SVEUČILIŠTE U ZAGREBU

FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 1096

OPTIMIRANJE KONVOLUCIJE SLIKE VEKTORSKIM INSTRUKCIJAMA U ARITMETICI S NEPOMIČNIM ZAREZOM

Stjepan Hadjić

Zagreb, siječanj 2010.

Sadržaj

1. Uvod

Obrada digitalne slike je veoma specifično područje u računarskoj znanosti. Specifičnost se sastoji u tome što se nad svim elementima slike izvodi mali broj sličnih, jednostavnih operacija. Kako slikovnih elemenata može biti jako puno, ključno je da se jednostavne operacije izvode što je moguće brže.

Procesori, zasnovani na Von Neumannovom principu, obrađuju jedan podatak u jednoj jedinici vremena. Zbog toga, kako bi se instrukcije mogle izvoditi paralelno nad mnogo podataka, moderni procesori opće namjene danas podržavaju vektorske instrukcije.

Jedno od područja gdje su vektorske instrukcije ubrzale obradu podataka je upravo obrada slike. Vektorskim instrukcijama je omogućeno procesiranje većeg broja elemenata slike jednom instrukcijom, što uvelike ubrzava ukupnu obradu. Upravo zato, algoritmi zasnovani na vektorskim instrukcijama mogu biti važna karika u modernoj obradi slike.

U poglavlju 2. razmatraju se koncepti i teorija koja je vezana uz ovaj rad. Objašnjeni su tehnički pojmovi potrebni za razumijevanje rada, te matematički formalizam koji se koristi za postizanje glavnog cilja. U poglavlju 3. je kratko objašnjeno razvojno okruženje u kojem je program nastao. Poglavlje 4, glavni dio rada, prikazuje različite verzije algoritama i njihove implementacije. U poglavlju 5. razmatraju se eksperimentalni rezultati dobiveni testiranjem implementacija algoritama i algoritama za usporedbu.

2. Pregled korištenih koncepata

2.1 Obrada slike

Obrada digitalne slike najčešće podrazumijeva višestruko ponavljanje iste operacije nad elementima slike. U pravilu, slike se sastoje od iznimno velikog broja slikovnih elemenata (engl. 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 (0 do 255).

Jedan od načina na koji se slike obrađuju je korištenje susjednih operatora (engl. neighborhood operators), tj. filtriranja. Kod takve obrade, za određivanje izlazne vrijednosti nekog elementa u obzir se uzimaju njegovi susjedni elementi. Takve metode filtriraju slike kako bi se izoštrili ili zamutili detalji, maknuo šum ili naglasili rubovi.

2.1.1 Konvolucija

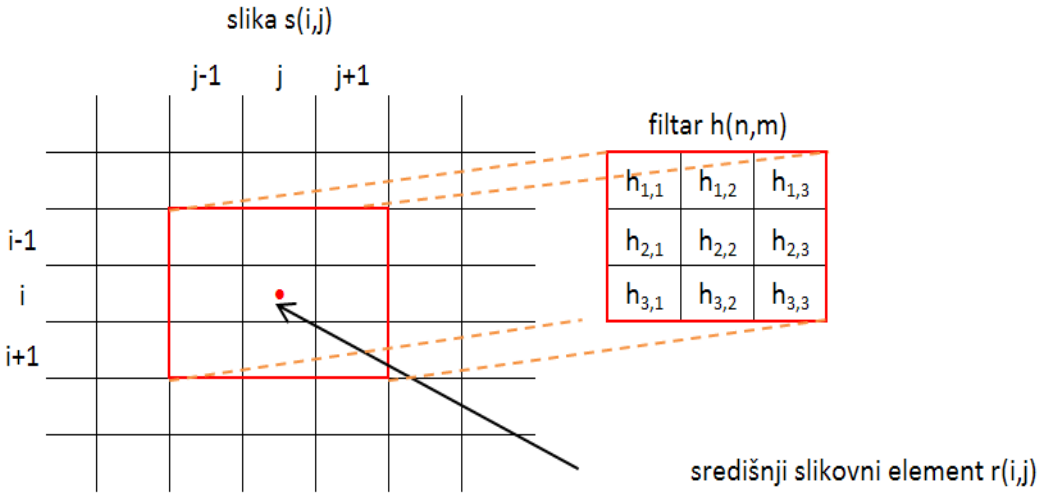
Konvolucija[1] je najčešće korišteni tip susjednih operatora, gdje se vrijednost elemenata izlazne slike određuje kao težinska suma odgovarajućeg susjedstva izvorne slike. Jednadžba (2.1.) prikazuje dvodimenzionalnu diskretnu konvoluciju.

$$r(i,j) = \sum_{n}\sum_{m} s(i-n, j-m)h(n,m) \qquad (2.1.)$$

Vrijednost određenih elemenata izvorne slike množe se sa vrijednostima konvolucijske jezgre (engl. kernel). To množenje se obavlja po jednadžbi (2.1) te se sumiraju dobiveni rezultati. Dobiveni rezultat je vrijednost jednog elementa odredišne slike.

Slika 1. prikazuje dvodimenzionalnu konvoluciju za filtar veličine 3x3.

[1] Jednadžba kontinuirane jednodimenzionalne konvolucije se prikazuje kao:
$r(x) = \int s(x-u)h(u)du.$



Slika 1. Dvodimenzionalna konvolucija za filtar veličine 3x3

U nastavku je rastavljena jednadžba (2.1) za primjer iz slike 1., te je prikazano koji se točno elementi množe te potom zbrajaju.

$$r(i,j) = h_{1,1} \cdot s(i-1,j-1) + h_{1,2} \cdot s(i-1,j) + h_{1,3} \cdot s(i-1,j+1) + h_{2,1} \cdot s(i,j-1) + h_{2,2} \cdot s(i,j) + h_{2,3} \cdot s(i,j+1) + h_{3,1} \cdot s(i+1,j-1) + h_{3,2} \cdot s(i+1,j) + h_{3,3} \cdot s(i+1,j+1) \quad (2.2.)$$

Konvolucija zahtijeva $n \cdot m$ operacija po elementu slike, gdje su n i m širina i visina filtra. Ako se prvo izvrši konvolucija nad retcima slike, te tada nad stupcima slike, broj operacija pada na $n+m$ po elementu slike, što doprinosi ubrzanju obrade. Konvolucijske jezgre za koje je to moguće postići, zovu se separabilne.

U primjeru (2.3) je pokazano kako se separabilna konvolucijska jezgra može napisati kao produkt dva jednodimenzionalna vektora.

$$\begin{bmatrix} 0{,}0625 & 0{,}125 & 0{,}0625 \\ 0{,}125 & 0{,}25 & 0{,}125 \\ 0{,}0625 & 0{,}125 & 0{,}0625 \end{bmatrix} = \begin{bmatrix} 0{,}25 \\ 0{,}5 \\ 0{,}25 \end{bmatrix} * \begin{bmatrix} 0{,}25 & 0{,}5 & 0{,}25 \end{bmatrix} \quad (2.3.)$$

2.1.2 Konvolucija slike separabilnim filtrom

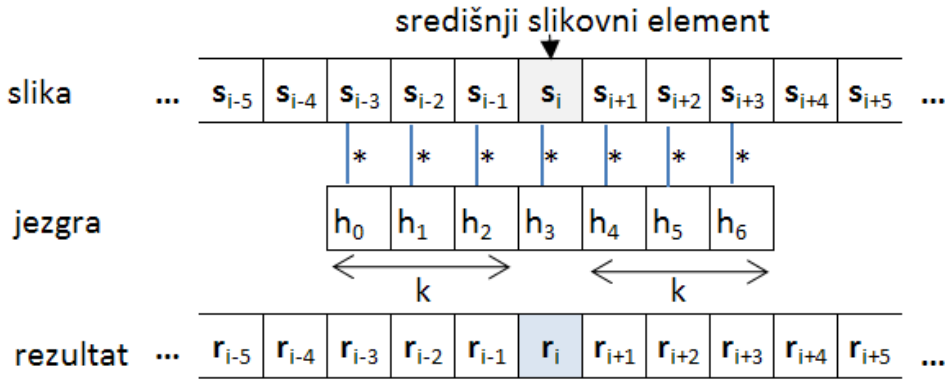
Ako je korištena konvolucijska jezgre separabilna, konvoluciju nad slikom možemo podijeliti na dvije jednodimenzionalne konvolucije. Jednodimenzionalna konvolucija se provodi odvojeno nad retcima izvorne slike te potom nad njenim stupcima kako bi rezultat odgovarao

dvodimenzionalnoj konvoluciji nad izvornom slikom. Uobičajeno je da konvolucijska jezgra ima neparan broj elemenata.

