SVEUČILIŠTE U ZAGREBU

**FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA**

DIPLOMSKI RAD br. 543

# Programska implementacija PCA temeljenog biometrijskog sustava u CUDA okruženju

Goran Futač

Zagreb, lipanj 2013.

Ovdje se umeće izvorni tekst zadatka!!

Ovdje ostaje prazna stranica!!! Dakle, bez ovog teksta!

# Sadržaj

# 1 Uvod

Od pojave prvih računala čovjek se pita je li moguće i hoće li ikada uspjeti napraviti inteligentan stroj čija će se inteligencija moći mjeriti s inteligencijom čovjeka. Oko pojma inteligencije i koja njena svojstva bi stroj morao zadovoljavati da bi se smatrao inteligentnim vode i stručnjaci s područja računarske znanosti i filozofi, no svi će se složiti da bi inteligentan stroj morao moći raspozavati objekte i donositi zaključke o njima ili na temelju njih.

Jedan od posebno zanimljivih problema sa područja raspoznavanja uzoraka je identifikacija osoba na temelju nekih karakteristika koje su jedinstvene za svaku osobu. Konvencionalni načini identificiranja su identfikacija pomoću identifikacijskog dokumenta (npr. osobna iskaznica, putovnica) ili pomoću sigurnosnog ključa (npr. kod bankovnih transakcija), no te metode pate od nekih nedostataka.

Neke nedostake tih metoda rješavaju biometrijske metode identifikacije. Biometrijski sustav za identifikaciju nepoznate osobe provodi usporedbu *jedan-na-više* (eng. *one-to-many*) sa svim osoba čiji se biometrijski podaci nalaze u bazi podataka. Biometrijski podaci mogu biti slika lica, otisak prsta, potpis, termogram lica, hod itd.

Računalna identifikacija osobe na temelju biometrijskih karakteristika ne samo da je zanimljiva s teorijske strane nego ima i praktičnu primjenu, npr. u raznim sigurnosnim sustavima. Kriterij koji bi takav sustav trebao zadovoljiti je čim veća točnost koja se izražava kao omjer točno identificarnih uzoraka u odnosu na ukupan broj uzoraka. Osim visoke točnosti sustav bi trebao rezultat identifikacije vratiti u što kraćem vremenu. Brzina odziva takvog sustava uvjetovana je brzinom računala, ali i brojem korisnika u bazi podataka s kojima se nepoznati (u tom trenu) korisnik treba usporediti. S obzirom da se postupak usporedbe osobe s svakom od osoba u bazi provodi po nekom algoritmu, a isti algoritam se izvodi onoliko puta koliko ima različitih korisnika u bazi, postupak identifikacije je moguće paralelizirati. Paralelno programiranje je još do prije desetak godina bilo vezano gotovo isključivo za razne znanstvene i istraživačke projekte, a veći proboj na tržište doživjelo je pojavom višejezgrenih procesora kakvi se danas nalaze u

gotovo svakom osobnom računalu. Danas je osim paralelnog programiranja višejezgrenih procesora moguće programirati i grafičke procesore.

Pojavom Nvidijine *CUDA* tehnologije 2006. godine programiranje masivno paralelnih grafičkih procesora za neke opće namjene postalo je moguće svakome tko posjeduje *CUDA* grafičku karticu. *CUDA* grafički procesori su iz godine u godinu sve prisutniji u različitim granama znanosti poput biologije, meteorologije, fizike, računarstva pa čak i financija.

U ovom radu napravit će se usporedba implementacija biometrijskih sustava za identifikaciju osoba na temelju lica, na temelju dlana i na temelju kombinacije lica i dlana temeljenih na *PCA* metodi. Posebno će biti ispitane serijska implementacija i paralelna implementacija na *CUDA* grafičkom procesoru te će biti određena točnost svakog biometrijskog sustava. Dodatno, bit će ocijenjen i faktor ubrzanja paralelnih implementacija u odnosu na serijsku.

Nastavak rada je organiziran po poglavljima kako slijedi. U poglavlju 2 opisane su tehnologije i arhitekture koje su utjecale na razvoj *CUDA* grafičkih procesora. Poglavlje 3 govori općenito o paralelnom programiranju. U poglavlju 4 dan je detaljan opis arhitekture *CUDA* grafičkih procesora i najbitnije značajke *CUDA* programskog modela koje se koriste u paralelnoj implementaciji *PCA* temeljenih biometrijskih sustava. Poglavlje 5 objašnjava osnove biometrijskih sustava. U poglavlju 6 opisane su baze slika korištene u programskoj implementaciji. U poglavlju 7 opisuje se originalna metoda *PCA* kao i njezine dvije inačice. Poglavlje 8 sadrži kratak opis izgrađenih biometrijskih sustava i izvršenih eksperimenata. U poglavlju 9 se nalazi opis programske implementacije multimodalnih biometrijskih sustava. Poglavlje 10 sadrži analizu rezultata. U poglavlju 11 dan je zaključak rada.

# 2 Utjecajnije tehnologije i arhitekture

## 2.1 Razvoj supercomputinga

Revolucija računarstva kakvo danas poznajemo započela je pedesetih godina prošlog stoljeća s pojavom prvih mikroprocesora. Po današnjim standardima ti uređaji su bili veoma spori, sporiji i od mikroprocesora u današnjim mobilnim telefonima. Unatoč tome njihova pojava vodila je prema razvoju sve snažnijih i snažnijih procesora, a samim time i prema razvoju super – računala. Super – računala (eng. *supercomputers*) [4, 5, 6] je pojam koji označava najjača i najbrža računala kroz povijest. Njih većinom posjeduju svjetske vlade, akademske institucije ili velike korporacije. Koštaju milijune dolara, troše ogromne količine energije i zahtjevaju cijele timove stručnjaka da rade na njihovom održavanju.

Komercijalna, osobna računala približnih performansi mogu se očekivati kroz desetak godina [7]. Zanimljiv je podatak da je 2010. godine super-računalo temeljeno na Nvidijinim grafičkim procesorima bilo drugo najbrže računalo na svijetu [5, 7], s teoretski maksimalnih 3 petaflopsa (eng. *FLOPS – Floating Point Operation Per Second*) što je bilo više od tada najmoćnijih računala *IBM Roadrunner* i *Cray Jaguar*, a već 2011. godine Nvidia objavljuje da sa svojim grafičkim procesorima kreće u osvajanje titule najbržeg super – računala na svijetu.

Danas i super - računala i osobna računala nastoje povećati performanse kombinirajući snagu središnjeg procesora (eng. *Central Processor Unit, CPU*) i grafičkog procesora (eng. *Graphics Processor Unit, GPU*). Dva danas najveća globalna projekta koja koriste snagu grafičkih procesora su BOINC (nekomercijalna platforma za mrežno računarstvo) i Folding@home (distribuirani računalni projekt stvoren za kompleksne simulacije slaganja proteina). Oba projekta su distribuirani projekti koji za svoj rad koriste snagu tisuća računala diljem svijeta. Ti projekti običnim ljudima omogućuju da daju svoj doprinos konkretnim znanstvenim projektima. Doprinos od računala koja za izračune koriste i *CPU* i *GPU* daleko nadmašuje doprinos od računala koja koriste samo *CPU*. Zabilježen je podatak da je u studenom 2011. 5.5 milijuna računala zajedno imalo

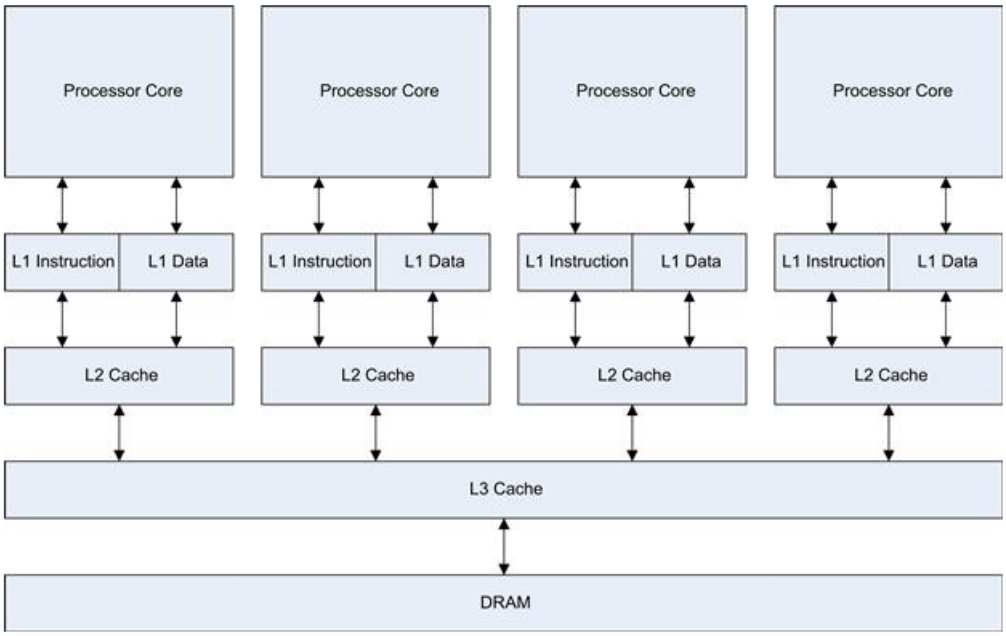
performanse od 5.3 petaflopsa, što je u tom trenutku bilo oko polovice od tada najjačeg super – računala na svijetu, japanskog *K-Computer* kojeg je proizvela tvrtka *Fujitsu* [7].

*Titan*, zamjena za najbrže američko super-računalo se sprema za 2013. godinu, a sačinjavati će ga gotovo 300 000 procesora i 18 000 grafičkih procesora, a trebao bi postizati između 10 i 20 petaflopsa. Taj podatak dovoljno govori sam za sebe – *GPGPU* (eng. *General Purpose Graphics Processor Unit*), odnosno programiranje grafičkih procesora za neke opće primjene uzima sve većeg i većeg maha. Usporedbe radi, početkom 2000. godine najmoćnije super – računalo je imalo 9632 *Pentium* procesora (*IBM ASCI Red supercomputer*) [7], dok danas slične performanse imaju osobna računala opremljena modernim grafičkim proceorima, a dostižu performanse od nekoliko teraflopsa. Ta činjenica nameće pitanje o performansama računala kroz sljedećih deset godina, no jasno je i da će se grafički procesori sve više i više koristiti, istraživati i razvijati, a rasti će i potreba za dobrim *GPU* programerima.

## 2.2 Von Neumannova arhitektura

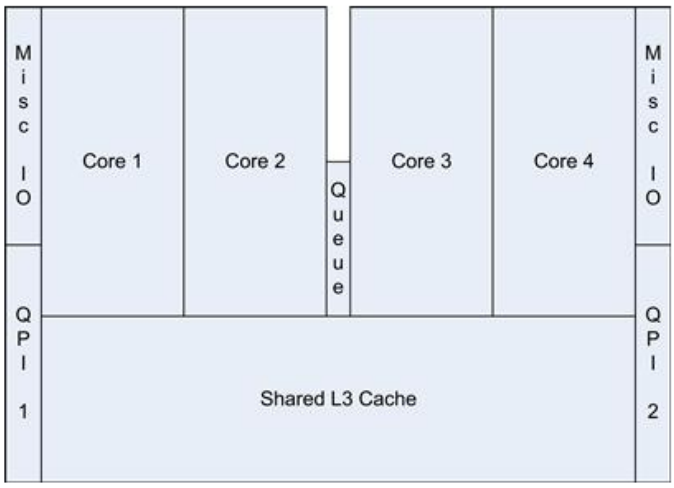
Von Neumannovu arhitekturu je važno spomenuti jer je ona služila kao temelj za razvoj ostalih arhitektura, odnosno njezin utjecaj seže sve do najmodernijih procesora. Von Neumannova struktura arhitekture podrazumijeva memoriju koja pohranjuje program i podatke, procesnu jedinicu koja izvodi aritmetičke i logičke operacije te upravljačku jedinicu kojoj je zadaća interpretiranje programa. U Von Neumannovom modelu procesor prvo dohvaća istrukciju iz memorije, a zatim izvršava , odnosno stalno se izmjenjuju faze „pribavi“ i „izvrši“. Grananje je ostvarno eksplicitnom promjenom programskog brojila (registar koji sadrži adresu sljedeće instrukcije). Glavni nedostatak Von Neumannove arhitekture je memorijsko usko grlo. Zbog dijeljene sabirnice procesor ne može u istom trenutku iz memorije dohvaćati i instrukcije i podatke zbog čega opada propusnost. Taj nedostatak djelomično rješava Harvarsdka arhitektura kod koje su podatkovna i instrukcijska sabirnica odvojene. Dodatno, odvojene su i podatkovna i programska memorija s time da se programska memorija može samo čitati što onemogućava da program izmjenjuje sam sebe.

Problem uskog grla između memorije i procesora prisutan je i danas. Najveći razlog tome je razlika u brzinama između procesora i memorije (današnji procesori rade na brzinama do 4GHz, a memorije do 2GHz [7]), čemu je pak razlog nezavisan razvoj procesorskih jedinica i memorija. Jedan od načina kojim se donekle može doskočiti tom problemu jest upotreba priručne memorije (eng. c*ache memory*). Priručnu memoriju je vrlo bitno spomenuti jer će se u narednim poglavljima ona promatrati u kontekstu arhitekture grafičkog procesora i programskog modela *CUDA*. Organizacija priručne memorije može se prikazati sljedećom slikom:




***Slika 1:*** **Organizacija priručne memorije [7]**

Ako se podatak kojeg procesor zahtijeva ne nalazi unutar prve razine priručne memorije (L1) isti se mora potražiti na sljedećoj razini ili u globalnoj memoriji. L1 priručna memorija obično radi na taktu jednakom ili bliskom taktu glavnog procesora pa je dohvat iz nje vrlo brz. No, L1 je obično vrlo malenog kapaciteta, obično 16 ili 32KB. L2 je nešto sporija, ali znatno veća, tipično 256KB. Neki procesori imaju i treću razinu prirčne memorije, L3, i ona je kapaciteta nekoliko megabajta, ali kao što je slučaj sa L2 u usporedbi sa L1, L3 je sporija od L2. Treba napomenuti da korisnost priručne memorije opada s njezinom veličinom – Intelov procesor *I7* ima 6 – 8 MB L3 priručne memorije i ona zauzima oko 30% površine procesora.




***Slika 2:*** **Raspored Intel I7 procesora [7]**

Kako raste kapacitet priručne memorije, raste i površina koju ona zauzima na čipu, s povećanjem čipa raste i cijena njegove proizvodnje, ali raste i vjerojatnost da će se u proizvodnom procesu dogoditi greška pa će se procesor morati otpisati ili prodavati kao trojezgreni ili dvojezgreni (sa onemogućenim neispravnim jezgrama).

## 2.3 Utjecajnija super – računala i arhitekture

U nastavku slijedi kratki pregled nekih najpoznatijih i najutjecajnijih super – računala i arhitektura kroz povijest. Bitno ih je spomenuti jer su neka od tih računala po prvi puta koristila koncepte koji se i danas koriste u modernim procesorima i računalnim sustavima. Dodatno, neka od tih računala su i otvorila vrata prema paralelnom programiranju, a samim time i prema razvoju raznih paralelnih arhitektura pa tako i razvoju aktualnih Nvidijinih *CUDA* grafičkih procesora koji su trenutno vrh tehnološke ponude.

