVUPS	3 – 4
HADDPS	9	MULPS	4
HADDPS	9	ADDPS	3
ADDPS	3		
Ukupno = 31 – 33		Ukupno= 16 – 17	

Kao što se može vidjeti iz tablice usporedbe, treća implementacija je dvostruko brža što se tiče asemblerskog dijela koda.

Na slici niže se može vidjeti graf usporedbe izvođenja implementacija uz varijabilnu veličinu jezgre. Kako prva i druga verzija implementacije daju gotovo identične rezultate, niže su prikazani grafovi samo za drugu i treću implementaciju, kao i za implementaciju u C-u.



Slika 5.4. Graf brzine izvođenja algoritma

Kao dodatna mogućnost ubrzanja algoritma, analizirala se konvolucija sa cjelobrojnim podacima. Instrukcije SSE omogućavaju istovremene operacije nad ukupno 8 cjelobrojnih podataka duljine 16 bitova. Operacije sa cjelobrojnim podacima su puno brže od operacija sa pomičnim zarezom, ali imaju nekoliko negativnih osobina. Jedna takva osobina je dijeljenje elemenata jezgre sa određenim faktorom. Naime, u jedan registar XMM moglo se spremati osam slikovnih elemenata, svaki bi od njih bio cjelobrojni podatak veličine 16 bita. U tom slučaju prilikom množenja sa vrijednostima jezgre postoji mogućnost da dođe do preljeva iznad maksimalne vrijednosti za cjelobrojne podatke. Da bi se to spriječilo, vrijednosti slikovnih elemenata bilo bi potrebno prethodno podijeliti faktorom čija vrijednost iznosi 256. Dodatna negativna okolnost je nepostojanje određenih instrukcija SSE koje možemo naći za podatke sa pomičnim zarezom. Riječ je instrukciji horizontalnog zbrajanja sa cjelobrojnim podacima. Takva instrukcija ne postoji niti u jednoj inačici proširenja SSE. Navedena instrukcija je najavljena za nadolazeći SSE5 i AVX. Došlo se do zaključka kako zbog toga u budućnosti vrijedi pokušati implementirati verziju sa cjelobrojnim podacima radi provjere mogućnost još većeg ubrzanja ovog algoritma.

5. Zaključak

Kroz ovaj rad je prikazana primjena vektorskih proširenja instrukcijskog skupa arhitekture x86 s ciljem ubrzanja obrade slike. Iz širokog skupa algoritama obrade slike odabrali smo algoritam oduzimanja pozadine i konvoluciju slike. Najjednostavniju varijantu algoritma oduzimanja pozadine implementirali smo instrukcijama MMX. Također, razmotrili smo nekoliko mogućih verzija optimizacije algoritma konvolucije slike. Implementacija svake od ovih verzija ostvarena je instrukcijama SSE i pri tome se koriste instrukcije iz skupa proširenja SSE2 i SSE3.

Ubrzanje obrade slike ovim algoritmima postigli smo paralelnom obradom čitavog skupa podataka. Tako se kod algoritma oduzimanja pozadine istovremeno izvodi jedna instrukcija MMX nad osam slikovnih elemenata veličine jednog okteta. Optimizaciju algoritma konvolucije slike postigli smo paralelnim izvođenjem četiri operacije nad 32-bitnim podacima u notaciji pomičnog zareza.

Primjena vektorskih instrukcija MMX uvelike je ubrzala algoritam oduzimanja pozadine. Analizom rezultata pokazano je kako se algoritam implementiran instrukcijama MMX izvodi i do deset puta brže od implementacije u C-u. Suprotno od toga, prvi pokušaj optimizacije algoritma konvolucije instrukcijama SSE nije pokazao ubrzanje. Iz to razloga, nastavilo se raditi na poboljšanju prve verzije implementacije. Svaki od pokušaja detaljno je opisan u radu. Analiza rezultata krajnje implementacije instrukcijama SSE pokazuje dvostruko brže izvođenje algoritma konvolucije.

Možemo zaključiti da primjena vektorskih instrukcija ne garantira uvijek ubrzanje izvođenja pojedinih dijelova koda. Jedna takva instrukcija je ***haddps*** iz skupa proširenja SSE3. Korištenje ove instrukcije za izvođenje operacije horizontalnog zbrajanja uvelike je usporilo algoritam konvolucije.

Kao daljnji rad nameće se mogućnost implementacije algoritma konvolucije sa cjelobrojni podacima. Također, stečena iskustava mogu se primijeniti pri implementaciji ostalih algoritama obrade slike vektorskim instrukcijama.

6. Literatura

[1] Intel® 64 Architecture x2APIC Specification, lipanj 2008. http://www.intel.com/Assets/PDF/manual/318148.pdf, 25.02.2009.