Vrijednosti svakog element izvorne slike zajedno sa vrijednostima njegovih susjednih elementima množe se sa vrijednostima odgovarajućih elemenata konvolucijske jezgre. Dobiveni umnošci se sumiraju i zapisuju na odgovarajuću poziciju odredišne slike prema jednadžbi (2.4.):

$$R[i] = \sum_{j=0}^{2k-1} s[i-k+j] * h[j] \qquad (2.4)$$

U jednadžbi (2.4), i je indeks trenutno računatog elementa odredišne slike, a k je polovica veličine konvolucijske jezgre.



Slika 2. Jednodimenzionalna konvolucija

U primjeru sa slike 2. za izračunavanje svakog slikovnog elementa odredišne slike izvršit će se $2 * k + 1$ operacija, gdje je k jednak polovini duljine zadane konvolucijske jezgre.

2.2 Aritmetika s nepomičnim zarezom

U računarskoj znanosti, brojevi s nepomičnim zarezom se koriste kod prikaza realnih brojeva koji imaju točno određen broj decimalnih mjesta. Brojevi s nepomičnim zarezom su jako korisni kod prikaza razlomaka, uobičajeno u bazi 2 ili bazi 10, kada izvršavajući procesor nema matematički koprocesor (engl. FPU - Floating Point Unit), ili kada aritmetika s nepomičnim zarezom značajno ubrzava izvođenje programa.

Broj s nepomičnim zarezom je po definiciji cijeli broj skaliran za određeni faktor. Na primjer, realni broj 1.23 se može prikazati kao 123/100 gdje je faktor skaliranja 100.

Kod zbrajanja i oduzimanja brojeva s nepomičnim zarezom dovoljno je zbrojiti tj. oduzeti cjelobrojni dio broja. No kod množenja i dijeljenja, rezultat je potrebno skalirati – za množenje, rezultat je potrebno podijeliti faktorom skaliranja, dok kod dijeljenja se rezultat mora pomnožiti faktorom skaliranja. To možemo prikazati jednadžbom $aS \cdot b = abS$, gdje bi rezultat trebao biti ab, stoga produkt treba podijeliti sa S.

U ovom radu se koriste 16 bitni binarni brojevi s nepomičnim zarezom, gdje će viših osam bitova predstavljati cjelobrojni dio, a nižih osam bitova decimalni dio broja. Kao usporedbu koriste se 32 bitni brojevi s pomičnim zarezom kod kojih matematički koprocesor određuje cjelobrojni i decimalni dio broja.

2.3 Vektorski instrukcijski podskupovi za arhitekturu x86

Pojam x86 se odnosi na instrukcijske skupove bazirane na arhitekturi procesora Intel 8086. Pojam je nastao iz činjenice da su imena procesora, koji su bili kompatibilni s procesorom Intel 8086, također završavala sa „86“.

Među mnogim instrukcijskim skupovima iz obitelji x86 nalaze se i vektorske instrukcije, tj. SIMD (Single Instruction, Multiple Data) instrukcije. Vektorske instrukcije se koriste u situaciji kada nad više podataka treba izvršiti jednu instrukciju. Takav paralelizam nad podacima ubrzava izvršavanje algoritama.

2.3.1 Proširenje MMX

MMX je proširenje instrukcijskog skupa koje je Intel uveo u 1996. godini. Proširenje MMX uvodi 64 bitni vektor sa cjelobrojnim vrijednostima kao novi tip podatka. Proširenje MMX uvodi u arhitekturu osam novih registara (od MMX0 do MMX7), te 57 novih instrukcija. Glavna namjena proširenja MMX je koncept pakiranih tipova podatka (engl. packed data types), što znači da

umjesto korištenja cijelog registra za jedan 64 bitni broj, registar se može koristiti za dva 32 bitna broja, četiri 16 bitnih brojeva ili osam 8 bitnih brojeva.

No, kod proširenja MMX se pojavio problem jer su novi registri zapravo bili postojeći 80 bitni registri matematičkog koprocesora. Zbog toga je bilo dosta teško raditi sa brojevima s pomičnim zarezom i MMX tipovima podataka u istoj aplikaciji.

2.3.2 Proširenje SSE

Intel se pozabavio problemima proširenja MMX kroz proširenje SSE (Streaming SIMD extension), gdje je uveo 128 bitne vektorske registre te podršku za brojeve s jednostrukom i dvostrukom preciznošću.

SSE je proširenje instrukcijskog skupa koje je Intel dizajnirao i uveo 1999. godine u svojim procesorima Pentium III. Proširenje SSE uvodi novih osam 128 bitnih registara, te 70 novih instrukcija.

Novih osam 128 bitnih registara (od XMM0 do XMM7) su potpuno odvojeni registri, ne kao kod proširenja MMX, te se time omogućuje istovremeno računanje vektorskim operacijama i brojevima sa pomičnim zarezom.

Kao i kod proširenja MMX, proširenje SSE koristi koncept pakiranog tipa podatka. Budući da su ovdje registri dvostruko dulji nego kod proširenja MMX, u proširenju SSE se može u jedan registar smjestiti:

- četiri 32 bitna realna broja sa jednostrukom preciznosti
- dva 64 bitna realna broja sa dvostrukom preciznosti
- dva 64 bitna cijela broja
- četiri 32 bitna cijela broja
- osam 16 bitna cijela broja
- šesnaest 8 bitna cijela broja

2.3.3 Dodatna proširenja

Proširenje SSE2, predstavljen sa Pentium IV procesorima, proširuje MMX instrukcije kako bi se izvršavale sa XMM registrima, te time izbjegavali registre matematičkog koprocesora. Proširenjem se dobivaju nove instrukcije za 64-bitne brojeve sa dvostrukom preciznošću.

Proširenje SSE2 uvodi sveukupno 144 novih instrukcija koje rade kako za realne brojeve tako i za cijele brojeve, te se time dobiva puno bolja pokrivenost za aritmetiku s nepomičnim zarezom.

Proširenje SSE3 dodatno proširuje instrukcije SSE2. Uvodi se 13 novih instrukcija za rad nad podacima, uglavnom multimedijalnih podataka (konverzija brojeva sa pomičnim zarezom u cjelobrojne vrijednosti, horizontalne operacije, sinkroniziranje dretvi, itd.)

Proširenje SSE4, uvedeno 2009. godine, donosi nove 54 instrukcije. Za razliku od prijašnjih proširenja SSE, proširenje SSE4 donosi instrukcije koji nisu namijenjene za korištenje u multimedijske svrhe (uspoređivanje tekstualnih tipova podataka, kondicionalno kopiranje elemenata, zaokruživanje brojeva, te prebrojavanje bitova).

Buduća proširenja uključuju proširenje SSE5, koje bi trebalo krenuti u produkciju 2011. godine, proširenje AVX (engl. Advanced Vector Extensions), koje bi trebalo uvesti 256-bitne vektorske registre, proširenje FMA (engl. Fused Multipy-Add), te druge.

3. Razvojno okruženje

Za pisanje programa korišten je razvojni program Eclipse. Eclipse je program otvorenog koga (engl. open source) koji je primarno namijenjen kao razvojno okruženje (engl. IDE - Integrated Development Environment) za programski jezik Javu. Kako bi se Eclipse mogao koristiti za pisanje programa u programskom jeziku C++, potrebno je instalirati CDT (C/C++ Development Tools), dodatak za Eclipse. Nadalje, potrebno je instalirati prevoditelj za programski jezik C++. Da bi se Eclipse,unutar operacijskog sustava Windows, povezao sa prevoditeljem gcc (GNU Compiler Collection) potrebno je instalirati pomoćni program MinGW (Minimalistic GNU for Windows).