### 2.3.1 Cray

Kao prvo pravo super – računalo ističe se *Cray – 1* iz 1976. godine koje je ime dobilo po Seymouru Crayu, jednom od članova tima *Cray Research*. Računalo je bilo povezano sa velikim brojem raznih kabela kojih je bilo toliko da su čak zapošljavali žene da ih spajaju jer su njihove ruke bile tanje od muških pa su se mogle lakše provlačiti kroz gomilu žica. To računalo je tipično radilo samo po nekoliko sati, a natjerati ga da radi cijeli dan smatralo se velikim uspjehom. *Cray – 1* koštao je gotovo 9 milijuna dolara i postigao je za to vrijeme ogromnih 160 megaflopsa. Uspredbe radi, Nvidijin grafički procesor iz *Fermi* serije ima teoretski maksimum od jednog teraflopsa. Nasljednik računala *Cray – 1* bio je *Cray – 2.* U odnosu na svog prethodnika nosio je znatna poboljšanja. Koristio je arhitekturu dijeljene memorije koja je bila raspodijeljena u banke koje su bile povezane s jednim, dva ili četiri procesora. Na temeljima te ideje nastali su današnji *SMP* (eng. *Symmetrical MultiProcessor*) sustavi u kojima više procesora dijeli isti memorijski prostor. Kao i mnoga računala u to vrijeme *Cray – 2* je podržavao vektorske instrukcije koje i danas postoji u obliku *SSE* i *MMX* ekstenzija. Računala pod imenom *Cray* postoje i danas, npr. već spomenuti *Cray Jaguar*.

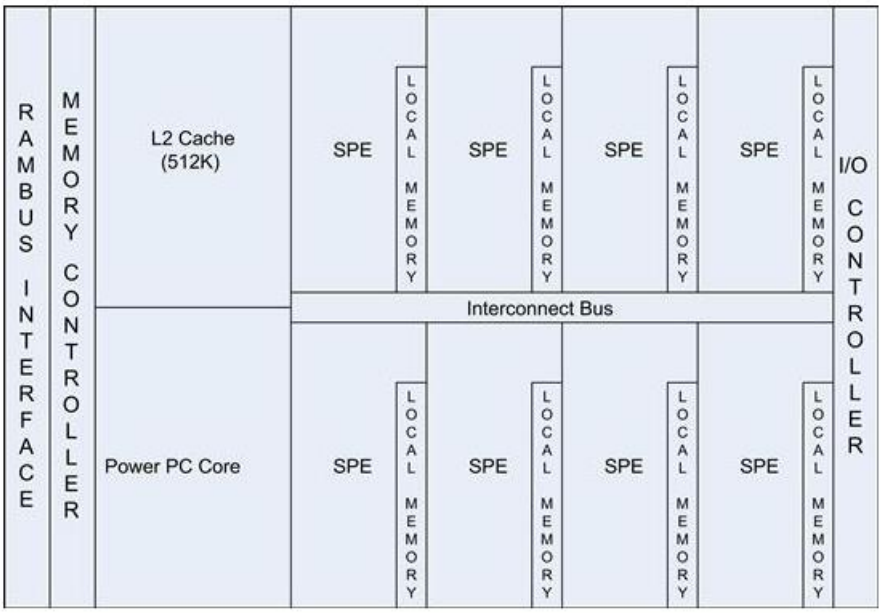
### 2.3.2 Connection Machine

*Connection Machine* je super – računalo koje je 1982. proizvela tvrtka *Thinking Machines.* To računalo je zanimljivo zbog svog inovativnog, ali u principu jednostavnog dizajna. Naime, stručnjaci koji su radili na dizajnu ovog računala stvorili su procesor sa 16 jezgri i takvih 4096 procesora su ugradili u jedan stroj. Umjesto jednog vrlo brzog procesora koji bi obrađivao sve podatke slijedno, ovdje je 65 536 procesorskih jedinica radilo svaka na svom dijelu podataka. Gledajući *Connection Machine* s obzirom na Flynnovu taksonomiju, to je *SIMD* (eng. *Single Instruction Multiple Data*) računalo. Moderni procesori *SIMD* način rada imaju ostvaren preko *SSE* i *MMX* ekstenzija. Kod *SIMD* procesora definira se raspon podataka koji će se obrađivati, a procesor tada primjeni istu operaciju nad cijelim nizom podataka. *Connection Machine* je izgrađen s ciljem da služi za

proučavanje umjetne inteligencije, ali kasnije verzije računala su imale više uspjeha s raznim znanstvenim proračunima i simulacijama. Problem kod *Connection Machinea* predstavljala je sinkronizacija.

### 2.3.3 Cell Processor

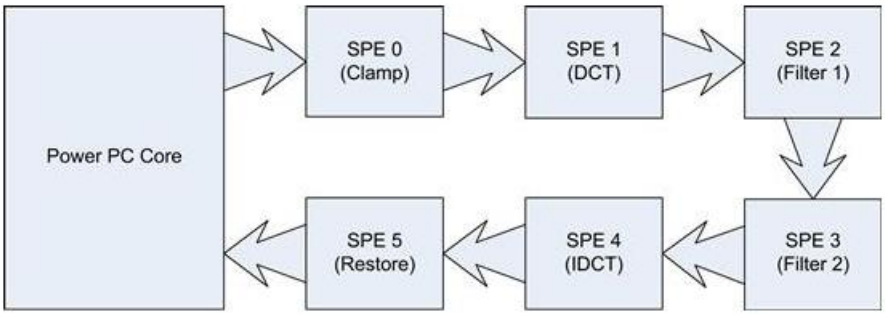
*Cell* je procesorska arhitektura koju je razvio *IBM*. Ideja je da u sustavu postoji jedan procesor koji djeluje kao „nadzornik“ brzim *stream* procesorima. Ulogu „nadzornika“ ima *Power Processor Element* (*PPE*) i on kontrolira osam *Synergistic Processor* elemenata (*SPE*) koj su *SIMD* procesori. *Cell* procesor je posebno zanimljiv iz razloga što je po dizajnu vrlo sličan Nvidijinom *G80* i kasnijim procesorima.




***Slika 3:*** **Organizacija *Cell* procesora [7]**

Programiranje ovakve arhitekture nije jednostavno. Programer je dužan napisati serijski kod koji će se izvoditi na *PPE,* ali i kod za svaki od osam *SPE* elemenata. Svaki *SPE* je zapravo jezgra za sebe, može izvoditi program nezavisan od onih koji se izvode na ostalim *SPE* elementima. Dodatno, svaki *SPE* može komunicirati sa svim ostalim

elementima (*PPE* i *SPE*) preko dijeljene sabirnice. Pri programiranju su moguća dva pristupa. Prvi je da svaki od *SPE* elemenata bude programiran da izvodi određenu fazu posla, a *PPE* daje podatke prvom *SPE* elementu i skuplja podatke od zadnjeg elementa, poput pokretne trake u tvornici gdje svaki radnik obavlja dio posla. Na ovaj način je osigurana protočnost, ali to uzrokuje i neke probleme. Kao glavni nedostatak ovog pristupa jest taj da je brzina cijelog procesa jednaka brzini najsporije faze, odnosno elementa. Ovaj problem je glavni problem bilo kakvog protočnog modela izvršavanja.




***Slika 4:*** **Tok operacija na *Cell* procesoru, protočni model [7]**

Alternativni pristup je da svaki *SPE* odradi kompletan posao, što bi bilo analogno tome da svaki radnik na pokretnoj traci odradi cijeli posao, npr. sastavi kompletan proizvod. Ovakav pristup nije dobar za složenije probleme zbog ograničene količine memorije kojom svaki *SPE* raspolaže, odnosno moguće je da problem neće „stati“ na *PPE*. Uz to, na ovaj način *PPE* mora dostaviti i sakupiti podatke od svih osam *SPE,* za razliku od protočnog modela kada mora komunicirati samo sa dva *SP* elementa.

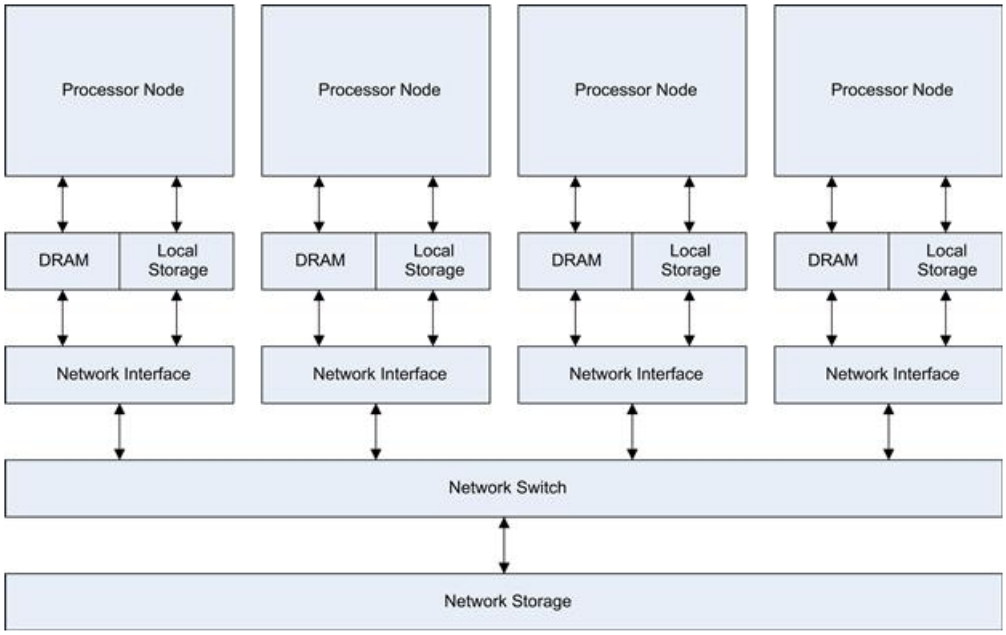
*IBM* je koristio *Cell* procesore u svom *Roadrunner* super – računalu koje je 2010. godine bilo treće najbrže na *Top500* listi. *Roadrunner* se sastojao od 12 960 *PP* i 103 680 *SP* elemenata. Ima performanse od teoretskih 1.71 petaflopsa, koštao je 124 milijuna američkih dolara, zauzima 560 $m^2$ i troši 2.35 MW električne energije.

### 2.3.4 Grozdovi računala (eng. *Clusters, Multinode computing*)

Popularnost grozdova računala narasla je devedesetih godina prošlog stoljeća. Ideja grozda je jednostavna – više računala se umreži i na taj način se dobije sustav sa vrlo dobrim performansama za manju cijenu nego da se kupovalo jedno računalo visokih performansi, odnosno, kombinirana snaga jeftinijih i sporijih računala nadmašuje snagu i izračunsku moć jednog skupog računala. Uzevši tu mogućnost u obzir, razne škole, sveučilišta i informatički odjeli mogli su nabaviti snažne sustave. Grozdovi računala su u to vrijeme bili poput *GPGPU* (eng. *General Purpose Graphics Processor Unit*) programiranja danas, tehnologija koja je promijenila svijet programiranja.

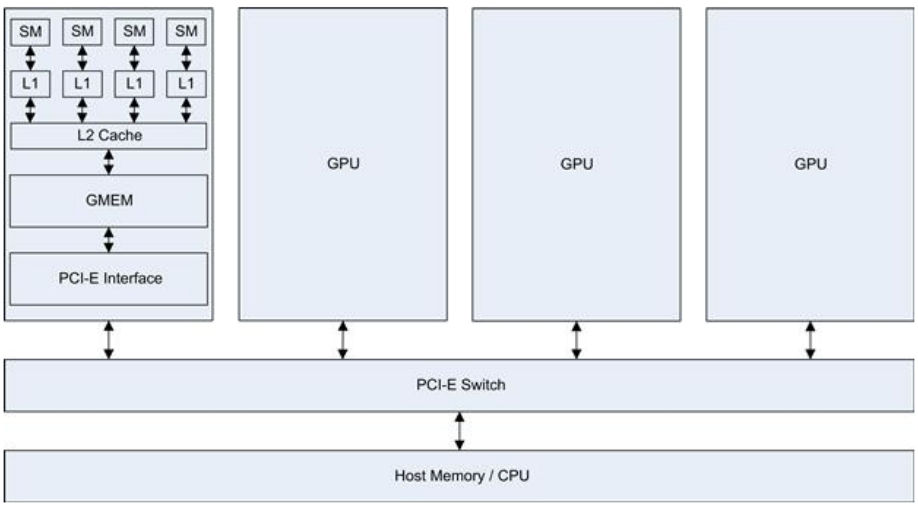
Iako je ideja grozda računala vrlo primamljiva (za relativno nisku cijenu može se napraviti sustav dobrih performansi), grozdovi, kao što je i sa većinom raznih tehnologija, pate od nekih nedostataka. Glavni problem je komunikacija između čvorova (eng. *node*) unutar grozda. Ako čvorovi ne moraju svi međusobno komunicirati problem ne dolazi do izražaja, no ukoliko svaki čvor mora primati od i slati svakom preostalom čvoru, puno vremena, u odnosu na utrošeno procesorsko vrijeme, će se trošiti na komunikaciju. Sličan problem može se javiti i prilikom programiranja grafičkog procesora, o čemu će se detaljnije govoriti u narednim poglavljima.

Kada pogledamo strukturu tipičnog grozda i usporedimo ju sa organizacijom priručne memorije tipičnog procesora (slika 1), možemo uočiti mnoge sličnosti. Ako svaku procesorsku jezgru gledamo kao čvor grozda, L2 priručnu memoriju kao RAM čvora, L3 priručnu memoriju kao mrežni komutator (eng. *switch*) na koji je čvor spojen, a RAM kao mrežni tvrdi disk, imamo minijaturni grozd.




***Slika 5:*** **Struktura tipičnog grozda računala [7]**

Arhitekura modernog grafičkog procesora nije mnogo dugačija – postoji više *streaming* multiprocesora (SM) koji su analogni sa jezgrama procesora opće namjene (*CPU*). Svaki SM je povezan sa dijeljenom memorijom (*eng. Shared memory*) i L1 priručnom memorijom, koja je pak povezana sa L2 priručnom memorijom. Podaci se čuvaju u globalnoj memoriji iz koje se mogu kopirati u globalnu memoriju računala ili preko *PCI – E* sučelja direktno na drugi *GPU* ako takav postoji. Prijenos preko *PCI – E* je puno brži nego prijenos mrežom.




***Slika 6:*** **GPU u usporedbi s grozdom [7]**

# 3 Paralelno programiranje

## 3.1 Razvoj paralelnog programiranja

Jedan od glavnih problema modernih procesora jest što je takt dosegnuo granicu od oko 4 GHz. Pri visokim taktovima stvara se puno topline te se zahtijevaju posebni i vrlo često skupi sustavi hlađenja jer s povećanjem takta procesora raste i potrošnja energije, a dodatno, kako se povećava količina topline koju procesor stvara uslijed povećanja takta, raste i potrošnja energije uzrokovana svojstvima silicija. Takvo neučinkovito korištenje snage obično znači da se procesor ne može adekvatno ili napajati ili hladiti, odnosno da se dostiže termalna granica.