[2] R. Blum, Professional Assembly Language, Wiley Publishing, Inc., Indianapolis, Indiana, 2005

[3] J. Skoglund i M. Felsberg, Fast image processing using SSE2, Computer Vision Laboratory, Linköping University, Sweden, 2002. http://www.cvl.isy.liu.se/ScOut/Publications/Papers/ssba05_sf.pdf

[4] Trung H. Tran, H.-M. Cho i S.-B. Cho, Performance Enhancement of Motion Estimation Using SSE2 Technology, srpanj 2008. http://www.waset.org/pwaset/v30/v30-33.pdf

[5] C. Soviany, Embedding Data and Task Parallelism in Image Processing Applications, doktorska disertacija, Technische Univ. Delft, 2003. http://www.waset.org/pwaset/v30/v30-33.pdf

[6] Š. Bašić, P. Čepo, I. Dodolović, D. Dostal, S. Grbić, M. Gulić, I. Horvatin, S. Louč, I. Sučić, Cannyev detektor rubova, tehnička dokumentacija predmeta Projekt iz programske potpore, Fakultet elektrotehnike i računarstva, 2008.

[7] Hayes technologies, Software Speed Optimization http://www.hayestechnologies.com/en/techsimd.htm

[8] Streaming SIMD Extensions, *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/Streaming_SIMD_Extensions

[9] MMX (instruction set), *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/MMX_(instruction_set)

[10] SSE, *Wikipedia, the free encyclopedia*, http://en.wikipedia.org/wiki/SSE

[11] S. Tommesani, MMX Primer, prosinac 2004. http://www.tommesani.com/MMXPrimer.html, 12.3. 2009.

[12] Streaming SIMD Extensions (SSE), http://softpixel.com/~cwright/programming/simd/sse.php

[13] A. Majić, Pronalaženje prometnog traka korištenjem upravljivih filtara, Završni rad, Fakultet elektrotehnike i računarstva, 2008.

[14] A. Fog, Instruction tables, Lists of instruction latencies, throughputs and microoperation breakdowns for Intel and AMD CPU's, Copenhagen University College of Engineering, 2009.
http://www.agner.org/optimize/instruction_tables.pdf

[15] D. Rožman i V. Tomas, Motion Detector, Seminar, Fakultet elektrotehnike i računarstva, 2004.
http://dosl.zesoi.fer.hr/seminari/2003_2004/rozman-tomas/Motion_Detector.pdf

7. Sažetak / Abstract

U ovom radu razmatra se uporaba vektorskih ekstenzija MMX i SSE u obradi slike. Navedene ekstenzije uvode nove instrukcije i tipove podataka koji omogućuju aplikacijama postizanje boljih performansi.

Algoritam oduzimanje pozadine implementiran je u programskom jeziku C++ s umetnutim dijelovima koda pisanim u asembleru, koristeći pri tome instrukcije MMX. Ovaj algoritam oduzimanja pozadine temelji se na razlici slikovnih elemenata trenutnog i prethodnog okvira. Sve korištene MMX instrukcije detaljno su opisane, kao i njihova uloga u cijeloj proceduri.

Drugi dio ovog rada bavi se sa separabilnim dvodimenzionalnim filtrima koji se izvode kao dvije jednodimenzionalne konvolucije. Prva konvolucija izvodi se nad recima slike, dok se druga konvolucija izvodi po stupcima rezultata prve konvolucije. Algoritam jednodimenzionalne konvolucije implementiran je instrukcijama SSE (sve do SSE3) pisanih u umetnutim dijelovima asemblera. Svaka instrukcija SSE korištena u algoritmu detaljno je objašnjena.

Usporedba brzine izvođenja implementacije u standardnom C-u i implementacija koje koriste vektorske instrukcije pokazala je da prednost vektorske inačice iznosi od dva do osam puta.

KLJUČNE RIJEČI: MMX, SSE, SSE2, SSE3, proširenja, algoritam oduzimanja pozadine, jednodimenzionalna konvolucija, separabilni filtri, obrada slike.

Optimization of image processing algorithms with vector instructions

This work considers the use of MMX and SSE vector extensions in image processing. Those extensions include various new instruction and data types allowing applications to achieve greater level of performance.

Motion detect algorithm described in this work is written in C++ programming language with embedded parts of inline assembly code used to provide MMX instructions functionality. This motion detection algorithm is based on the difference of a pixel in current and previous frame. All MMX instruction used in the algorithm are heavily documented and described, as well as their role in the whole procedure.

The second part of this works deals with two dimensional separable filters which can be applied as two one-dimensional convolutions. First convolution is applied on image rows, while the second one is applied on columns of first convolution's results. One-dimensional convolution algorithm is implemented using SSE instructions (up to SSE3) written in inline asembly. Each SSE instruction used in the algorithm is described in detail.

Comparisons between standard C algorithms and other algorithms using vector instructions showed that vector instruction algorithms inscrease performance two to eight times.

KLJUČNE RIJEČI: MMX, SSE, SSE2, SSE3, extensions, algorithm Motion detect, one-dimensional convolutions, separable filters, image processing.