Za izvedbu algoritama korišteno je programsko okruženje cvsh (engl. *computer vision shell*)[6]. 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.

3.1 Umetanje strojnog koda u programski jezik C++

Unutar koda pisanog u programskom jeziku C++ mogu se umetnuti dijelovi asemblerskog koda (engl. Inline assembler). Ovakav način umetanja koda pisanog u nižem programskom jeziku omogućava korištenje instrukcija proširenja 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__  ("instrukcija"
              : izlazna_vrijednost
              : ulazna_vrijednost
              : korišteni_registri
    );
```

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 koriste 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

Primjer korištenja modifikatora:

```
__asm__  (  "mov %0,%%xmm1"
            „mov %1,%%xmm2“
            „addpd %%xmm1,%%xmm2,%%xmm3“
            „mov %%xmm3,%2“
            : „=m“(izlazna_variabla)
            : „m“(prva_variabla),“m“(druga_variabla)
            : „%xmm1“,“%xmm2,“%xmm3“
    );
```

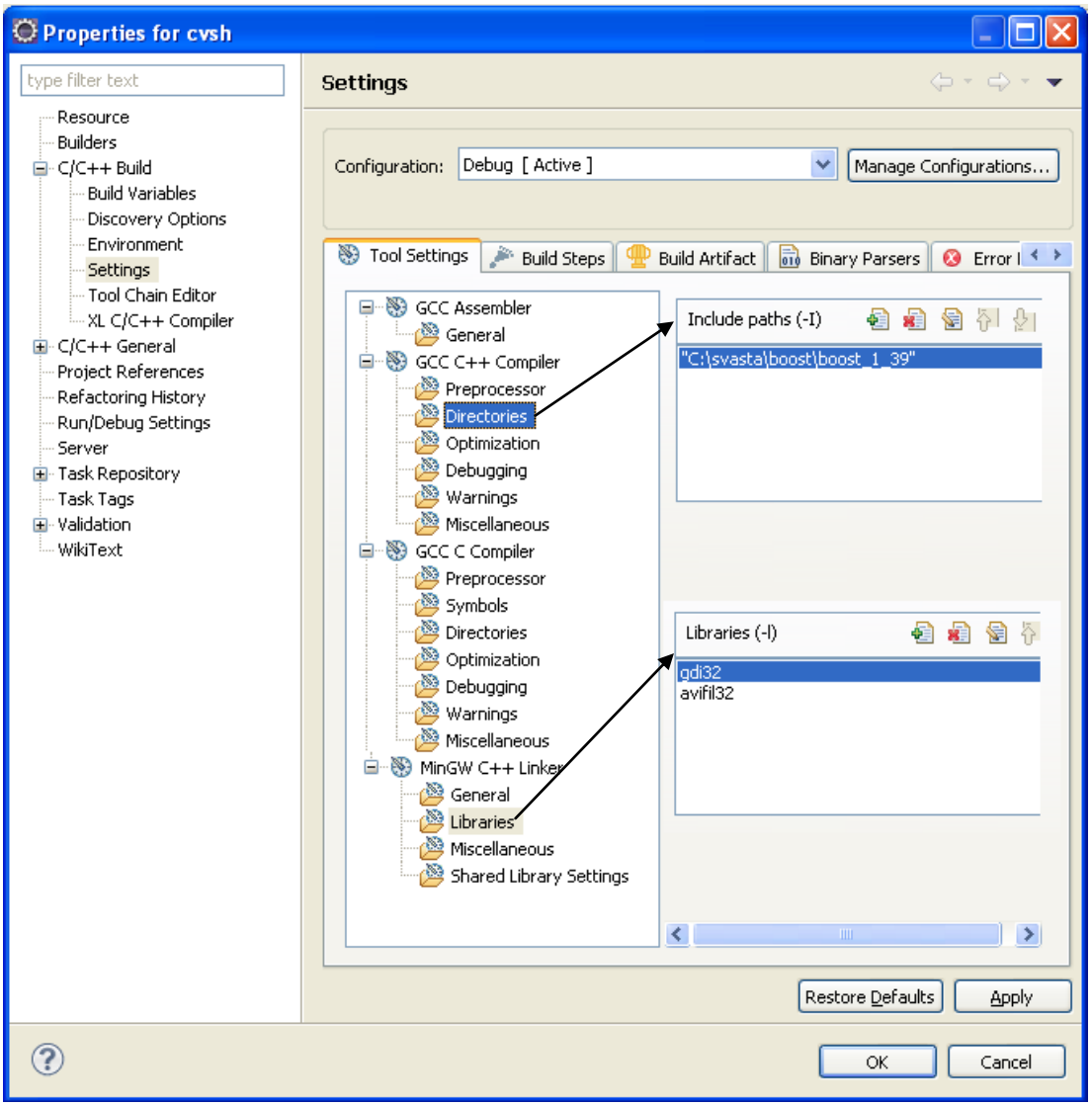
3.2 Podešavanje razvojnog okruženja

Prvo je potrebno preuzeti Eclipse sa http://www.eclipse.org (kod pisanja algoritama korištena je verzija 3.5 (Galileo)). Zatim je potrebno preuzeti i instalirati najnoviju verziju CDT-a (http://www.eclipse.org/cdt). Potrebno je isto tako sa stranica http://www.mingw.org preuzeti MinGW. Uz osnovni paket MinGW-a dolazi sve potrebno za pisanje i izvođenje programa u programskom jeziku C++. Ako se želi i pratiti izvođenje programa potrebno je preuzeti GNU Debbuger za MinGW. Naposljetku je potrebno dodati put do MinGW-a unutar varijable PATH.

Da bi se okruženje cvsh moglo prevesti, potrebno je sa stranica http://www.boost.org preuzeti biblioteku „boost“.

Podešavanje postupka prevođenja cvsh:

- stvoriti novi C++ projekt unutar Eclipse
- prebaciti daoteke iz cvsh_src unutar novo napravljenog projekta
- u projekt dodati stazu do direktorija s bibliotekom boost
    - Project → Properties → C/C++ Build →Settings →GCC C++ Compiler → Directories → Add… → staza_do_boost_direktorija
- isto tako dodati dodatne biblioteke u projekt
    - Project → Properties → C/C++ Build → Settings → MinGW C++ Linker → Libraries → Add…
    - potrebno dodati gdi32 i avifil32



Slika 3. Podešavanje postupka prevođenja cvsh

3.3 Pokretanje programa iz cvsh ljuske

Nakon uspješnog izvođenja funkcije „Build“, stvori se novi direktorij Debug u kojemu se nalazi izvršna datoteka (exe). Parametri potrebni za pokretanje cvsh ljuske su:

- -sf=“put_do_slike“ - put do izvorne slike
- -df=“put_do_slike“ - ako želite krajnju sliku pospremiti na disk
- -i - interaktivan način rada
- -a=zeljeni_algoritam - odabir željenog algoitma

Neke od naredbi koje se mogu zadati u interaktivnom načinu rada:

- c (conf) - mijenjanje konfiguracije programa
- p (process) - pokretanje algoritama nakon promjene konfiguracije
- q (quit) - izlaz iz programa

Uz ovaj rad priložen je CD sa dva programa. Ljuska cvsh u kojoj su dodani algoritmi konvolucije slike sa i bez korištenja proširenja SSE, te program koji testira algoritme konvolucije nad nasumičnim vektorima. Potrebno je pročitati datoteku README.txt u kojoj je objašnjeno korištenje programa cvsh i programa za testiranje konvolucije.