Suočeni sa tim problemom, proizvođači procesora su krenuli drugim putem. Umjesto pokušaja da se takt procesora još više poveća odlučili su dodati više jezgri na jedan procesor pa tako danas većina osobnih računala ima dvije ili četiri jezgre, dok poslužitelji imaju još i više. No, nije sve ni tako jednostavno. Višejezgreni sustavi zahtijevaju da se programeri sa tradicionalnog serijskog, odnosno jednodretvenog pristupa programiranju prebace na višedretvene programe u kojima se sve dretve izvode istovremeno. Pojam dretva (eng. *Thread*) odnosi se na element u višedretvenom sustavu koji predstavlja najmanju nezavisnu cjelinu koja se izvodi. Programer sada mora razmišljati o većem broju dretvi, njihovoj međusobnoj interakciji i komunikaciji, što je daleko kompliciranije od jednodretvenog modela. Pojavom dvojezgrenih procesora prije nešto manje od deset godina stvar je još uvijek bila relativno jednostavna jer su programeri obično dio pozadinskog posla preselili na drugu jezgru. Dolaskom četverojezgrenih procesora mnogo programera je i dalje programiralo na staromodan način, odnosno programirali su klasične jednodretvene programe, a čak niti industrija igara nije odmah počela iskorištavati potencijal četverojezgrenih procesora, a to je jedna od industrija od koje bi se očekivalo da pokuša izvući maksimum od dostupne tehnologije. Postoje i ekonomski razlozi. Tvrtke koje se bave razvojem softvera proizvod na tržište trebaju plasirati čim prije, no iako su rješenja koja iskorištavaju mogućnosti četverojezgrenog procesora bolja i naprednije, neće puno vrijediti ako je konkurencija već ugrabila dio tržišta.

Velik broj današnjih programera je učio programirati u vrijeme kada su dominirali klasični serijski programi, a paralelno programiranje je bilo nešto egzotično što je privlačilo samo nekolicinu entuzijasta. Većina ljudi koja se odluči školovati na području računarstva odabire baš to područje zbog interesa prema tehnologiji. No, očekuju i da će nakon školovanja imati pristojna primanja, što najčešće znači da se trebaju specijalizirati za određenu granu struke. Izuzevši znanstvenike i istraživače, interes za paralelnim programiranjem je uvijek bio malen.

Danas je paralelno programiranje raspršeno, odnosno postoji mnogo različitih tehnologija i jezika koji nikad nisu u potpunosti komercijalno zaživjeli. Osim toga, nikada nije ni postojalo neko veće tržište paralelnih programa i tehnologija, što za posljedicu ima i malu potražnju za iskusnim paralelnim programerima. U prilog tome ide i činjenica da su proizvođači procesora svakih godinu ili dvije na tržište izbacili brži procesor te na taj način produljivali život klasičnih serijskih programa.

Za razliku od serijskih programa, paralelni programi su često vrlo usko vezani uz neku konkretnu arhitekturu. Želja za većim performansama često sa sobom povlači smanjenu prenosivost razvijenog programskog rješenja. Dodatno, ako je znanje i iskustvo programera vezano za neki konkretan procesor ili paralelnu tehnologiju, njegovo znanje će biti vrijedno samo dok je i taj procesor aktualan na tržištu.

Unatoč svim problemima kroz vrijeme je nastalo nekoliko standarda koji postoje i danas. *OpenMP* je dizajniran za višeprocesorske sustave s dijeljenom memorijom. Programiranje je relativno jednostavno jer se oko većine *low – level* stvari brine *OpenMP* sam. Drugi bitan predstavnik je *MPI (Message Passing Interface*) standard koji je namijenjen grozdovima računala. Često se koristi u ogromnim mrežama od po nekoliko stotina pa i tisuća računala gdje svaki čvor rješava dio problema. *OpenMP* i *MPI* se mogu koristiti i zajedno u svrhu što boljeg iskorištavanja paralelizma unutar čvora i između čvorova. Važno je još napomenuti da je glavna boljka svih mrežnih sustava brzina mreže, odnosno koliko brzo dva čvora mogu komunicirati.

*CUDA* može raditi u spoju sa obje navedene tehnologije.

## 3.2 Problemi i poteškoće paralelnog programiranja

Višedretveni programi sa sobom su donijeli i neke probleme. Glavni problem je dijeljenje sredstava. Taj problem se obično rješava nekim sinkronizacijskim mehanizmom, npr. semaforom, gdje dretva smije pristupiti nekom sredstvu ako i samo ako neka druga dretva već ne pristupa tom istom sredstvu. Problem se može javiti ukoliko dvije dretve, A i B (radi jednostavnosti) drže različite resures (npr. a i b) i pri tome dretva A želi do resursa b, a dretva B do resursa a, a pošto nijedna dretva ne može otpustiti sredstvo koje koristi prije nego što dođe u posjed traženog sredstva dolazi do potpunog zastoja (eng. *deadlock*). Sličan problem je i problem izgladnjivanja (eng. *starvation*) u kojem neke dretve nikada ne dođu na red za izvršavanje. Ukoliko pak nema sinkronizacijskog mehanizma, a dvije ili više dretvi pristupa nekom sredstvu mogući su nepredvidivi rezultati (eng. *race condition)*. Dužnost programera je voditi brigu o resursima i mogućim problemima jer su višedretveni programi vrlo često nepredvidljivi (u smislu da se ne može točno odrediti koji dio koda koje dretva će se u kojem trenutku izvoditi ukoliko nema pravilne sinkronizacije) pa otkrivanje grešaka može biti vrlo teško.

Alternativa dretvama su procesi. Proces se definira kao skup računalnih resursa koji omogućuju izvođenje programa. Proces se sastoji od barem jedne dretve, zasebnog adresnog prostora, adresnog prostora rezerviranog za pojedinu dretvu, stoga, kazaljke stoga, opisnika datoteka itd. Komunikacija između procesa se obično odvija slanjem poruka.

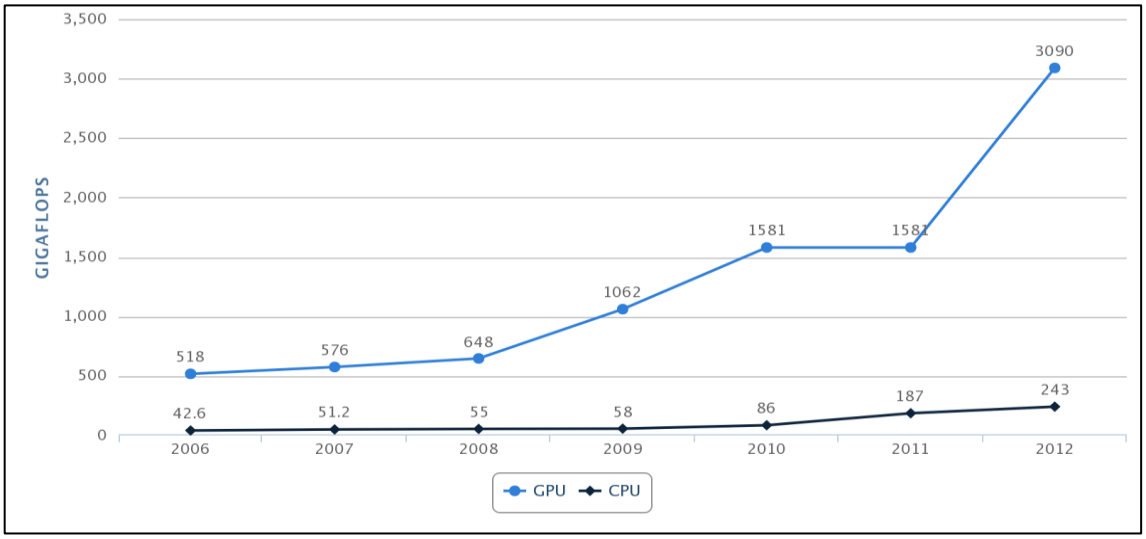
Po mnogočemu višedretveni model odgovara *OpenMP,* a višeprocesni model odgovara *MPI.*

## 3.3 Počeci *GPGPU* programiranja

Grafički procesori su prisutni u većini modenih osobnih računala. Središnjem procesoru pružaju osnovne operacije poput renderiranja slike u memoriji i prikaza iste na ekranu. Grafički procesor tipično obrađuje skup poligona koji predstavljaju scenu, na njih lijepi teksture i zatim provodi sjenčanje. Važan korak u razvoju grafičkih procesora je bio razvoj programirljivih *shadera*. *Shaderi* su u principu programi koje grafički procesor pokreće za izračun različitih efekata. Prvi korak prema programiranju grafičkih procesora za neke općenite primjene napravljen je kada je omogućeno manipuliranje *shaderima*, odnosno njihovo reprogramiranje. Ono što je još uvijek bilo ograničenje jest činjenica da su *shaderi* mogli raditi samo sa poligonima (koji su skupovi od po tri točke), odnosno sa skupom podataka gdje je svaki od podataka reprezentiran sa tri komponente. To ograničenje je samo potaknulo znanstvenike i istraživače da razviju grafički procesor koji će se moći programirati jednostavno poput središnjeg procesora računala. Razvijeno je nekoliko različitih modela od kojih je svaki imao svoje nedostatke - niti jedan nije bio lagan za naučiti niti ga je znao velik broj ljudi pa se niti u jednom trenutku nije dogodio proboj na veće tržište kao što je to situacija sa Nvidijinim *CUDA* programskim sučeljem.

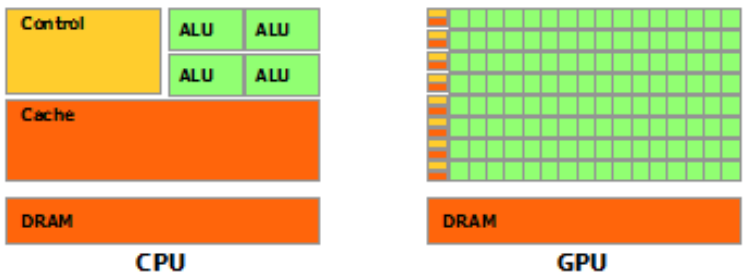
# 4 CUDA grafički procesori

Vođeni sve većim tržišnim zahtjevima za visoko kvalitetnom 3D grafikom, grafički procesori su evoluirali u masivno paralelne, višedretvene i višejezgrene procesore s ogromnom izračunskom moći i visokom memorijskom propusnošću.




***Graf 1*: Broj operacija s pomičnim zarezom na CPU i GPU (prema podacima iz[7])**

Sa grafa 1 je vidljivo da postoji raskorak u broju operacija s pomičnim zarezom između *CPU* i *GPU*. Razlog tomu je specijaliziranost grafičkog procesora za intenzivne i paralelne izračune, odnosno to što je najveći broj tranzisora namijenjen obradi podataka i izračunima, a manji dio priručnoj memoriji i kontroli toka (slika 7).



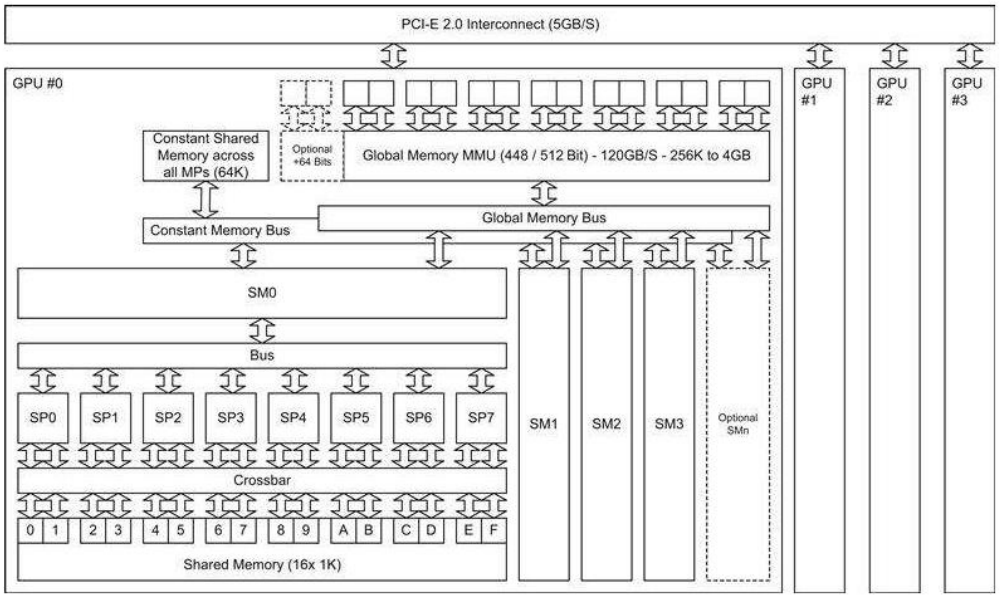

***Slika 7*: Ilustracija raspodjele broja tranzistora na CPU i GPU [16]**

Određenije, *GPU* je posebno dobar u rješavanju problema koji se mogu podijeliti u više manjih potproblema, tj. koji se mogu paralelizirati pa se isti program izvršava na puno podataka u paraleli (eng. *data level parallelism*), pri čemu je odnos broja aritmetičkih operacija naspram memorijskih operacija vrlo velik. S obzirom da se nad svakim podatkom izvodi isti program, zatjevi za sofisticiranom kontrolom toka su niski. Osim paralelizma na razini podataka postoji i paralelizam na razini zadataka (eng. *task level parallelism*) gdje se istovremeno izvršava više različitih funkcija (ili programa) nad istim ili različitim podacima. Najbolji primjer paralelizma na razini zadataka jest (moderni) operacijski sustav (primjerice *Windows 7*) gdje se različiti zadaci, odnosno procesi, izvršavaju na različitim procesorskim jezgrama.

## 4.1 Arhitektura CUDA grafičkih procesora

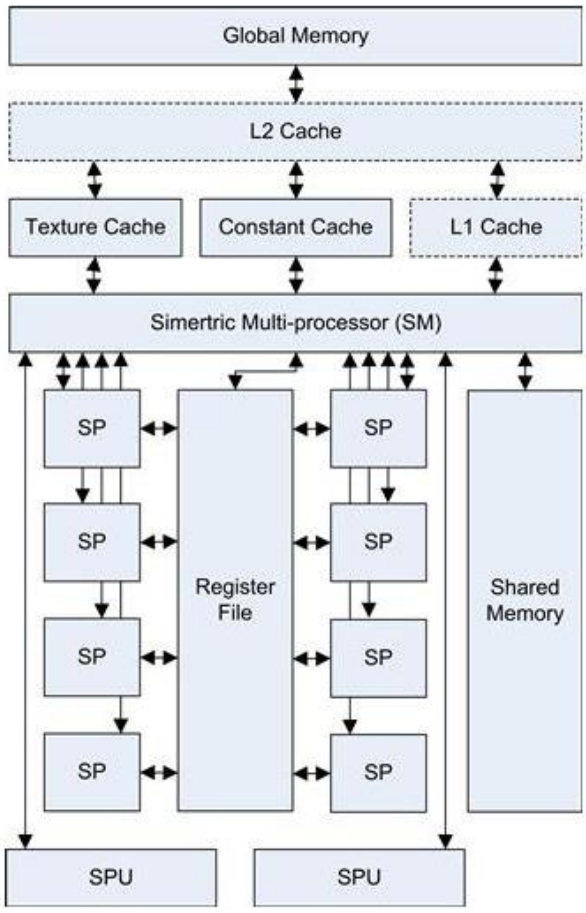
Slika 8 prikazuje arhitekturu *G80* grafičkog procesora, odnosno prvog *CUDA* grafičkog procesora. Bitno je primjetiti nekoliko ključnih blokova:

- Memorija – globalna, konstanta i dijeljena (eng. g*lobal, constant, shared memory*)
- *Streaming* multiprocesori (*SM*)
- *Streaming* procesori (*SP*)




***Slika 8*: Arhitektura G80 grafičkog procesora [7]**

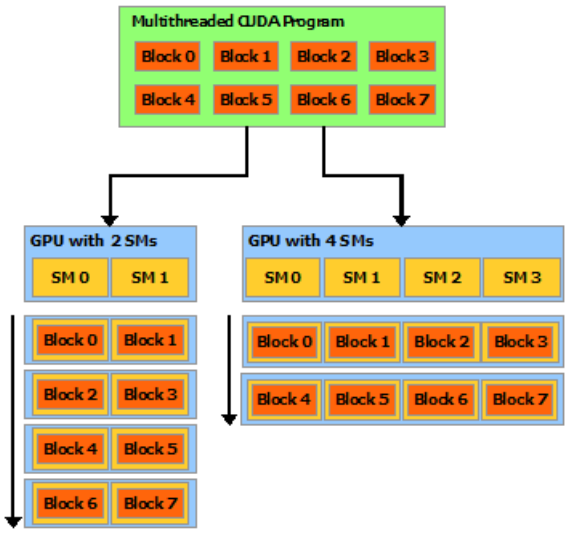
Najvažnija stvar za primjetiti je to da je grafički procesor zapravo polje *streaming* multiprocesora, od kojih svaki ima određen broj *streaming* procesora, ovisno o seriji (*G80* 8 *SP, Fermi* 32 – 48, 8 – 192 *Kepler*). Broj *SM* i *SP* jedinica je ključna stvar o kojoj ovisi skalabilnost procesora, odnosno to znači da ukoliko procesor ima više *SM* jedinica da će istovremeno moći izvršavati više zadataka ili isti zadatak izvršiti prije, pod uvjetom da se paralelizacija tog zadatka može skalirati na više.