4. Primjena vektorskih instrukcija u implementaciji konvolucije

Jednodimenzionalna konvolucija opisana u 2.1.2 implementirana je korištenjem vektorskih instrukcija iz proširenja SSE3. Na ulazu se nalazi slika čiji su elementi veličine bajta, na izlazu se nalazi polje elemenata veličine 16 bita koji se smanjuju na veličinu bajta kako bi se stvorila odredišna slika. U jedan registar XMM veličine 128 bita može se spremiti osam 16 bitnih brojeva, tako da paralelno možemo računati s 8 elemenata slike. Zbog korištenja aritmetike s nepomičnim zarezom, konvolucijska jezgra se prije ikakvog računanja mora pripremiti tako da se svaki element pomnoži sa 2^8 i zaokruži kako bi se dobili cijeli brojevi. Poslije računanja i prije vraćanja rezultata u novu sliku, dobiveni broj se treba podijeliti sa 2^8 kako bismo dobili pravo rješenje.

$$\sum_{i=-k}^{k} h(i) \cdot 256 \cdot s(n+i) \;=\; r(n) \cdot 256 \qquad (4.1)$$

Kako je množenje najintenzivnija operacija konvolucije, u jednodimenzionalnim implementacijama konvolucije množenje se pokušava paralelizirati. Postoji više načina za ostvarivanje te paralelizacije, te time su nastale tri verzije implementacije.

U prvoj verziji implementacije prolazi se slijedno kroz elemente slike. Uzima se onoliko elemenata kolika je veličina konvolucijske jezgre te se međusobno množe pa potom zbrajaju kako bi se dobio rezultat. Dakle, jednim prolaskom se dobiva samo jedan element odredišne slike. Druga implementacija konvolucije koristi isti princip kao i prva, samo što ona koristi naredbu koja istovremeno množi i zbraja elemente. Također se dobiva samo jedan element odredišne slike jednim prolaskom kroz algoritam.

Treća verzija implementacije se uvelike razlikuje od prve dvije u tome što se ovdje paralelno izvodi isti korak konvolucije nad različitim podacima. Tako je osam različitih slikovnih elemenata pohranjeno u jedan XMM registar

te se istovremeno dobivaju 8 elemenata odredišne slike čime se dosta ubrzava izvođenje algoritma.

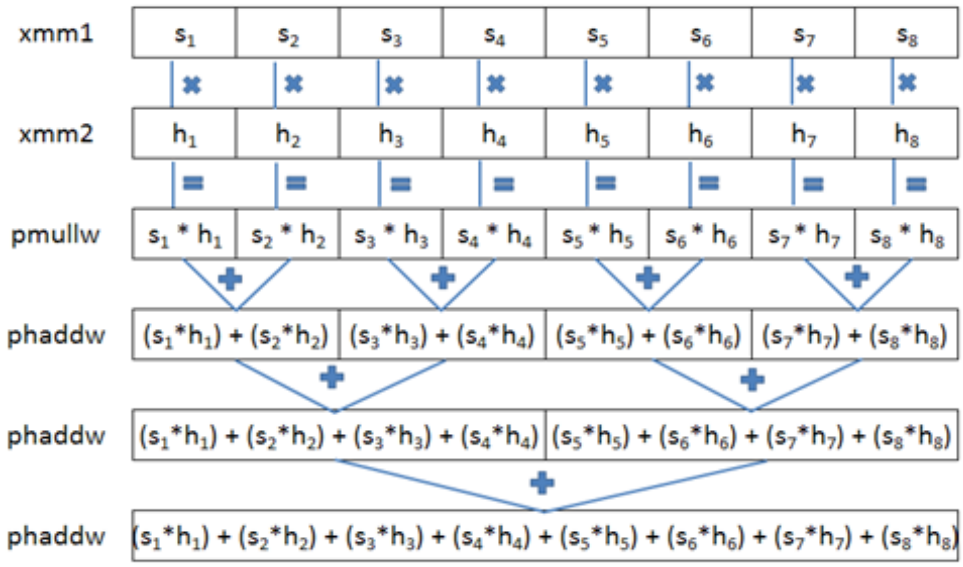
Potrebno je naglasiti kako se u prvoj i drugoj implementaciji paralelizira samo množenje elemenata, a u trećoj implementaciji je paralelizirano i množenje i zbrajanje elemenata.

4.1 Paralelno određivanje jednog izlaznog elemenata (prva verzija)

Implementacija je podijeljena na dvije petlje. U prvoj, vanjskoj petlji krećemo se po elementima slike i petlja povećava svoj brojač elemenata za jedan jer jednim prolazom kroz tu petlju računamo samo trenutni središnji element. Pošto se u jednom XMM registru može pospremiti samo osam 16-bitnih brojeva, a veličina konvolucijske jezgre može biti i veća od osam, potrebna nam je druga, unutarnja petlja koja prolazi po elementima konvolucijske jezgre. Svaki element slike tada množimo odgovarajućim elementom konvolucijske jezgre pomoću funkcije *pmulhw* (Multiply Packed Signed Integers and Store Low Result), te kako bi dobili sumu svih produkta moramo tri puta iskoristiti funkciju *phaddw* (Packed Horizontal Add).

Slika 4. opisuje prvu verziju implementacije[2]:

[2] Impementacija odgovara drugoj implementaciji studentice Ozane Živković[5]



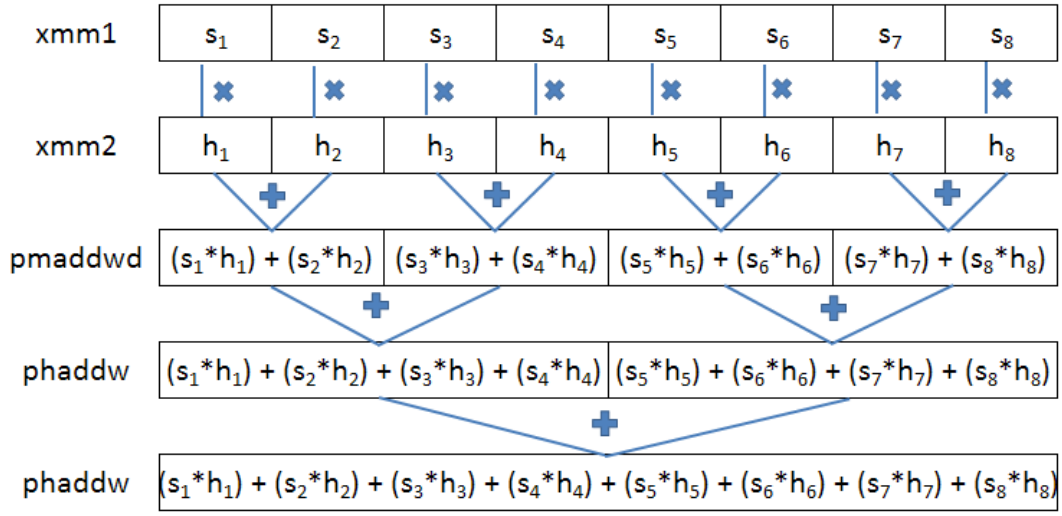
Slika 4. Paralelno određivanje jednog izlaznog elemenata, prva verzija

Slijedi kod opisanog algoritma:

```
int margina = velicinaJezgre / 2;
for(int i = margina; i < velicinaSlike - margina; ++i){
  najljevijiElement = i - margina;

    __asm__ volatile(
      "\n\t pxor %%xmm3,%%xmm3      \t# 0 -> xmm3"
      :
      :
      :"%xmm3"
    );
    for(int j = 0; j < velicinaJezgre; j += 8){

      __asm__ volatile(
        "\n\t movdqa %0,%%xmm1       \t# jezgra -> xmm1"
        "\n\t movdqu %1,%%xmm2       \t# slika -> xmm2"
        "\n\t pmullw %%xmm1,%%xmm2   \t# slika * jezgra"
        "\n\t phaddd %%xmm2,%%xmm2   \t# horizontalno zbraja"
        "\n\t phaddd %%xmm2,%%xmm2   \t# horizontalno zbraja"
        "\n\t phaddd %%xmm2,%%xmm2   \t# horizontalno zbraja"
        "\n\t paddw %%xmm2,%%xmm3    \t# niza 16 bita xmm2
                                              + xmm3 -> xmm3"
        :
        :"m"(jezgra[j]),"m"(slika[najljevijiElement + j])
        :"%xmm1", "%xmm2", „%xmm3"
      );
    }

    __asm__ volatile(
      "\n\t movd %%xmm3, %0     \t# rezultat -> odredisna slika"
      :"=m"(odredisnaSlika[i])
      :
      :"%xmm3"
  );
}
```

4.2 Paralelno određivanje jednog izlaznog elemenata (druga verzija)

Druga verzija implementacije je vrlo slična prvoj utoliko što imamo dvije petlje koje rade jednake stvari. Razlika je samo što umjesto naredbe koja množi elemente i konvolucijsku jezgru sada koristimo naredbu *pmaddwd* (Multipy and Add Packed Integers) koja istovremeno množi i zbraja elemente. Time smo htjeli smanjiti broj naknadnih horizontalnih zbrajanja i uštedjeti na vremenu izvođenja. Slika 5 opisuje drugu verziju implementacije algoritma:



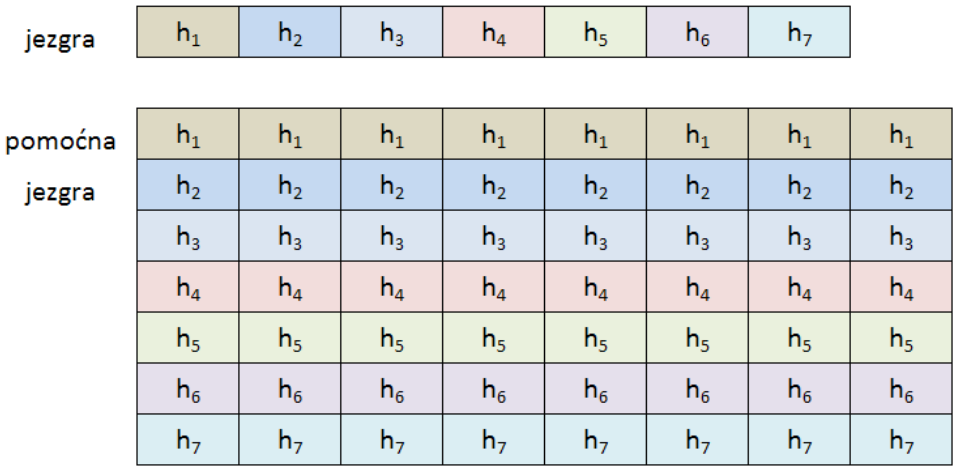
Slika 5. Paralelno određivanje jednog izlaznog elementa (druga verzija)

Slijedi dio koda implementacije gdje se ona razlikuje od prve:

```
__asm__ volatile(
  "\n\t movdqa %0,%%xmm1            \t# jezgra -> xmm1 "
  "\n\t movdqu %1,%%xmm2            \t# slika -> xmm2"
  "\n\t pmaddwd %%xmm1,%%xmm2       \t# istovremeno
                                    mnozenje i zbrajanje elemenata"
  "\n\t phaddd %%xmm2,%%xmm2  \t# horizontalno zbraja"
  "\n\t phaddd %%xmm2,%%xmm2        \t# horizontalno zbraja"
  "\n\t paddw %%xmm2,%%xmm3         \t# niza 16 bita xmm2
                                              + xmm3 -> xmm3"
  :
  :"m"(jezgra[j]),"m"(slika[najljevijiElement+j])
  :"%xmm1", "%xmm2", „%xmm3"
);
```

4.3 Paralelno određivanje osam izlaznih elemenata

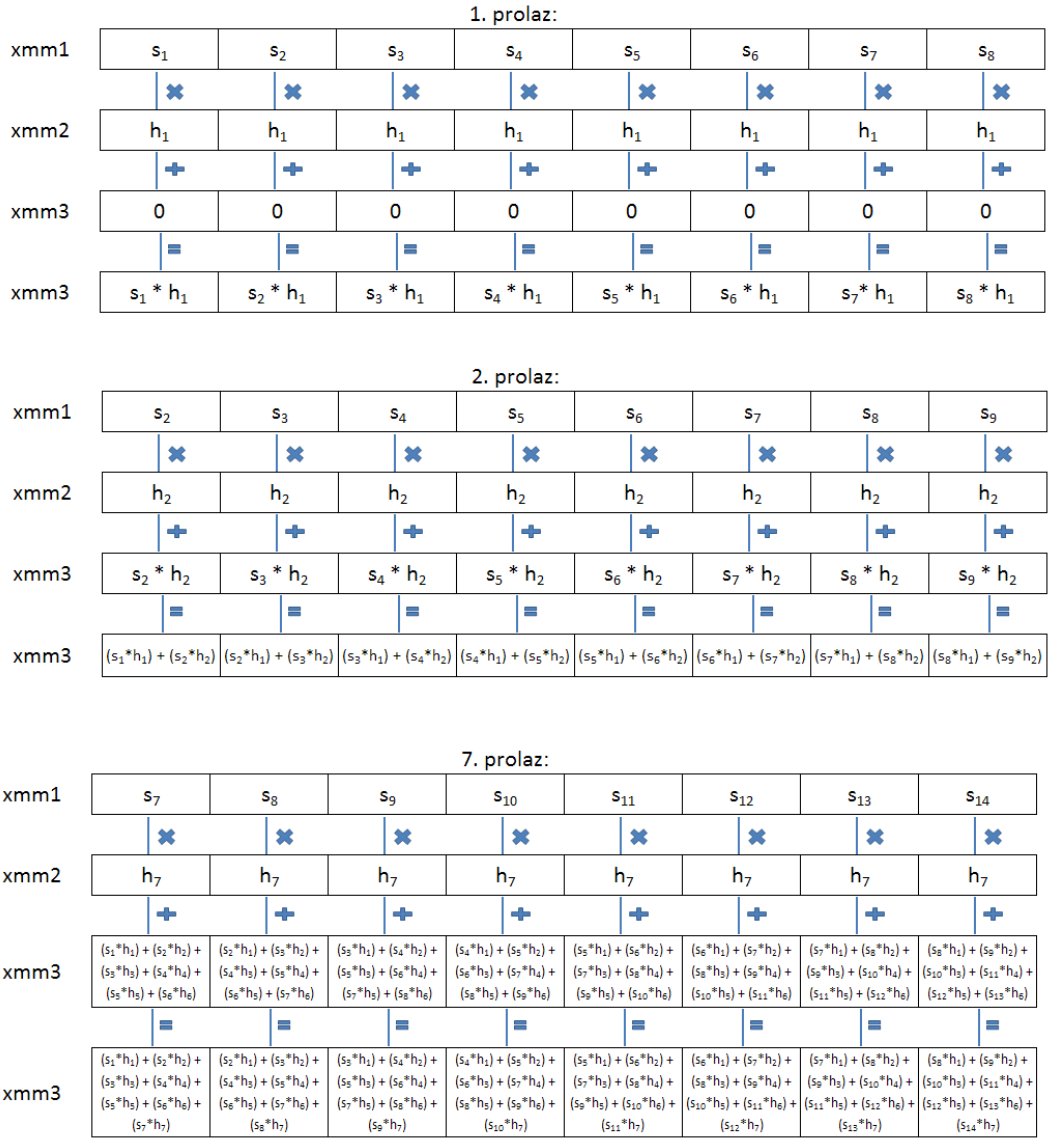
Treća implementacija koristi malo drugačiju logiku nego prve dvije. Naime, u njoj paralelno gledamo osam slikovnih elemenata kao središnje elemente. To se postiže tako da se brojač vanjske petlje povećava za 8, dok unutarnja petlja upravlja prolascima kroz elemente konvolucijske jezgre. Ideja je da se u jedan registar stavi 8 elemenata slike, dok se u drugi registar sprema jedan element konvolucijske jezgre ponovljen 8 puta.Nakon toga, ta se dva registra množe naredbom *pmullw* te zbrajaju sa trećim registrom naredbom *paddw* (Add Packed Word Integers). Da bi jedan element konvolucijske jezgre bio 8 puta kopiran, treba se memorija prije upotrebe pripremiti.



Slika 6. Stvaranje pomoćne konvolucijske jezgre

Nakon što se unutarnja petlja završi, u trećem registru ćemo imati završni rezultat osam elemenata koja pospremamo u odredišnu. Slika 7 grafički objašnjava ideju algoritma[3]:

[3] Implementacija odgovara trećoj implementaciji studentice Ozane Živković[5]



Slika 7. Paralelno određivanje osam izlaznih elemenata za jezgru od 7 elemenata

Kod algoritma glasi:

```
int margina = velicinaJezgre / 2;
for(int i = margina; i < velicinaSlike - margina; i += 8){
	najljevijiElement = i - margina;

	__asm__ volatile(
		"\n\t pxor %%xmm3,%%xmm3	\t# 0 -> xmm3"
		:
		:
		:"%xmm3"
	);
	for(int j = 0; j < velicinaJezgre; j++){
		__asm__ volatile(
		    "\n\t movdqa %0,%%xmm1 \t# pomocna jezgra -> xmm1"
		    "\n\t movdqu %1,%%xmm2 \t# slika -> xmm2"
		    "\n\t pmullw %%xmm1,%%xmm2  \t# slika*jezgra"
		    "\n\t paddw %%xmm2,%%xmm3   \t# slika*jezgra+xmm3"
		    :
		    :"m"(jezgra[j*velicinaSMIDVektora])  //%0
		    ,"m"(slika[najljevijiElement+j])     //%1
		    :"%xmm1", "%xmm2"
		);
	}

	__asm__ volatile(
		"\n\t movdqu %%xmm3, %0   \t# xmm3 -> odredisna slika"
		:"=m"(odredisnaSlika[i])     //%0
		:
		:"%xmm3"
	);
}
```

4.4 Ostali algoritmi korišteni u experimentiranju

Kako bi mogli uspoređivati implementacije konvolucije pisane uz pomoć proširenja SSE, napravljene su dvije implementacije koje koriste čisti programski jezik C++. U prvoj implementaciji koriste se samo 16 bitni cijeli brojevi (unsigned short) kako bi se mogla provjeriti točnost implementacija algoritma pisana asemblerom. U drugoj implementaciji se koriste 32 bitni brojevi s pomičnim zarezom (float) kako bi se mogli usporediti rezultati i vidjeti koliko se preciznosti gubi.

Izvorni kod algoritama:

```
void konvolucija_u_C (
      unsignes short *slika;
      int velicinaSlike;
      unsigned short *jezgra;
      int velicinaJezgre;
      unsigned short *odredisnaSlika;
){
  int margina = velicinaJezgre / 2;
  for(int i = margina; i < velicinaSlike - margina; ++i){
        unsigned short rezultat=0;