***Slika 9*: Arhitektura jednog streaming multiprocesora [7]**

To se može objasniti i na sljedeći način - kao i kod središnjeg procesora, ako se programer ograniči na kod koji će iskorištavati samo dvije jezgre, zamjena procesora sa nekim četverojezgrenim procesorom neće donjeti nikakvo poboljšanje (ako zanemarimo brzinu

procesora) jer će se kod i dalje izvršavati samo na dvije od četiri dostupne jezgre. Kod *CUDA* procesora stvar je ponešto drugačija – za određenu paralelnu funkciju prije njenog pozivanja je moguće odrediti koliko dretvi će odrađivati isti posao, odnosno, puno lakše je skalirati isti problem na veći broj procesora, čak i bez modificiranja izvornog koda programa – grafički procesor će to učiniti sam. *Slika 10* grafički prikazuje spomenutu automatsku skalabilnost. Ukoliko imamo višedretveni program s određenim brojem blokova dretvi, procesor sa više *streaming* multiprocesora će prije završiti od onog s manjim brojem, i to sve bez uplitanja programera.




***Slika 10:*** **Automatska skalabilnost [16]**

Ako se detaljnije promotri slika 9 može se vidjeti slijedeće: svaki *SM* ima pristup određenom broju registara (*register file*) koji rade na brzini bliskoj brzini *streaming* procesora (*SP*), što znači da je čitanje iz njih i pisanje u njih vrlo brzo. Količina registara ovisi od generacije do generacije procesora pa tako *Kepler* serija omogućava svakoj dretvi korištenje maksimalno 255 registara za pohranu lokalnih varijabli, no taj broj može biti i manji, ovisno o broju dretvi unutar bloka. Potrebno je napomenuti da polja ne mogu biti smještena u registre, već samo skalarne varijable jer se registar ne može dinamički adresirati. Polja se smještaju u globalnu memoriju.

Dijeljena memorija (eng. s*hared memory*) je dio memorije koji je dostupan pojedinom *SM*. Čitanje u i pisanje iz dijeljene memorije je po brzini odmah nakon brzine pisanja u i čitanja iz registara. Pristup dijeljenoj memoriji imaju sve dretve iz jednog bloka. Dijeljena memorija djeluje kao programirljiva priručna memorija gdje programer određuje njen sadržaj.

Svaki *SM* je odgovoran za raspoređivanje vlastitih resursa, jezgara i ostalih jedinica, ima zasebnu sabirnicu prema memoriji tekstura, konstantnoj memoriji i globalnoj memoriji. Memorija tekstura je poseban pogled prema globalnoj memoriji i ona je korisna za podatke koji će se koristiti za interpoliranja. Konstanta memorija, kao što joj i ime kaže, se koristi za čuvanje nepromijenjivih podataka. Ona je također dio globalne memorije.

Dodatno, svaki od *streaming* multiprocesora ima i dvije ili više jedinice posebne namjene (eng. *Special purpose unit, SPU*) koje su sklopovska podrška za raličite matematičke operacije poput vrlo brzog izračunavanja sinusa, kosinusa, potenciranja i sl.

## 4.2 CUDA programski model

*CUDA* programski model preptostavlja da se *CUDA* dretve izvode na fizički odvojenom uređaju koji radi kao koprocesor središnjem procesoru računala koji izvodi program. Odvojeni uređaj, odnosno grafički procesor se po konvenciji naziva *device*, a središnji procesor *host*. Jednako se nazivaju i dijelovi koda koji se izvode na grafičkom procesoru, odnosno *device* kod, ili na središnjem procesoru, odnosno *host* kod. Dodatna pretpostvka je i da *host* i *device* sami održavaju stanje svojih odvojenih memorijskih prostora u *DRAM* memoriji, a koja se naziva *host* memorija (eng. *host memory*) i *device* memorije (eng. *device memory*). *Host* dio programa je zadužen za održavanje stanja *device* globalne memorije. To uključuje alociranje i dealociranje memorije te prijenos podataka iz *host* memorije u *device* memoriju i obrnuto.

### 4.2.1 Kernel

Kernel je paralelna funkcija koju će, kada se pozove, *N* dretvi izvršiti u paraleli na *CUDA* grafričkom procesoru.

```
// definicija kernela
__global__ void VecAdd(float* A, float* B, float* C){
    int i = threadIdx.x;
    C[i] = A[i] + B[i];
}

int main() {
    // ...
    // poziv kernela, N dretvi
    VecAdd<<<1, N>>>(A, B, C);
    // ...
}
```

***Kod 1*: Kernel koji zbraja dva vektora**

*CUDA* C jezik je proširenje standardnog jezika C koji omogućava definiranje paralelnih funkcija, odnosno kernela, koji će se izvršavati na grafičkom procesoru. I kratki primjer kojeg prikazuje kod 1 je dovoljan da bi se objasnila osnovna proširenja koja se koriste u *CUDA* C jeziku. Da bi funkcija bila kernel obavezno mora biti kvalificirana sa `__global__` i mora biti `void`. Dodatno proširenje koje *CUDA* C jezik donosi jest sintaksa za određivanje konfiguracije izvršavanja (eng. *execution configuration*) gdje se unutar `<<<...>>>` određuje broj blokova dretvi i broj dretvi unutar svakog bloka. Svaka dretva koja izvršava kod kernela ima jedinstven identifikacijski broj koji se nalazi u ugrađenoj varijabli `threadIdx`. Kod 1 prikazuje jednostavan primjer u kojem će se pomoću grafičkog procesora izračunati suma dva vektora. Kernel će istovremeno izvršavati *N* dretvi, a svaka dretva će obaviti točno jedno zbrajanje. Sve dretve pripadaju jednom bloku.

S obzirom na to da se sve dretve jednog bloka izvode na jednoj jezgri to znači da one dijele memorijske resurse te jezgre pa je zbog toga broj dretvi po bloku ograničen. Na *Kepler* grafičkim procesorima blok može sadržavati najviše 1024 dretve. Unatoč

ograničenju na broj dretvi po bloku, kernel može izvršavati i više blokova jednakih dimenzija. U tom slučaju se veličina bloka određuje tako da je umnožak broja dretvi po bloku i broja blokova jednak ukupnom broju dretvi.

### 4.2.2 Hijerarhija dretvi

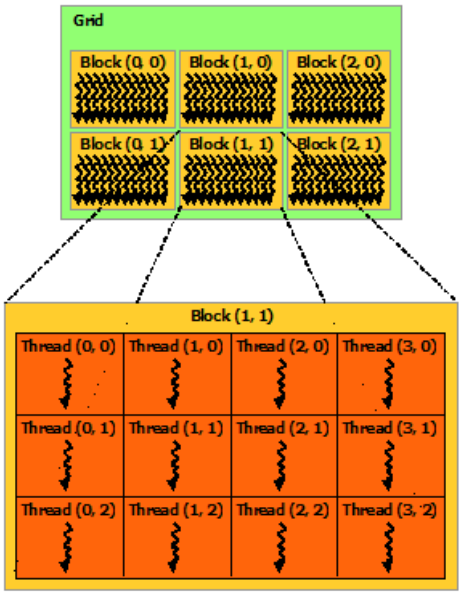
U kodu 1 za određivanje indeksa vektora koristi se komponenta `x` ugrađene varijable `threadIdx`. Varijabla `threadIdx` je trokomponentni vektor pa tako dretve mogu biti identificirane sa jednodimenzionalnim (x komponenta), dvodimenzionalnim (x i y komponente) ili trodimenzionalnim (x, y, i z komponente) indeksom i na taj način tvoriti jednodimenzionalni, dvodimenzionalni ili trodimenzionalni blok dretvi. Višedimenzionalni blokovi dretvi pružaju prirodan pristup raznim izračunima ovisno o domeni pa tako jednodimenzionalni blokovi odgovaraju vektorskim problemima, dvodimenzionalni matričnim, a trodimenzionalni prostornim problemima.

```
// definicija kernela
__global__ void MatAdd(float A[N][N], float B[N][N],float C[N][N]){
    int i = blockIdx.x * blockDim.x + threadIdx.x;
    int j = blockIdx.y * blockDim.y + threadIdx.y;

      C[i][j] = A[i][j] + B[i][j];
}

int main(){
    // ...
    // poziv kernela
    int N = 1024;
    dim3 threadsPerBlock(16, 16);
    dim3 numBlocks(N / threadsPerBlock.x, N / threadsPerBlock.y);
    MatAdd<<<numBlocks, threadsPerBlock>>>(A, B, C);
    // ...
}
```

***Kod 2*: Primjer korištenja dvodimenzionalnih blokova dretvi**

Kod 2 prikazuje zbrajanje dvije matrice dimenzija N × N uz korištenje dvodimenzionalnih blokova. Svaki blok sadrži 256 dretvi i dimenzija je 16 × 16. Uz pretpostavku da je N jednak 1024, postojat će ukupno 4096 blokova koji će biti smješteni u polje (*grid*) dimenzija 64 ×

64, gdje 64 dolazi od $\frac{1024}{16}$. U primjeru se vidi korištenje jos dvije ugrađene varijable kojima svaka dretva ima pristup. To su `blockIdx` i `blockDim`. Obje su trokomponentni vektori kao i `threadIdx`, a označavaju indeks bloka i dimenziju bloka u kojem se dretva nalazi.




***Slika 11*: Grafički prikaz hijerarhije dretvi [16]**

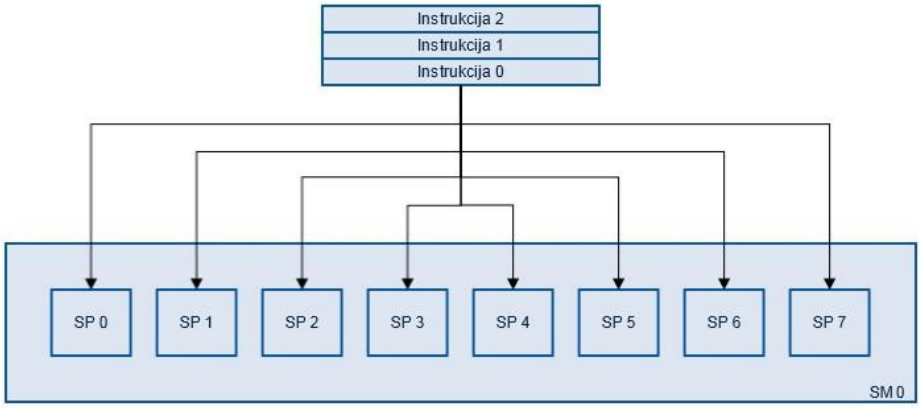
Već iz jednostavnog primjera koji prikazuje kod 2 se vidi da će postojati ukupno $1024 \cdot 1024 = 1{,}048{,}576$ dretvi. Stvoriti i pokrenuti takav broj dretvi na *CPU* je nezamislivo. Jedna od glavnih razlika između *CPU* i *GPU* je način na koji mapiraju registri. *CPU* izvršava više dretvi koristeći preimenovanje registara i stog. U slučaju da treba izvršiti novi zadatak (dretvu) radi zamjenu konteksta koja uključuje pohranu registara dretve koja se trenutno izvršava na stog, i skidanje istih sa stoga pri nastavku njezina izvođenja. Ako se na *CPU* pokrene previše dretvi većina vremena će se trošiti na zamjenu konteksta, a ne na koristan posao pa će efikasnost opadati kako broj dretvi raste.

Kod grafičkog procesora situacija je potpuno drugačija. Za razliku od *CPU*, *GPU* ne koristi tehniku preimenovanja registara, već svakoj dretvi dodijeljuje određen broj fizičkih registara pa je zamjena konteksta gotovo bez ikakvog troška. Kada se zamjena konteksta treba dogoditi pokazivač na trenutni skup registara se preusmjeri na skup registara *warpa*

koji će se izvršiti. Iako je rečeno da svaka dretva dobije svoj skup registara zamjena konteksta se ne odvija na razini dretve nego na razini *warpa*, jer na izvršavanje ne odlazi pojedina dretva već cijeli *warp*. Zbog niskog troška zamjene konteksta grafički procesor očekuje vrlo velik broj dretvi jer na taj način sakriva vrijeme koje troše memorijske operacije – u trenutku kada jedan *warp* čeka na dohvat podatka iz memorije on je praktički zaustavljen pa raspoređivač *warpova* na izvršavanje rasporedi neki drugi *warp*, a kako je zamjena konteksta puno brža od memorijskih operacija, čekanje uzrokovano pristupom memoriji je sakriveno sa korisnim radom drugog *warpa*. Malen broj dretvi na grafičkom procesoru obično znači da će procesor većinu vremena biti u mirovanju jer će često čekati na završetak memorijskih operacija.

### *4.2.3 Warp*

*CUDA* je bazirana na varijanti *SIMD* modela kojeg je Nvidia nazvala *SIMT* (eng. *Single Instruction Multiple Thread*). Multiprocesor stvara, upravlja, raspoređuje i izvršava grupe od po 32 paralelne dretve (eng. *warp*). Dretve koje čine jedan *warp* počinju od iste adrese, ali svaka ima svoje programsko brojilo i registar stanja što im omogućuje nezavisno izvršavanje i grananje. Kada se multiprocesoru na izvršavanje da jedan ili više blokova dretvi, one se dijele u *warpove*. Način na koji je blok dretvi raspoređen u *warpove* je uvijek isti – svaki *warp* sadrži uzastopni niz dretvi, gdje su dretve označene indeksima od 0 na dalje tako prvi *warp* sadržavati dretve s indeksima 0 – 31, drugi 32 – 63 itd. Sve dretve koje pripadaju jednom *warpu* se izvršavaju u *lock – step* načinu rada, odnosno, svaka instrukcija iz reda insktrukcija se proslijeđuje svakom *SP* unutar *SM*. Ako pretpostavimo da imamo N *SP* jedinica, zbog *lock – step* načina rada ista instrukcija se iz memorija dohvaća samo jednom, a ne *N* puta, što smanjuje broj pristupa memoriji. No, to ima i svoju cijenu – ako svaka dretva iz pojedinog *warpa* ne slijedi istu nit izvođenja, npr. zbog uvjetnog grananja, dolazi do divergencije dretvi unutar *warpa* (eng. *warp divergence*).



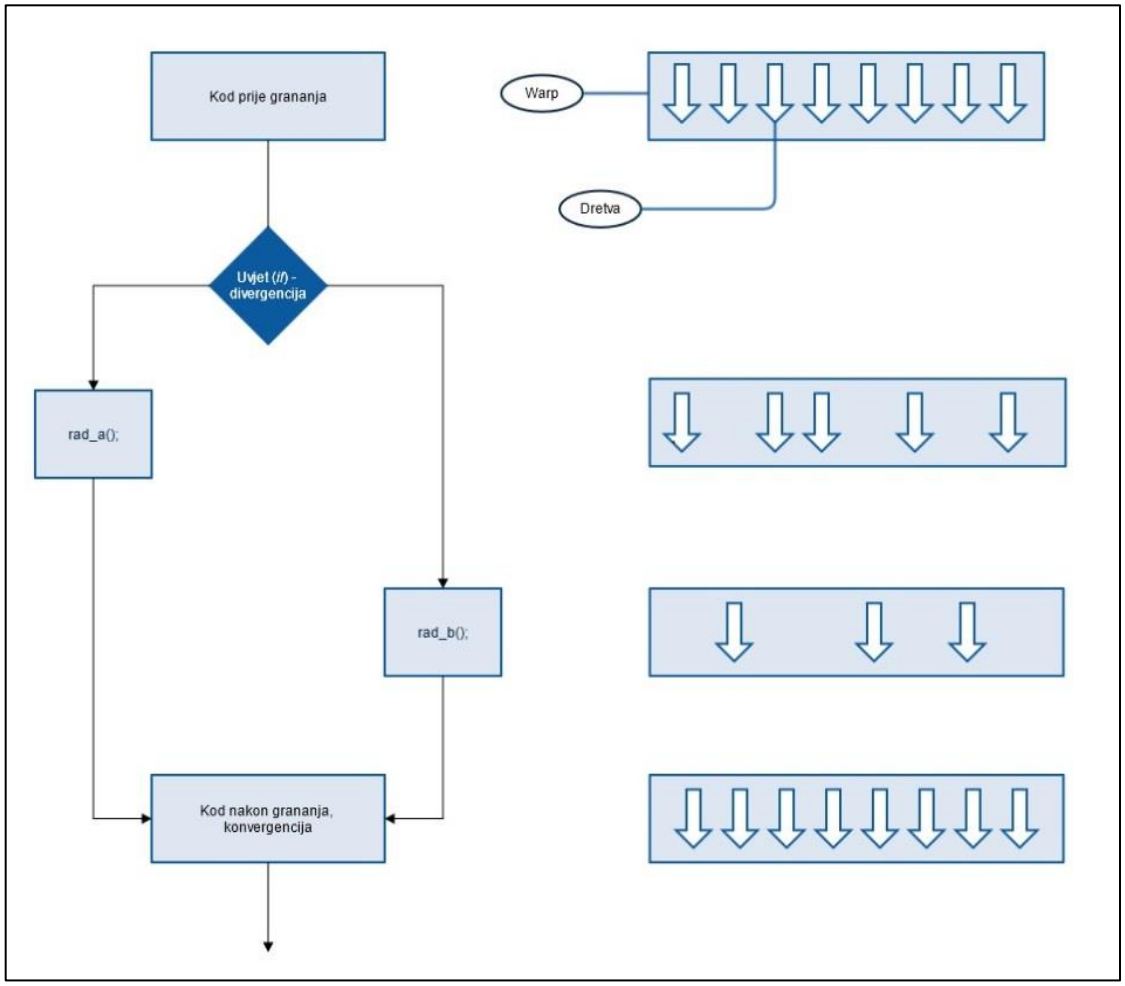

***Slika 12*: *Lock - step* način rada**

Za razliku od središnjeg procesora računala koji ima kompleksno sklopovlje za predviđanje grananja, grafički procesor je u tom pogledu puno jednostavniji (slika 7). Dretve koje zadovolje uvjet grananja i uđu u granu su aktivne, a one koje ne zadovolje uvjet postaju neaktivne. Programer mora biti svjestan da svako uvjetno grananje (npr. *if* naredba) može značajno sniziti performanse unutar *SM* jer se svaka grana svakog uvjetnog grananja mora evaluirati. Dugi nizovi naredbi unutar grana mogu prouzročiti i dvostruko usporenje, *N* ugniježđenih petlji usporenje od $2^N$, a maksimalno usporenje od 32 puta je moguće ako svaka dretva unutar *warpa* izvršava zasebnu granu. Usporenje se javlja jer se grane uvjetnog grananja moraju serijalizirati. No, grafički procesori novijih generacija (*Fermi* i noviji) imaju mogućnost micanja kraćih grananja na način da izvršavaju obje strane *if* grane paralelno i na taj način izbjegavaju krivo predviđena grananja i samu divergenciju. Dodatno, u nekim slučajevima i prevodioc može pomoći – ako je kod unutar grana dug, a da se izbjegne dvostruko usporenje, prevodioc može umetnuti dio koda koji će izvršiti *warp voting* koji provjerava da li sve dretve *warpa* odabiru istu granu. Ako sve dretve *warpa* „glasaju“ za istu granu, grananje se miče i ne dolazi do usporavanja.

```
// kod dretve prije grananja
// ....
if (uvjet) {
      rad_a();
} else {
      rad_b();
}
// kod nakon grananja
// ....
```

***Kod 3*: Primjer grananja i divergencije**

*Kod 3* je jednostavan primjer kojim će se uzrokovati divergencija. Čim se *uvjet* evaluira doći će do divergencije u barem jednom bloku dretvi jer u suprotnom ne bi ni imalo smisla imati neko uvjetno testiranje, već bi sve dretve išle istim putem. Pretpostavimo da će dretve sa parnim indeksima izvršiti prvu (*if*) granu, a one s neparnim drugu (*else*). Kako se cijeli *warp* izvršava u *lock – step* načinu rada, u jednom trenutku se može izvršavati samo jedna instrukcija, odnsno, u ovom primjeru će samo polovica *warpa* biti aktivna. Nakon što se izvrše obje grane, *warp* ponovo konvergira. Slika 13 to prikazuje grafički, a radi jednostavnosti prikazano je samo 8 dretvi unutar *warpa*.

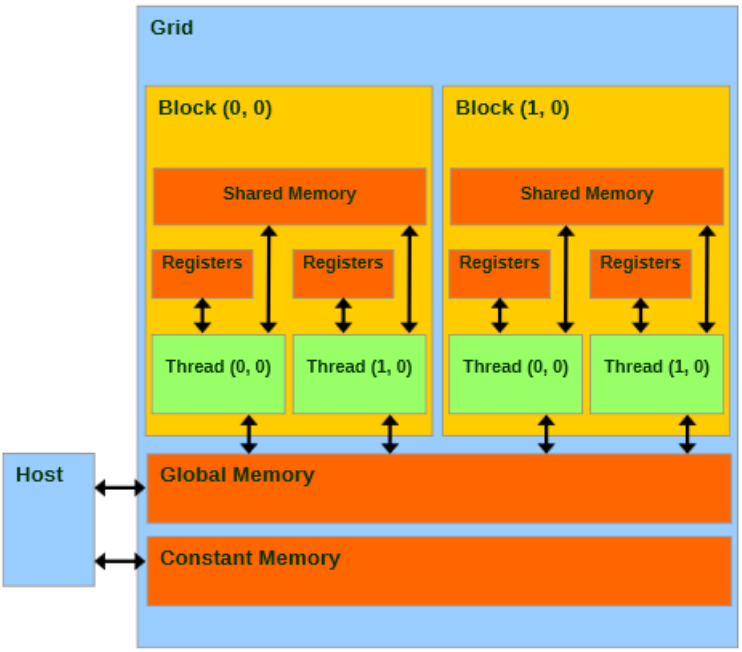



***Slika 13*: Divergencija i konvergencija**

## 4.3 Upravljanje memorijom

Da bi GPGPU aplikacija postigla najviše performanse zahtijeva se mogućnost višekratnog iskorištavanja istih podataka. Razlog je je to što globalna memorija jednostavno nije dovoljno brza da bi zadovoljila potreble svih *streaming* multiprocesora. Prijenos podataka između *hosta* i *device-a* dodatno pogoršava problem jer sve *DMA* (eng. *Direct Memory Access*) operacije idu kroz globalnu memoriju, što unosi dodatno kašnjenje. *CUDA* programeru omogućava korištenje nekoliko tipova memorija i programirljive priručne memorije s ciljem ostvarenja višekratnog iskorištavanja istih podataka.

S obzirom na to da se većina *CUDA* programa razvija progresivno, čest je slučaj da se u ranim fazama isključivo koristi globalna memorija. Tek nakon inicijalne implementacije kreće se s korištenjem ostalih tipova memorija, ali unatoč tome već od samog početka je potrebno razmišljati o njima i kako bi one utjecale na performanse programa ovisno o tome gdje i ako se mogu iskoristiti. Jedna od najvažnijih stvari koje bi svaki *CUDA* programer trebao savladati jest pravilno korištenje određenih tipova memorija, a u svrhu povećanja performansi napisanog programa.




***Slika 14*: Tipovi memorija i njihovi odnosi [16]**

U nastavku će biti opisani bitni tipovi memorija te kako oni utječu na performanse programa.

***Tablica 1*: Tipovi prosotra za pohranu podataka sa vremenima pristupa**

| **Tip** | **Registar** | **Dijeljena memorija** | **Globalna memorija** |
|---|---|---|---|
| **Vrijeme pristupa** | 1 ciklus | 1 do 32 ciklusa | 400 – 600 ciklusa |

### 4.3.1 Registri i lokalna memorija

Registri su najbrži oblik nekog prostora za pohranu podataka na grafičkom procesoru. Svaki *SM* raspoređuje određen broj blokova dretvi koje vidi kao logičke skupine nezavisnih *warpova*. Broj registara koji će biti dodijeljen svakoj dretvi je određen tijekom prevođenja programa. Svi blokovi su iste veličine i imaju poznat broj dretvi pa *GPU* zna koliko registara svaka dretva treba. Na grafičkim procesorima iz *Kepler* serije maksimalan broj registara koji jedna dretva može dobiti jest 255. No, taj broj ovisi o broju blokova, dimenzijama bloka i ukupnom broju registara po jednom *SM.* Kod *Kepler* serije svakom *SM* – u je dostupno 65536 32 - bitnih registara. Ako *kernel* zahtijeva previše registara po jednoj dretvi to može ograničiti broj blokova koje *GPU* može rasporediti na *SM,* a samim time i broj dretvi koje će istovremeno biti pokrenute. Premalen broj dretvi vodi do nedovoljnog iskorištenja procesora i performanse počinju opadati.

Registri se koriste za pohranu lokalnih varijabli dretvi, no postoje i neke iznimke u kojima je varijabla pohranjena u lokalnu memoriju dretve, a ne u registar. To je slučaj sa poljima koja nisu indeksirana sa konstantama, velikim strukturama ili poljima koja bi zauzimala previše registara i varijablama koje se preliju u lokalnu memoriju jer *kernel* zahtijeva više registara nego što mu je na raspolaganju (kada sve definirane varijable ne bi stale u registre). Svaka varijabla koja se deklarira unutar kernela se automatski sprema u jedan od alociranih registara osim u navedenim slučajevima.

Memorijske operacije sa varijablama koje se nalaze u lokalnoj memoriji su vrlo spore jer se lokalna memorija zapravo dio vrlo spore globalne memorije. Iako se lokalne varijable nalaze u globalnoj memoriji, kao što i samo ime kaže, vidljive su samo dretve

koja ih deklarira. Lokalna memorija je organizirana na način da uzastopnim 32 – bitnim riječima pristupaju dretve za uzastopnim identifikacijskim brojevima. Pristupi lokalnim varijablama su potpuno sjedinjeni (eng. *coalesced*) sve dok dretve unutar *warpa* pristupaju istim relativnim adresama, npr. istom elementu polja ili istoj članskoj varijabli neke strukture. To znači da neće svaka dretva *warpa* posebno čitati iz lokalne, odnosno globalne memorije, već će se obaviti samo jedno, sjedinjeno čitanje. Na novijim grafičkim procesorima pristupi lokalnoj memoriji se uvijek keširaju u L1 i L2.

### 4.3.2 Dijeljena memorija

Dijeljena memorija (eng. *shared memory*) je zapravo programirljiva L1 priručna memorija. L1 i dijeljena memorija zajedno dijele 64KB memorije po svakom *SM*, od čeka 16KB pripada L1, a 48KB dijeljenoj memoriji, no, programer može i ručno postaviti te vrijednosti prije poziva *kernela*. S obzirom da se nalazi na samom *SM*, dijeljena memorija ima mnogo veću propusnost i mnogo manju latenciju od lokalne ili globalne memorije. Da bi se postigla takva propusnost, dijeljena memorija je podijeljena u module jednakih veličina koji se nazivaju banke, a njima se može pristupati istovremeno. Ako memorijska operacija zahtijeva pristup *N* memorijskih lokacija koje padaju u *N* različitih banki, svim bankama će se pristupiti istovremeno, a to znači i propusnost veću *N* puta od slučaja da se koristi samo jedna banka. Ako pak dvije ili više memorijskih lokacija pada u istu banku dolazi do konflikta (eng. *bank conflict*) pa se pristupi moraju serijalizirati. Poseban slučaj jest kada sve dretve jednog *warpa* čitaju iz iste adrese u banki. U tom slučaju se aktivira mehanizam koji svim dretvama odašilje traženu vrijednost. Konflikte je uvijek potrebno izbjegavati jer slijedni pristup banki neće biti prekriven pokretanjem novog *warpa* (kao što je slučaj sa pristupom u globalnu memoriju pri čemu se aktivira neki drugi *warp*), već će *SM* uistinu biti zaustavljen sve dok se ne završe sve memorijske operacije.

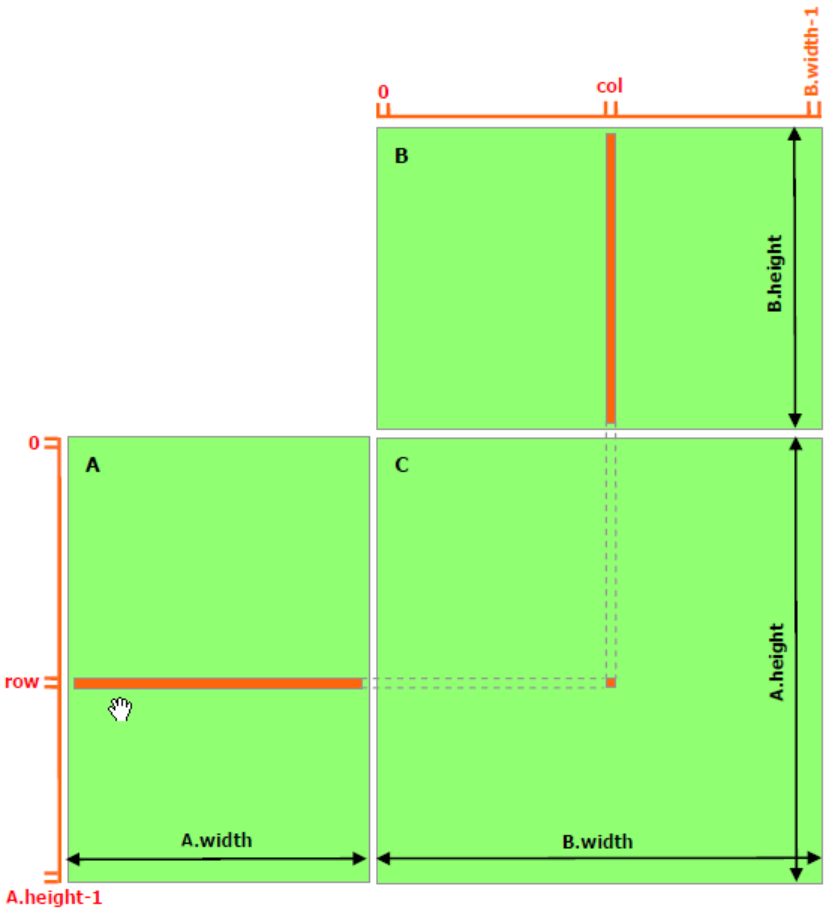
Varijablu koja će biti u dijeljenoj memoriji potrebno je kvalificirati sa `__shared__`. Varijable mogu biti bilo kojeg tipa i dimenzija, pri čemu je potrebno paziti na maksimalnu količinu dijeljene memorije po jednom bloku dretvi. Prostor za varijablu u

dijeljenu memoriju se može alocirati na dva načina, statički i dinamički, odnsno tijekom prevođenja programa ili tijekom pokretanja programa.