        for(int j = 0; j < velicinaJezgre; ++j){
            rezultat += slika[i-margina+j] * jezgra[j];
        }
        odredisnaSlika[i] = rezultat;
    }
```

Kod drugog algoritma su sve varijable float umjesto unsigned short.

Osim koda pisanog bez korištenja proširenja SSE, za usporedbu su korišteni algoritmi napisani od strane kolegice Živković. Njezini algoritmi koriste 32 bitne realne brojeve kod konvolucije slike.

5. Eksperimentalni rezultati

Testiranje je obavljeno na računalu specifikacija:

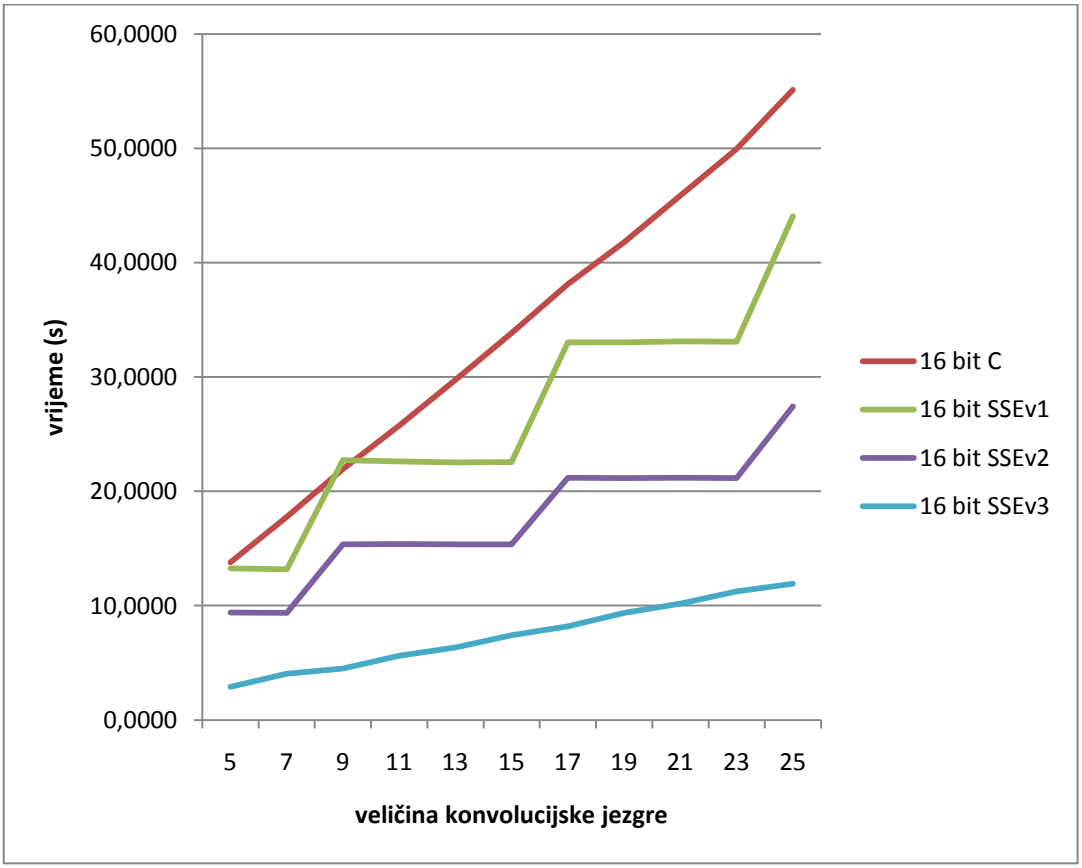
- Procesor: Intel® Core™ 2 CPU, T7200 @ 2.00GHz
- Memorija: 2048 RAM
- Operacijski sustav: Windows XP SP3

Kod kreiranja izvršnog programa kod je optimiziran sa –O3 optimizacijom.

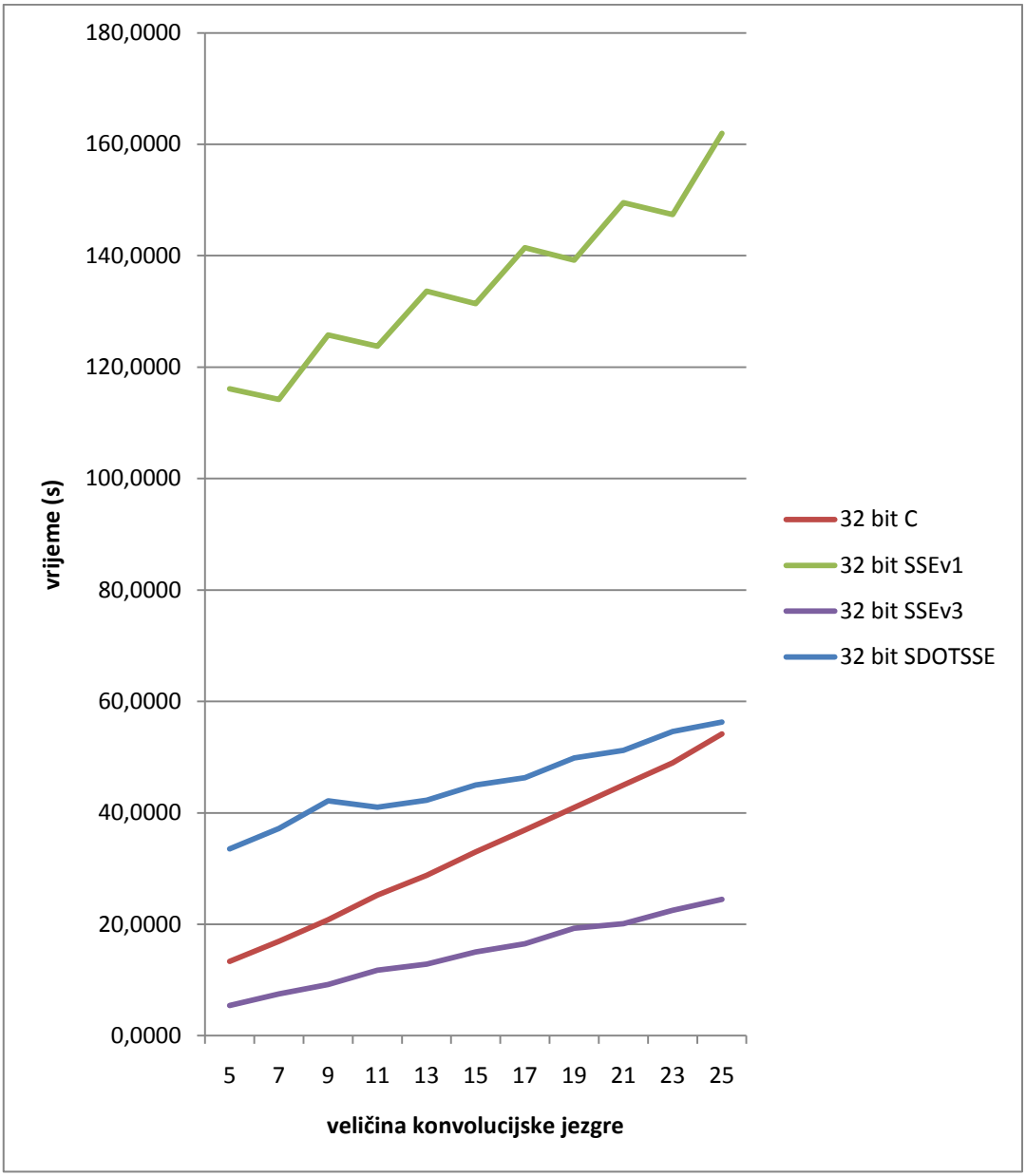
5.1 Testovi trajanja izvođenja programa

Slijedeći rezultati su dobiveni na ulaznom slučajnom polju veličine 10000 elemenata, gdje je svaki algoritam ponovljen 5 puta te su izvučene srednje vrijednosti.

Slijedeće tablice prikazuju dobivene rezultate:



Graf 1. Rezultati izvođenja algoritama sa 16 bitnim brojevima



Graf 2. Rezultati izvođenja algoritama sa 32 bitnim brojevima

Prvi graf prikazuje algoritme koji koriste 16 bitne brojeve, dok drugi graf prikazuje algoritme koji koriste 32 bitne brojeve. 16 bitne SSE verzije algoritma odgovaraju implementacijama opisanih u poglavlju 4. 32 bitne SSE verzije odgovaraju implementacijama opisanih u poglavlju 5. kolegice Živković[5]. Algoritam SDOTSSE koristi funkciju SDOT iz biblioteke mini-SSE-L1-BLAS [13].

Rezultati prikazuju kako se trajanje izvršenja algoritama pisanih u „čistom“ C-u linearno povećava sa povećanjem veličine konvolucijske jezgre. Može se primijetiti kako prve dvije 16 bitne SSE verzije algoritama stepeničasto rastu. Važno je primijetiti kako je trećoj verziji 16 bitnog SSE algoritma potrebno najmanje vremena za izvršenje, što je bio cilj ovog rada.

Uspoređujući 16 bitni algoritam pisan u C-u i treću verziju SSE algoritma, dobiva se ubrzanje od 4,5 puta, dok uspoređujući iste algoritme u 32 bitnim verzijama dobiva se ubrzanje od samo 2.2 puta.

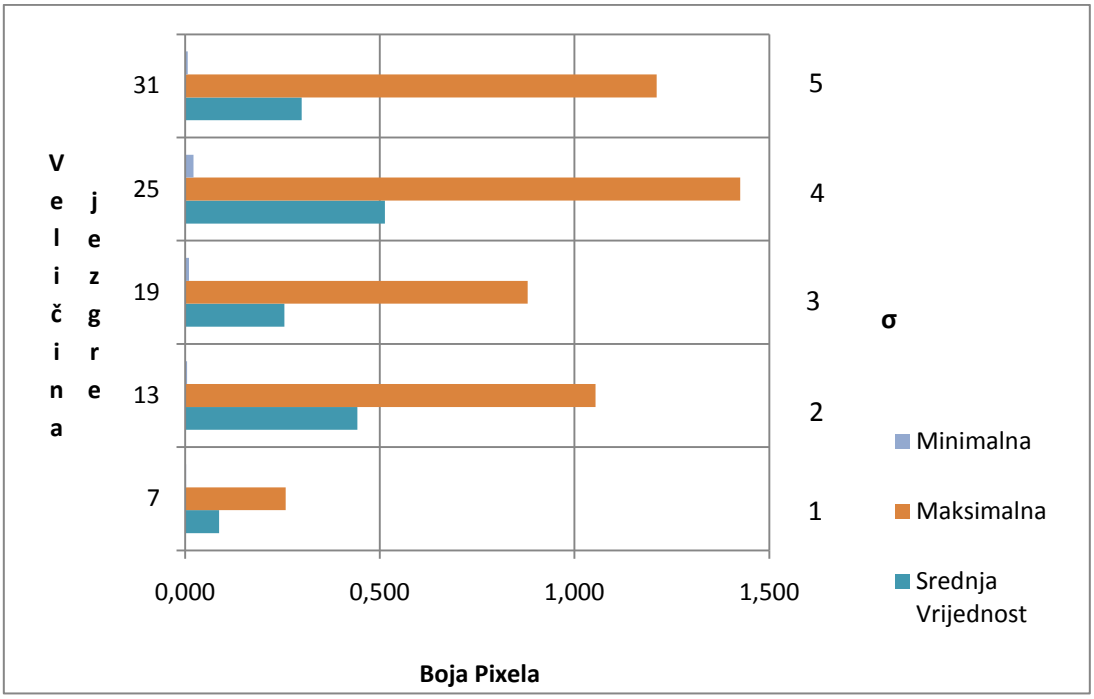
5.2 Testovi preciznosti

Druga vrsta testa je test preciznosti. Želimo utvrditi isplativost korištenja algoritma konvolucije s nepomičnim zarezom. Ti testovi su rađeni sa konvolucijskom jezgrom koja koristi Gaussovu razdiobu:

$$h(i) = \frac{1}{\sqrt{2\pi\sigma}} * e^{\left(-\frac{i^2}{2*\sigma^2}\right)}; -k < i < k$$

gdje je $k = veličina\ jezgre\ /\ 2$. Normalno se zadaje $k = 3 * \sigma$, no budući da radimo sa diskretnim brojevima razdioba neće biti pravilna jer će suma svih elemenata biti veća ili manja od 1. Da takva razdioba ne bi davala krive rezultate, treba obaviti male preinake. Svaki element konvolucijske jezgre je potrebno podijeliti sumom svih elemenata konvolucijske jezgre kako bi konačna suma bila jednaka 1.

Slijede rezultati koji su rađeni na slučajnim vektorima veličine 100, sa različitim σ:

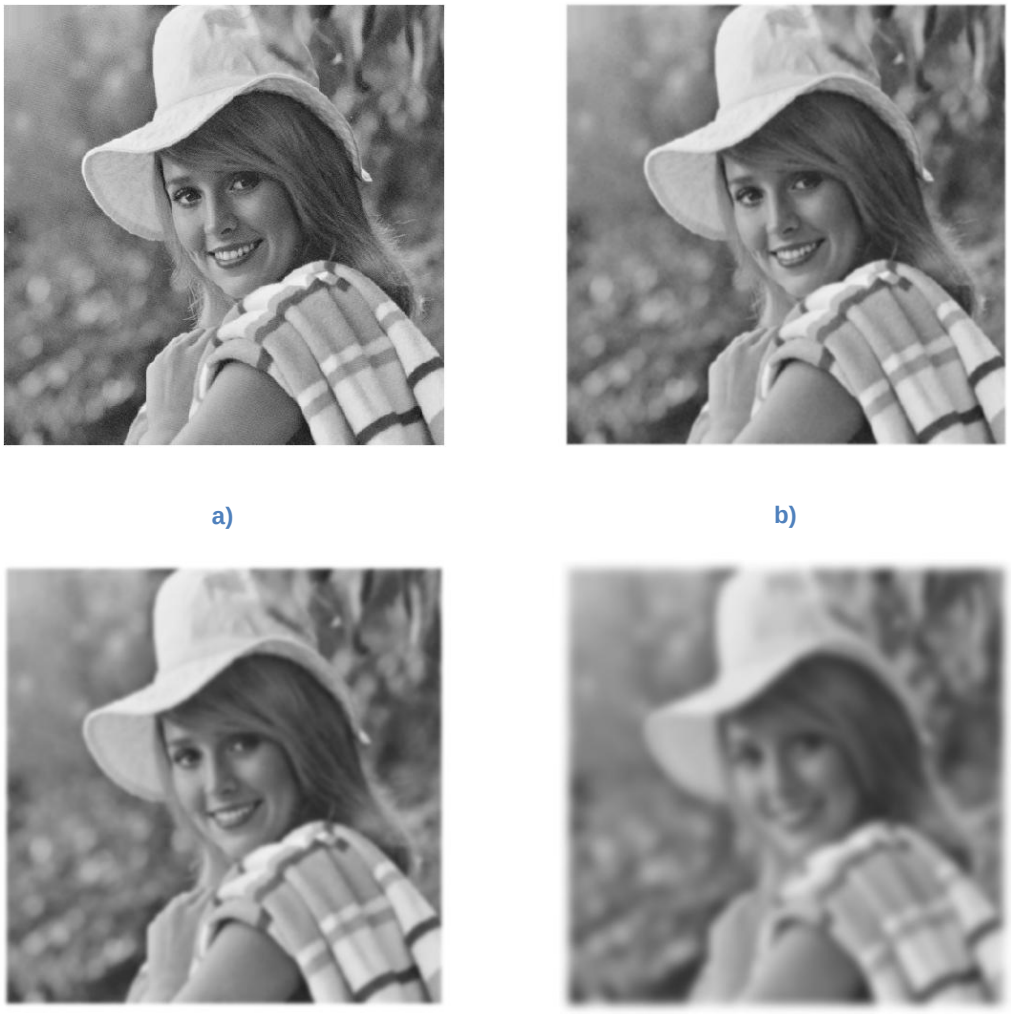


Graf 3. rezultati testova preciznosti

Graf prikazuje minimalnu, maksimalnu i srednju vrijednost razlike između rezultata dobivenih upotrebom konvolucije nad 16 bitnim i 32 bitnim brojevima. Ti rezultati su pogreške u nijansi boje jednog elementa. Kao što se vidi iz rezultata pogreške nisu velike, srednja pogreška je manja od 0,01%.

5.3 Primjeri konvolucije

Slika 8. a) prikazuje ulaznu sliku veličine 512x512 elemenata. Slika 8. b), c) i d) prikazuje izlaznu sliku nakon konvolucije s Gaussovim filtrom različitih σ.



a) b)

c) d)

Slika 8. a) ulazna slika, b) σ = 1, c) σ = 2, d) σ= 5

Na primjerima se može vidjeti kako povećanjem σ slika postaje sve mutnija, isto tako se povećanjem σ povećava i veličina konvolucijske jezgre zbog čega se sve više rubnih elemenata ne uzima u računanju, što se najbolje vidi na zadnjoj slici.

Razlika između algoritama koji koriste 16 bitne i 32 bitne brojeve je pokazana na slici 9.

a) b)

Slika 9. σ= 2, a) 16 bitni, b) 32 bitni

Iako postoje razlike u elementima izlaznih slika dvaju algoritama, kao što pokazuju testovi preciznosti, na slikama se ne primjećuju te razlike.

6. Zaključak

Iz ovog rada možemo vidjeti kako uporaba vektorskih instrukcija nad algoritmima obrade slike može ubrzati izvođenje algoritma. Za dodatno ubrzanje koristi se aritmetika s nepomičnim zarezom kako bi se izbjegao matematički koprocesor za računanje s realnim brojevima. Zbog korištenja cijelih brojeva u jedan XMM registar se može pospremiti do osam 16 bitnih brojeva, no zbog korištenja malog broja bitova dolazi do smanjenja preciznosti.