Razlika u performansama programa koji na ispravan način koristi dijeljenu memoriju od onoga koji ju ne koristi biti će prikazana u sljedećim primjeru množenja matrica. Klasični slijedni algoritam množenja matrica će u slučaju množenja matrica **A** i **B** dimenzija N × M i M × P imati složenost $O(MNP) \approx O(N^3)$, što će biti vrlo sporo čak i na najbržim procesorima ako se radi o matricama velikih dimenzija.

#### 4.3.2.1 Množenje matrica bez korištenja dijeljene memorije

U ovom primjeru svaka dretva čita jedan redak matrice **A** i jedan stupac matrice **B** i računa korespondentni element matrice **C**.




***Slika 15*: Paralelno množenje matrica bez korištenja dijeljene memorije [16]**

```
// struktura koja predstavlja matricu
// elementi su spremljeni po retcima (row-major)
typedef struct {
    int width;
    int height;
    float* elements;
} Matrix;

// Kernel zadužen za množenje matrica
__global__ void MatMulKernel(Matrix A, Matrix B, Matrix C){
    // Svaka dretva računa jedan element matrice C
    // Cvalue – lokalna varijabla (registar!)
    float Cvalue = 0;

    // redak matrice A
    int row = blockIdx.y * blockDim.y + threadIdx.y;

    // stupac matrice B
    int col = blockIdx.x * blockDim.x + threadIdx.x;

    // izračun elementa matrice C
    for (int e = 0; e < A.width; ++e)
        Cvalue += A.elements[row * A.width + e]
                * B.elements[e * B.width + col];

    // tek nakon izračuna rezultat se piše u globalnu m.
    C.elements[row * C.width + col] = Cvalue;
}

// ......

// Poziv kernel
int BLOCK_SIZE = 16;
dim3 dimBlock(BLOCK_SIZE, BLOCK_SIZE);
dim3 dimGrid(B.width / dimBlock.x, A.height / dimBlock.y);
MatMulKernel<<<dimGrid, dimBlock>>>(d_A, d_B, d_C);
```

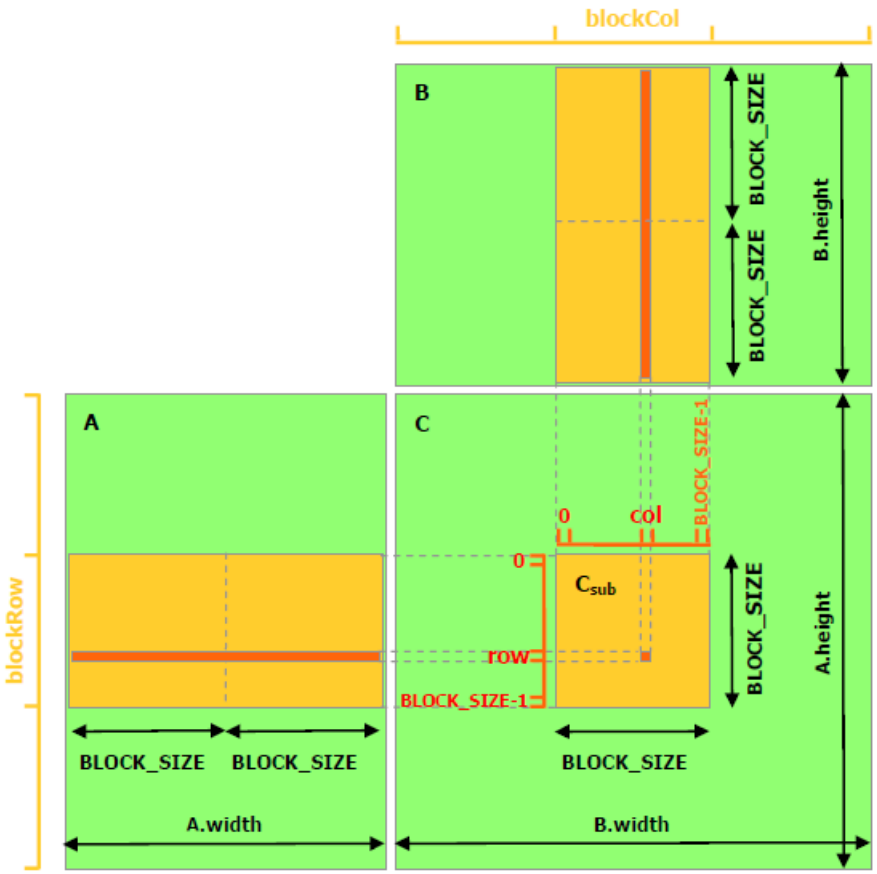
***Kod 4*: Paralelno množenje matrica bez korištenja dijeljene memorije**

#### *4.3.2.2 Množenje matrica sa korištenjem dijeljene memorije*

Ovaj primjer je ponešto kompliciraniji od prethodnog. U ovom primjeru svaki blok dretvi je odgovoran za izračun jedne kvadratne podmatrice $\mathbf{C}_{\text{sub}}$rezultantne matrice $\mathbf{C}$. Svaka dretva unutar bloka je odgovorna izračun jednog elementa podmatrice $\mathbf{C}_{\text{sub}}$. Kao što *Slika 16* prikazuje, $\mathbf{C}_{\text{sub}}$je jednaka produktu dvije matrice: podmatrice $\mathbf{A}$ dimenzija $\mathbf{A}_{\text{M}} \times \text{BK}$, i podmatrice $\mathbf{B}$ dimenzija $\text{BK} \times \mathbf{A}_{\text{M}}$, gdje je $\mathbf{A}_{\text{M}}$ broj stupaca matrice $\mathbf{A}$, a BK dimenzija bloka dretvi. Da bi se što bolje iskoristili dostupni resursi, te dvije matrice su podijeljene u pravokutne podmatrice dimenzija $\text{BK} \times \text{BK}$, a $\mathbf{C}_{\text{sub}}$se tada računa kao suma

produkata tih pravokutnih podmatrica. Svaki od produkata se računa na način da se prvo učitaju dvije korespondetne kvadratne matrice iz globalne u dijeljenu memoriju, gdje jedna dretva učitava jedan element kvadratne matrice i računa jedan element produkta. Svaka dretva akumulira rezultat produkata u registar i na kraju zapiše rezultat u globalnu memoriju.

Takvom raspodjelom iskorištava se brza dijeljena memorija i smanjuje se broj sporih pristupa u globlanu memoriju jer je matrica $\mathbf{A}$ pročitana samo $\frac{\mathbf{B}_\mathrm{P}}{BK}$, a matrica $\mathbf{B}$ $\frac{\mathbf{A}_N}{BK}$ puta iz globalne memorije, pri čemu je $\mathbf{B}_\mathrm{P}$ broj stupaca matrice $\mathbf{B}$, a $\mathbf{A}_\mathrm{N}$ broj redaka matrice $\mathbf{A}$.




***Slika 16*: Paralelno množenje matrica sa korištenjem dijeljene memorije [16]**

```
// Dohvaća element matrice
__device__ float GetElement(const Matrix A, int row, int col){
    return A.elements[row * A.width + col];
}

// Set a matrix element
__device__ void SetElement(Matrix A, int row, int col,
                           float value){
    A.elements[row * A. width + col] = value;
}

// Dohvaća kvadratnu matricu dimenzija BK×BK, odnsno BLOCK_SIZE
// * BLOCK_SIZE zadane matrice A
 __device__ Matrix GetSubMatrix(Matrix A, int row, int col){
    Matrix Asub;
    Asub.width    = BLOCK_SIZE;
    Asub.height   = BLOCK_SIZE;
    Asub.stride   = A. width;
    Asub.elements = &A.elements[A.width * BLOCK_SIZE * row
                                        + BLOCK_SIZE * col];
    return Asub;
}


// Kernel
 __global__ void MatMulKernelShared(Matrix A, Matrix B, Matrix C)
{
    // Redak i stupac bloka
    int blockRow = blockIdx.y;
    int blockCol = blockIdx.x;

    // Svaki blok dretvi jednu podmatricu Csub matrice C
    Matrix Csub = GetSubMatrix(C, blockRow, blockCol);

    // Svaka dretva računa jedan element Csub
    // akumulirajući rezultat u Cvalue
    float Cvalue = 0;

    // Redak i stupac unutar Csub
    int row = threadIdx.y;
    int col = threadIdx.x;

    // Petlja prolazi kroz sve podmatrice matrica A i B
    // koje su potrebne za izračun Csub
    // Pomnoži svaki par podmatrica i akumuliraj rezultat
    for (int m = 0; m < (A.width / BLOCK_SIZE); ++m) {

        // Podmatrica Asub matrice A
        Matrix Asub = GetSubMatrix(A, blockRow, m);

        // Podmatrica Bsub matrice B
        Matrix Bsub = GetSubMatrix(B, m, blockCol);

        // Alokacija dijeljene memorije za Asub i Bsub
        __shared__ float As[BLOCK_SIZE][BLOCK_SIZE];
        __shared__ float Bs[BLOCK_SIZE][BLOCK_SIZE];

        // Učitavanje iz globalne u dijeljenu memoriju
        As[row][col] = GetElement(Asub, row, col);
        Bs[row][col] = GetElement(Bsub, row, col);
```

```
        // Sinkronizacija je potrebna da se osigura da su sve
        // podmatrice učitane u dijeljenu memoriju prije
        // računanja
        // Ovo je sinkronizacijski mehanizam koji se naziva
        // barijera
        __syncthreads();

        // Pomnoži Asub i Bsum
        for (int e = 0; e < BLOCK_SIZE; ++e)
            Cvalue += As[row][e] * Bs[e][col];

        // Sinkronizacija koja osigurava da su izračuni gotovi
        // prije učitavanja novih podmatrica Asub i Bsub
        // u slijedećoj iteraciji
        __syncthreads();
    }

    // Zapis Csub u globalnu memoriju. Svaka dretva piše jednu
    // vrijednost
    SetElement(Csub, row, col, Cvalue);
}
```

***Kod 5*: Paralelno množenje matrica uz korištenje dijeljene memorije**

Na korištenom računalu rezultati ovisno o korištenju ili nekorištenju dijeljene memorije su prikazani u *Tablica 2*.

***Tablica 2*: Vremena izvođenja primjera množenja matrica**

| **Varijanta** | **Bez dijeljene memorije** | **Sa dijeljenom memorijom** |
|---|---|---|
| **Vrijeme** | 145,857 ms | 43,488 ms |

### 4.3.3 Globalna memorija

Osim dobrog poznavanja dijeljene memorije posebno je važno poznavati i globalnu memoriju. Za globalnu memoriju grafičkog procesora se kaže da je globalna iz razloga što se u nju može pisati i sa *device* i *host* strana, a zapravo joj se može pristupiti s bilo kojeg uređaja spojenog na *PCI – E* sabirnicu. To znači da ako sustav ima više grafičkih procesora oni mogu prenositi podatke između sebe bez uplitanja *CPU* – a. Globalnoj memoriji sa *host* strane može se pristupiti na jedan od tri načina:

- Eksplicitno sa blokirajućim (sinkronim) prijenosom podataka – kontrola se inicijatoru prijenosa (*CPU*) vraća tek nakon što je prijenos u potpunosti završio. Funkcija kojom se vrši blokirajući prijenos podataka jest *cudaMemcpy().*
- Eksplicitno sa neblokirajućim (asinkronim) prijenosom podataka – kontrola se inicijatoru prijenosa vraća odmah nakon iniciranja prijenosa. Funkcija kojom se vrši neblokirajući prijenos podataka jest *cudaAsymcMemcpy().*
- Implicitno sa alociranjem mapirane memorije (eng. *pinned/mapped memory, zero – copy*) – dio *host* memorije se mapira u globalnu memoriju grafičkog procesora, ali bez eksplicitnog kopiranja, koristi se kod brzih asinkronih prijenosa. Funkcija kojom se vrši alociranje mapirane memorije jest *cudaHostAlloc().*

Globalna memorija se može alocirati na dva načina:

- Sa *host* strane programa pozivom funkcije *cudaMalloc()*. Funkcija je ekvivalent funkciji *malloc()* u jeziku C. *cudaMalloc()* vraća pokazivač na alocirani segment memorije ako je alokacija uspjela. Važno je znati da pokazivač čuva adresu iz memorijskog prostora globalne memorije grafičkog procesora. Sa *host* strane nije moguće direktno pristupati memorijskoj lokaciji na koju pokazuje taj pokazivač.
- Sa *device* strane deklariranjem varijable ili polja koja je kvalificirana sa `__device__`. Sa istim kvalifikatorom se kvalificiraju i sve funkcije i strukture koje leže i izvršavaju se na grafičkom procesoru.

#### *4.3.3.1 Primjer zbrajanja dva vektora – kompletan kod*

U svim do sad prikazanim odjsečcima kodova bili su prikazani samo dijelovi koda koji su se ticali *kernela* i *device* strane koda. U sljedećem primjeru će biti prikazana i *host* strana koja uključuje alociranje i prijenos memorije.

```
#include <stdio.h>


// Definicija kernela
__global__ void
vectorAdd(const float *A, const float *B, float *C, int
numElements){
    // odredivanje indeksa vektora
    int i = blockDim.x * blockIdx.x + threadIdx.x;

    if (i < numElements) {
        C[i] = A[i] + B[i];
    }
}


int main(){
    // 50000 elemenata
    int numElements = 50000;
    size_t size = numElements * sizeof(float);

    // alociranje memorije za host vektor A
    float *h_A = (float *)malloc(size);

    // alociranje memorije za host vektor B
    float *h_B = (float *)malloc(size);

    // alociranje memorije za host vektor C
    float *h_C = (float *)malloc(size);


    // incijaliziranje host vektora
    for (int i = 0; i < numElements; ++i) {
        h_A[i] = rand()/(float)RAND_MAX;
        h_B[i] = rand()/(float)RAND_MAX;
    }

    // alociranje memorije za vektor A na device strani
    float *d_A = NULL;
    cudaMalloc((void **)&d_A, size);

    // alociranje memorije za vektor B na device strani
    float *d_B = NULL;
    cudaMalloc((void **)&d_B, size);

    // alociranje memorije za rezultntni vektor na device strani
    float *d_C = NULL;
    cudaMalloc((void **)&d_C, size);
```

```
    // kopiranje vektora A na device
    cudaMemcpy(d_A, h_A, size, cudaMemcpyHostToDevice);
      // kopiranje vektora B na device
    cudaMemcpy(d_B, h_B, size, cudaMemcpyHostToDevice);

    // odredivanje konfiguracije izvodenja i pozivanje kernela
    int threadsPerBlock = 256;
    int blocksPerGrid =(numElements + threadsPerBlock - 1) /
threadsPerBlock;
    vectorAdd<<<blocksPerGrid, threadsPerBlock>>>(d_A, d_B, d_C,
numElements);

    // kopiranje rezultantnog vektora sa device na host
    cudaMemcpy(h_C, d_C, size, cudaMemcpyDeviceToHost);

      // u ovom trenutku hostu je dostupan izracaunati vektor C

    // oslobadanje defice memorije
    cudaFree(d_A);
    cudaFree(d_B);
    cudaFree(d_C);

    // oslobadanje host memorije
    free(h_A);
    free(h_B);
    free(h_C);

    return 0;
}
```

***Kod 6*: Kompletan kod - prikazane su i *host* i *device* strane**

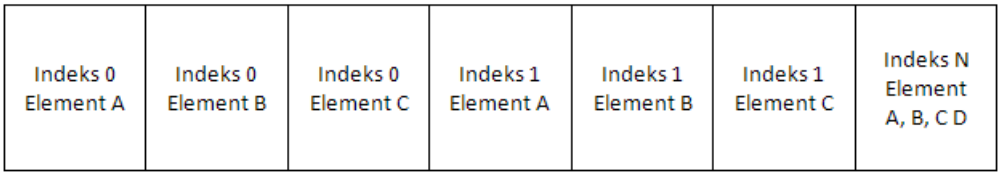
Kod korištenja globalne memorije vrlo je bitno koristiti sjedinjene (eng. *coalesced*) pristupe memoriji. Ako je odnos između uzastopnih indeksa dretvi i uzastopnih memorijskih lokacija jedan na jedan, pojedinačni pristupi memoriji se kombiniraju u jedan te se izvršava samo jedna transakcija iz memorije. Primjerice, ako sve dretve *warpa* pristupaju segmentu polja cijelobrojnih elemenata (4 bajta) umjesto 32 transakcije po 4 bajta dogoditi će se jedna transakcija od 128 bajtova. S obzirom na to da je pristup globalnoj memoriji spor (visoka latencija – visoko vrijeme od trenutka iniciranja pristupa nekoj memorijskoj lokaciji do trenutka primanja podatka s te adrese ) više pojedinačnih pristupa će biti puno sporije od jednog pristupa u kojem se zahtijeva veća količina podataka. Sjedinjene pristupe memoriji nije uvijek moguće koristiti, ali preporuča ih se koristiti kada god je to moguće.

Vrlo često se u raznim problemima podaci javljaju u nekim strukturama, npr. struktura koja predstavlja vrh poligona u nekom tipičnom problemu računalne grafike, gdje je svaki vrh predstavljen sa tri *float* varijable.

```
typedef struct {
      float A;
      float B;
      float C;
} Vertex;

Vertex vertices[1024];
```

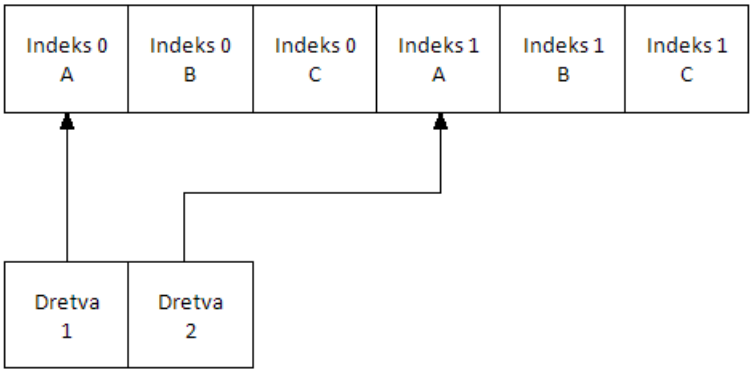
***Kod 7*: Struktura koja predstavlja vrh**

Organizacija podataka u ovakve strukture je uobičajen način u serijskom programiranju jer takve strukture odgovaraju prirodnok reprezentaciji objekta kojeg modeliraju. U slučaju kada bismo imali polje tako definiranih vrhova, njihov smještaj u memoriji bi bio kao što prikazuje *Slika 17*. Na ovaj način je osigurana lokalnost podataka.




***Slika 17*: Polje struktura (eng. *Array of structures*)**

Iako normalan način u serijskom programiranju, ako bi se podaci u *CUDA* programu smjestili u memoriju na ovakav način, a uz pretpostavku da jedna dretva vrši operacije nad jednom takvom strukturom, pristupi memoriji ne bi bili sjedinjeni i došlo bi do pada performansi.




***Slika 18*: Polje struktura - nesjedinjeni pristup memoriji**

Ovakav problem nesjedinjenih pristupa može se vrlo lako riješiti upotrebom polja struktura (eng. *Structure of arrays*) umjesto polja struktura.

```
typedef struct {
      float A[1024];
      float B[1024];
      float C[1024];
} Vertices;
```

***Kod 8*: Struktura polja - osiguravanje sjedinjenog pristupa globalnoj memoriji**

***Tablica 3*: Pregled tipova prostora za pohranu podataka**

| Deklaracija varijable | Memorija | Doseg | Životni vijek |
|---|---|---|---|
| `float A` | Registar | Dretva | Dretva |
| `float A[10];` | Lokalna | Dretva | Dretva |
| `__shared__ float A[10]` | Dijeljena | Blok dretvi | Blok dretvi |
| `__device__ float A[10]` | Globalna | Grid | Aplikacija |

# 5 Biometrijski sustavi za identifikaciju osoba

Identifikacija osoba može se provoditi na dva načina: pomoću identifikacijskih dokumenata i sigurnosnog ključa. Prilikom korištenja identifikacijskog dokumenta sam dokument predstavlja identite osobe koja ga nosi. To mogu biti osobna iskaznica, putovnica i sl. Samim time što dokument označava identitet osobe moguća je situacija u kojoj se neka osoba lažno predstavlja koristeći identifikacijski dokument druge osobe. Upravo mogućnost lažnog predstavljanja je nedostatak identifikacije pomoću identifikacijskog dokumenta.

Kod identifikacije pomoću sigurnosnog ključa osoba koristi niz znakova koji su samo njoj poznati. Ta metoda identifikacije se najčešće koristi kod bankovnih transakcija i nekih računalnih sustava. Unatoč tome što se sigurnosni ključevi najčešće sažimaju (eng. *hash*) i kriptiraju, ako je ključ prekratak ili prejednostavan postoji šansa da će biti probijen.

Navedene nedostatke moguće je umanjiti korištenjem biometrijskih sustava za identifikaciju osoba. Biometrija je znanost o automatiziranim postupcima za jedinstveno prepoznavanje ljudi na temelju jednog ili više urođenih tjelesnih obilježja, ili obilježja čovjekovog ponašanja [2]. Kod biometrijskog sustava osoba sama preuzima ulogu ključa. Biometrijski sustav je u principu sustav za raspoznavanje uzoraka koji prikuplja biometrijske podatke osobe i iz njih izlučuje biometrijske značajke koje tada uspoređuje sa predlošcima u bazi podataka.

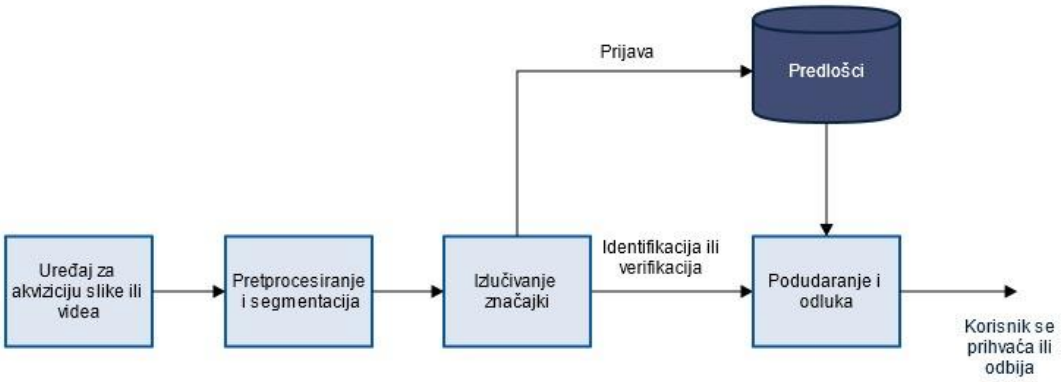
Glavne prednosti biometrijskih metoda identifikacije su [2]:

- Biometrijske značajke osobu definiraju u svakom trenutku
- Biometrijske značajke je teško kopirati i krivotvoriti
- Zahtijeva se da osoba koju je potrebno identificirati bude pristuna na mjestu identifikacije

Biometrijske karakteristike mogu biti [2]:

- fiziološke – lice, otisak prsta, otisak dlana, šarenica i mrežnica oka
- ponašajne – potpis, glas, dinamika tipkanja, hod

Biometrijski sustavi obično rade u dvije faze. Prva je faza prijave u kojoj se korisnik prvi puta prijavljuje na sustav sa svojim biometrijskim značajkama. Izlučene biometrijske značajke se tada spremaju u bazu podataka biometrijskog sustava. Druga faza je faza autorizacije u kojoj korisnik od sustava zahtijeva identifikaciju ili verifikaciju pri čemu se izlučene značajke uspoređuju s onima koje su pohranjene u bazi podataka, a koje su nastale u prvoj fazi. Na temelju usporedbe se donosi odluka o rezultatu identifikacije ili verifikacije.




***Slika 19*: Uobičajena struktura biometrijskih sustava temeljenih na licu ili dlanu prema [17]**

Prilikom rada biomentrijskog sustava mogu nastati dvije vrste pogrešaka. Pogreška lažnog odbijanja (*eng. False Rejection, FR)* javlja se kada biometrijski sustav odbija osobu koju je trebao propustiti. Pogreška lažnog prihvaćanja (eng. *False Acceptance, FA*) nastupa kada sustav propusti osobu koju je trebo odbiti. Ispravno odbijanje (eng. *Genuine Rejection*) i ispravno prihvaćanje (eng. *Genuine Acceptance*) nastupaju kada sustav ispravno odbije ili prihvati osobu. Navedeni slučajevi omogućuju izražavanje osnovnih mjera točnosti biometrijskog sustava. To su udio lažnog odbijanja (eng. *False Rejection Rate, FRR*) i udio lažnog prihvaćanja (eng. *False Acceptance Rate, FRR*). Obje mjere se izražavaju u postocima, a cilj kvalitetnog sustava je postizanje što nižih vrijednosti tih mjera.

$$FRR = \frac{FR}{GA + FR} \tag{1}$$

$$FAR = \frac{FA}{GA + FR} \tag{2}$$

Prilikom izgradnje biometrijskog sustava osim točnosti u obzir treba uzeti još neke karakteristike na kojima se sustav zasniva, npr. [17]:

- Univerzalnost – posjeduje li svaka osoba tu karakteristiku
- Jedinstvenost – je li ta karakteristika kod bilo koje dvije osobe različita
- Stalnost – mijenja li se ta karakteristika kroz vrijeme
- Mogućnost prikupljanja – može li se ta karakteristika na jednostavan način prikupiti i prenesti na računalo
- Učinkovitost – kolika se točnost postiže korištenjem karakteristike i koliko je brza obrada te karakteristike
- Prihvatljivost – koliko su ljudi spremni koristiti ovu značajku
- Mogućnost prevare – koliko teško je prevariti sustav koji koristi ovu karakteristiku

Danas se u biometrijskim sustavima koriste različite biometrijske karakteristike. Svaka od njih ima svoje prednosti i mane i izbor konkretnih karakteristika koje će se koristiti u sustavi ovisi o njegovoj primjeni. Ne postoji biometrijska karakteristika koja će ispuniti sve zahtjeve svih primjena, odnosno, nijedna karakteristika nije „optimalna“. S obzirom da se ovaj rad temelji na licu i dlanu, u nastavku slijedi njihov kratak opis.

Prikupljanje slike lica je nenametljiva metoda, a slika lica je jedna od najuobičajenih biometrijskih karakteristika koju ljudi koriste za međusobno raspoznavanje. Najpopularniji pristupi prepoznavanju lica temelje se ili na značajkama lica (npr. pozicija i oblik očiju, nosa, usta, obrva, brade i njihovi međusobni prostorni odnosi) ili na analizi lica kao cjeline (holistički pristup) gdje se slika lica prikazuje kao težinska suma neke vrste baznih lica (npr. PCA metoda koja je korištena u ovom radu). Sustavi za

prepoznavanje lica imaju poteškoća sa prepoznavanjem lica sa slika koje su prikupljene pod različitim svjetlosnim uvjetima i iz različitih kuteva (u odnosu na slike na temelju kojih je izgrađena baza korisnika). Situaciju dodatno otežavaju i moguće promjene na osobama, npr. starenje, promjena frizure, nošenje naočala i sl. Optimalan sustav za prepoznavanje lica bi trebao moći detektirati prisutnost lica na prikupljenoj slici, lokalizirati lice i prepoznati lice neovisno o kutu iz kojeg je slika snimljena [19].

Dlanovi ruku su puni izbočina i udubljenja koja tvore jedinstveni uzorak kod svake osobe. Područje dlana je veće od otiska prsta pa je za očekivati da će otisci dlanovi sadržavati više svojstvenih karakteristika. Da bi se pribavila slika dlana potreban je i veći, skuplji senzor od onoga koji bi se koristio za prikupljanje otisaka prstiju. Osim karakterističnih izbočina i udubljenja dlanovi sadrže i neke karakteristične linije koje mogu biti uhvaćene i korištenjem jeftinijih senzora manje razlučivosti. Korištenjem senzora visoke razlučivosti sve bitne karakteristike dlana mogu biti snimljene, a zatim i kombinirane prilikom izgradnje biometrijskog sustava visoke točnosti.