Usporedbom 16 bitnih algoritama uočava se ubrzanje od 4,5 puta (C16 vs SSE16). Uočava se i ubrzanje od 2 puta usporedbom 16 bitnog SSE i 32 bitnog SSE algoritma.

Korištenjem aritmetike s nepomičnim zarezom smanjena je preciznost dobivenih rezultata. Vrijednosti slikovnih elemenata izlazne slike kod algoritama koji koriste 16 bitne brojeve su za 0,1% veće ili manje nego kod algoritama koji koriste 32 bitne brojeve. No kako je ta razlika vrlo mala, na usporedbama se te razlike ne primjećuju.

7. Literatura

[1] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 1: Basic Architecture, rujan 2009. http://www.intel.com/Assets/PDF/manual/253665.pdf

[2] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2A: Instruction Set Reference, A-M, rujan 2009. http://www.intel.com/Assets/PDF/manual/253667.pdf

[3] Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2B: Instruction Set Reference, N-Z, rujan 2009. http://www.intel.com/Assets/PDF/manual/253667.pdf

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

[5] Ozana Živković, Optimiranje algoritama obrade slike vektorskim instrukcijama, Fakultet elektronike i računarstva, lipanj 2009.

[6] Craniev Detektor Rubova, Fakultet Elektronike i Računarstva, Projekt 2008/09

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

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

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

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

[11] Fixed-point arithmetic, Wikipedia, the free encyclopedia, http://en.wikipedia.org/wiki/Fixed-point_arithmetic

[12] GCC-Inline-Assembly, 01. Ožujak 2003, http://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html

[13] mini-SSE-L1-BLAS library: A fast library for SDOT,DDOT,SAXPY,DAXPY operations on x86 processor, 20. Siječanj 2010, http://www.applied-mathematics.net/miniSSEL1BLAS/miniSSEL1BLAS.html

[14] Richard Szeliski, Computer Vision: Algorithms and Applications, 10. Siječanj, 2010. http://research.microsoft.com/en-us/um/people/szeliski/Book/drafts/SzeliskiBook_20100110_draft.pdf

[15] Anil K. Jain, Fundamentals of DigitalI mage Processing, University of California, Pretence hall, 1986.

8. Sažetak / Abstract

U ovom radu razmatra se korištenje vektorskog proširenja SSE arhitekture x86 i aritmetike s nepomičnim zarezom u obradi slike. Takva kombinacija omogućuje brže izvođenje algoritama uz prihvatljive rezultate.

Algoritam konvolucije slike sa Gaussovim filtrom implementiran je u programskom jeziku C++ s umetnutim dijelovima koda pisanim u asembleru. Zbog svojstva Gaussove funkcije, takav algoritam se rastavlja na dvije jednodimenzionalne konvolucije. Prva konvolucija se provodi nad retcima slike, dok se druga konvolucija izvodi nad stupcima rezultata prve konvolucije.

Korištenjem vektorskih instrukcija proširenja SSE, obrada slikovnih elemenata se paralelizira čime se dobiva brža izvedba algoritama. Nadalje korištenjem aritmetike s nepomičnim zarezom u računanju konvolucije, zaobilazi se matematički koprocesor te se time ubrzava izračun uz prihvatljivo smanjenje kvalitete rezultata.

Ključne riječi: obrada slike, vektorska proširenja, aritmetika s nepomičnim zarezom, konvolucije slike, Gaussov filtar, XMM, SSE, SSE2

Optimization of image convolution using vector instructions in fixed-point arithmetic

This work considers the use of SSE vector instructions in image processing. In combination with fixed-point arithmetic, algorithms yields faster execution time with acceptable results.

Image convolution with Gaussian filter is implemented in programming language C++ with inline assembly. Properties of Gaussinan function allows algorithms to be separated into two one-dimension convolution. Image rows are first being processed, and then resulting image columns are put through the same algorithm.

Using vector instructions, processing of image pixels is being parallelized which speeds up algorithm execution. Furthermore, using fixed-point arithmetic in computation, floating point unit is not being used which speeds up calculation with acceptable quality loss.

Key words: image processing, vector extensions, fixed-point arithmetic, image convolution, Gaussian filter, XMM, SSE, SSE2