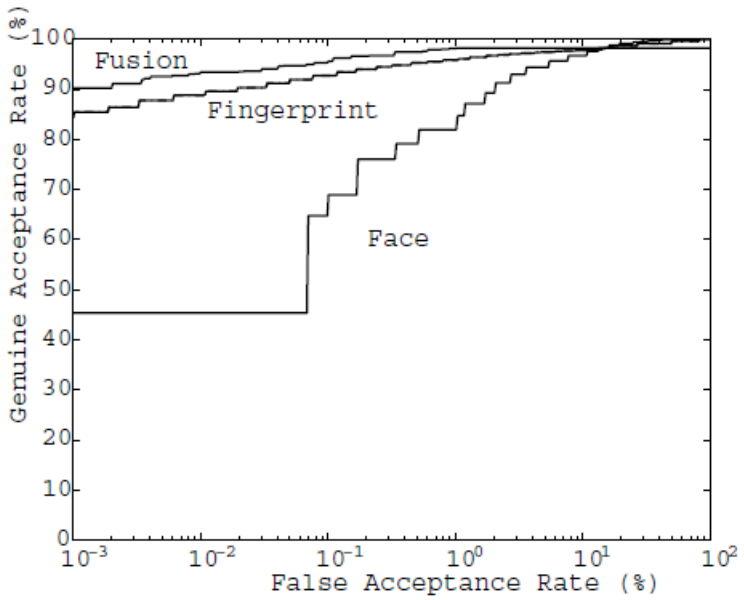
Biometrijski sustavi koji u svom radu koriste samo jednu biometrijsku karakteristiku nazivaju se jednonačinski (eng. *unimodal*) sustavi. Takvi sustavi imaju neke nedostatke [19]:

- Šum u prikupljenim podacima – podaci dobiveni sa senzora mogu biti zašumljeni ili iskrivljeni. Otisak prsta s ožiljkom ili promjena visine glasa zbog utjecaja hladnoće su primjeri podataka sa šumom. Šum u podacima može nastati i zbog neodržavanih senzora (npr. zbog nakupljene prašine na skeneru za uzimanje otisaka prsta) ili zbog loših uvjeta koji vladaju u prostoru u kojem se senzor nalazi (npr. loše osvjetljenje u prostoriji s kamerom gdje se prikuplja slika lica).
- Varijacije unutar razreda – biometrijski podaci prikupljeni od osobe prilikom prijave na sustav mogu biti bitno različiti od onih podataka koji leže u bazi i predstvljaju predloške za tu osobu. Ovakav tip varijacije obično nastaje zbog nepravilne interakcije korisnika i senzora.
- Sličnosti između razreda – iako se očekuje da između pojedinih razreda neka biometrijska karakteristika značajno varira, moguća je situacija da kod

velikih skupova podataka dođe do nekih preklapanja, odnosno sličnostima među karakteristikama između razreda

- Neuniverzalnost – usprkos očekivanju da će svaki korisnik posjedovati traženu biometrijsku karakteristiku, u stvarnosti to ne mora biti tako te je moguće je da će postojati podskup korisnika koji ju ne posjeduju. Primjerice, biometrijski sustav koji radi s otiscima prstiju možda neće moći izlučiti značajke otiska prsta ukoliko korisnik ima slabije izražene izbočine
- Lažno predstavljanje – ova vrsta napada se posebno odnosi na sustave koji se temelje na ponašajnim karakteristikama poput potpisa ili glasa

Neki nedostaci i ograničenja unimodalnih biometrijskih sustava mogu se savladati korištenjem više biometrijskih karakteristika odjednom, npr. kombinacijom slike lica i otiska prsta ili pak kombinacijom otisaka više prstiju. Takvi sustavi nazivaju se višenačinski (eng. *multimodal*) i smatra se da su oni pouzdaniji zbog prisutnosti više nezavisnih dokaza o identitetu osobe. Multimodalni biometrijski sustavi rješavaju problem neuniverzalnosti i lažnog predstavljanja jer je malo vjerojatno da će uljez uspješno lažirati više biometrijskih karakteristika istovremeno.




***Slika 20*: Usporedba rezultata unimodalnog i multimodalnog biomrtrijskog sustava**

## 5.1 Multimodalni biometrijski sustavi

U općenitom slučaju biometrijski sustav se sastoji od četiri osnovna modula [18]:

a) Modul koji sadrži senzor koji prikuplja biometrijsku karakteristiku u obliku sirovih biometrijskih podataka
b) Modul za izlučivanje značajki koji iz sirovih biometrijskih podataka izvlači skup bitnih značajki
c) Modul za usporedbu koji na temelju izlučenih značajki i predložaka u bazi podataka klasificira ulazni uzorak
d) Modul koji na temelju rezultata usporedbe iz b) donosi odluku identifikaciji ili verifikacije osobe

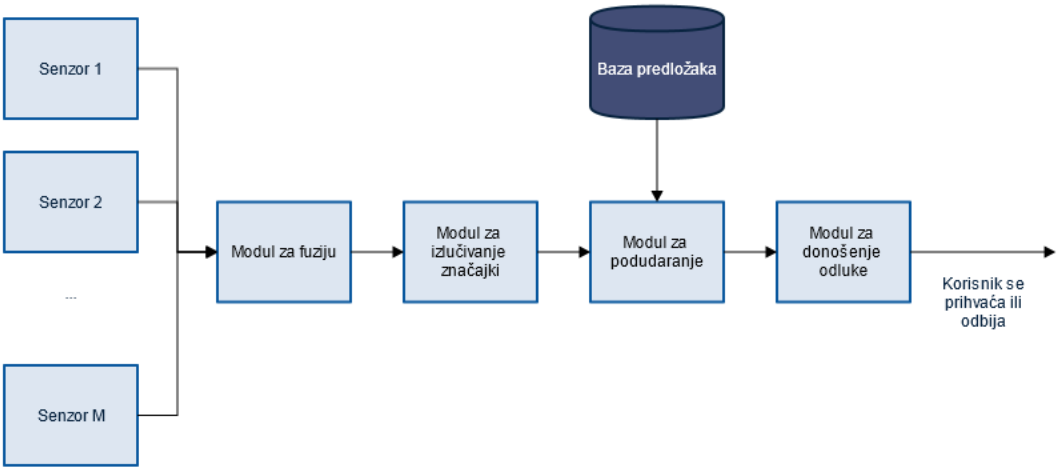
Kombiniranje više biometrijskih karakteristika, odnsno fuzija, moguće je ostvariti na nekoliko različitih načina [17]:

- Na razini senzora
- Na razini vektora značajki
- Na razini mjere podudaranja
- Na razini odluke

## 5.2 Fuzija u biometriji

### 5.2.1 Fuzija na razini senzora

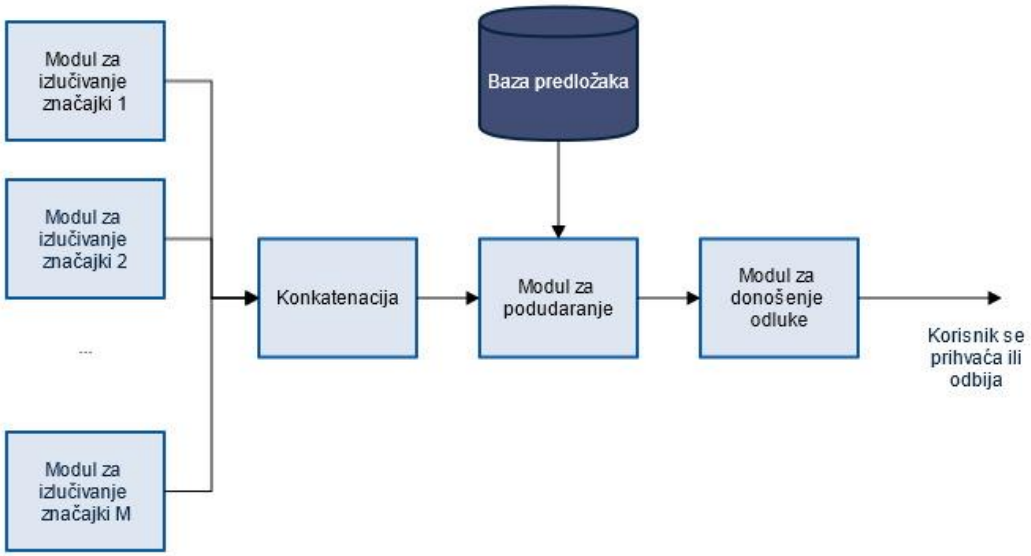
U slučaju fuzije na razini senzora izlazi više različitih senzora kombiniraju se da bi se dobili novi podaci za sljedeću fazu, odnosno izlučivanje značajki. Primjer fuzije na razini senzora može biti slučaj u kojem se kombiniranjem izlaza koji daje videokamera i izlaz senzora koji daje 3D dubinsku informaciju može dobiti 3D teksturirana slika snimljenog objekta [17].




***Slika 21*: Model biometrijskog sustava s fuzijom na razini senzora**

### 5.2.2 Fuzija na razini vektora značajki

Podaci prikupljeni iz svakog senzora se zajedno koriste da bi se dobio vektor značajki. Individualni vektori značajki se spajaju u jedinstveni vektor značajki. Takvi vektori se dalje mogu podudarati s predlošcima spremljenima u bazi podataka.

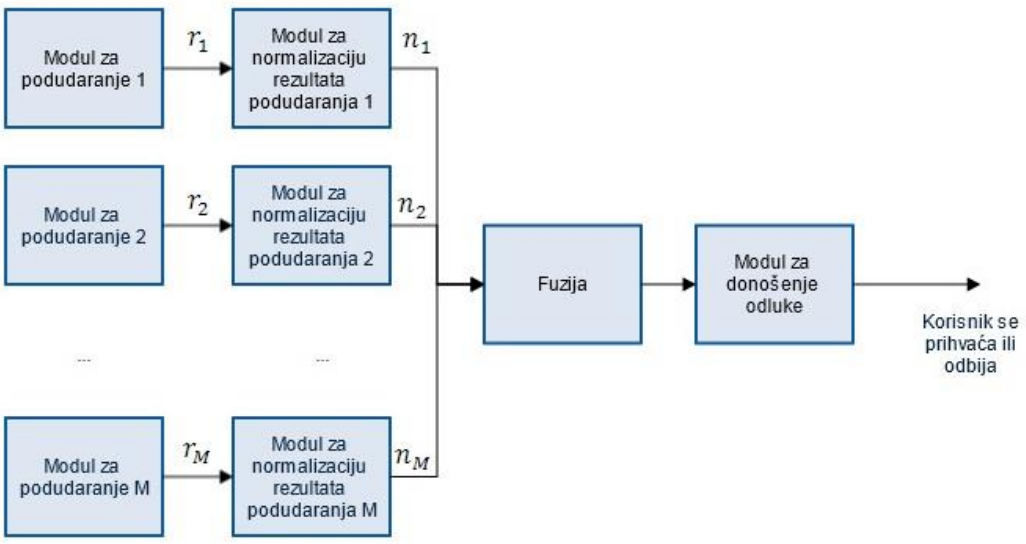



***Slika 22*: Model biometrijskog sustava s fuzijom na razini vektora značajki**

### 5.2.3 Fuzija na razini mjere podudaranja

Kod fuzije na razini mjere podudaranja mjere podudaranja individualnih biometrijskih karakteristika kombiniraju se u jedinstvenu mjeru podudaranja. Zbog jednostavnosti i dobrih rezultata koji se njome mogu postići ova razina fuzije je najčešće korištena.

U ovom tipu fuzije svaki od modula za podudaranje producira mjeru sličnosti ili različitosti biometrijskih značajki trenutnog korisnika i biometrijskih predložaka koji se nalaze u bazi podataka. Označimo te mjere sa $r_1, r_2, \dots, r_M$, gdje je *M* broj modula za podudaranje. Te mjere je prije fuzije potrebno normalizirati nekom od normalizacijskih metoda. Normalizacijom se uvijek dobivaju mjere sličnosti. Njih ćemo ozačiti sa $n_1, n_2, \dots, n_M$. Nakon normalizacije se na temelju nekog pravila fuzije računa ukupna mjera podudaranja (eng. *Total Similarity Measure*, *TSM*).




***Slika 23*: Model dijela biometrijskog sustava s fuzijom na razini mjere podudaranja**

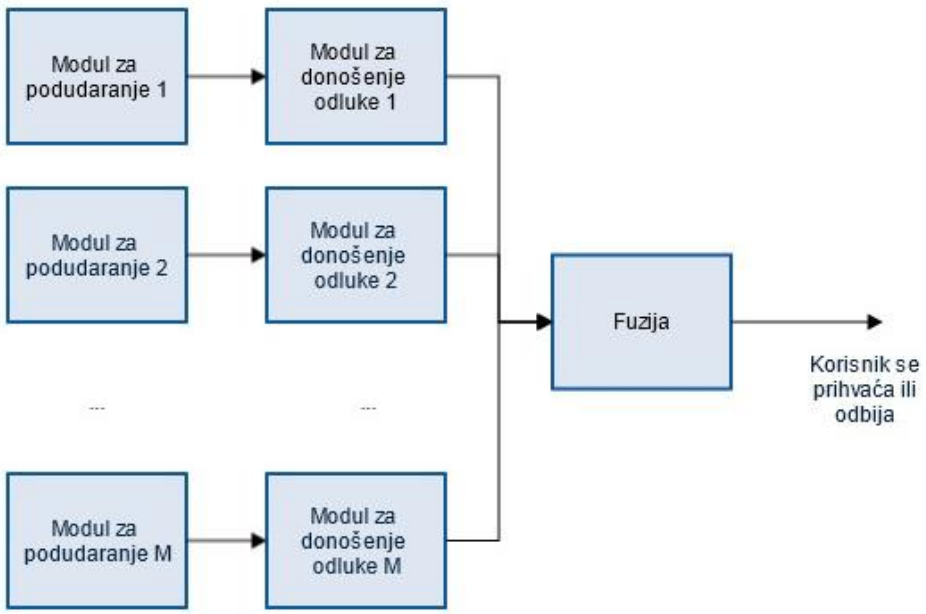
U multimodalnom biometrijskom sustavu s fuzijom na razini mjere podudaranja najčešće nije moguće izravno kombinirati mjere koje daju pojedini moduli za podudaranje zbog sljedećih razloga [17]:

- Mjere podudaranja za različite značajke mogu biti predstavljene na različite načine. Neki moduli za podudaranje unutar istog sustava mogu davati mjere sličnosti, a neki mjere različitosti.
- Izlazi modula za podudaranje mogu ležati u različitim intervalima. Većina tehnika normalizacije se rezultirati mjerama sličnosti u intervalu [0, 1].
- Rezultati podudaranja uzoraka iste osobe i uzoraka različitih osoba za različite karakteristike neće uvijek pratiti iste statističke razdiobe.

Normalizaciju je moguće izvesti na temelju Bayesovog teorema ili nekom od heurističkih metoda (min – max, z – score, median – MAD, tangens hiperbolni, dvostruka sigmoidalna funkcija). U ovom radu je korištena min – max normalizacijska tehnika. Njome se dobivaju vrijednosti u intervalu [0, 1] i očuvana je izvorna distribucija. Min – max za najbolje podudaranje na skupu za učenje daje vrijednost 1, a za najlošije vrijednost 0.

### 5.2.4 Fuzija na razini odluke

Kod ovog tipa fuzije odluke donesene na temelju individualnih biometrijskih karakteristika kombiniraju se u jedinstevenu odluku. Neki od načina donošenja odluke su glasanje (odluka za koju glasa najveći broj modula za donošenje odluke), težinsko glasananje (svakom modulu za donošenje odluke se pridjeljuje neka težina, npr. tako da su težine proporcionalne vjerojatnosti da modul donese ispravnu odluku), a na temelju rezultata dovenih na skupu za učenje može se izgraditi i stablo odluke.




***Slika 24*: Dio modela biometrijskog sustava s fuzijom na razini odluke**

Sustavi koji fuziju vrše u ranijim fazama rada smatraju se efikasnijima od onih koji to rade u kasnijim fazama [18]. Početni vektori značajki sadrže puno više informacija od izlaznih mjera podudaranja pa se očekuje da bi sustavi s fuzijom na razini vektora značajki trebali davati bolje rezultate, ali fuzija na toj razini može biti otežana zbog zbog nekompatibilnosti vektora značajki različitih komponenti. S druge strane, fuzija na razini klasifikacije se smatra previše ograničenom zbog količine informacija kojom se barata na toj razini. Multimodalni sustavi s fuzijom na razini mjere podudaranja se stoga preferiraju jer je rezultate različitih komponenti lako kombinirati.

# 6 Baze slika

U ovom poglavlju slijedi opis baza slika lica i dlanova koje se koriste za testiranje razvijenih biometrijskih sustava temeljenih na analizi glavnih komponenti.

## 6.1 Baza slika lica

Baza slika lica je *Faces94,* preuzeta sa [20]. Baza sadrži slike lica 152 osobe od čega su 132 osobe u bazi muškarci, a 19 žene. Slike su rezolucije 200 × 180 piksela. Po 20 slika svake osobe uzeto je iz video sekvence snimane fiksnom kamerom. Subjekti su tijekom snimanja pričali da bi se na slikama dobile varijacije u izrazima lica.

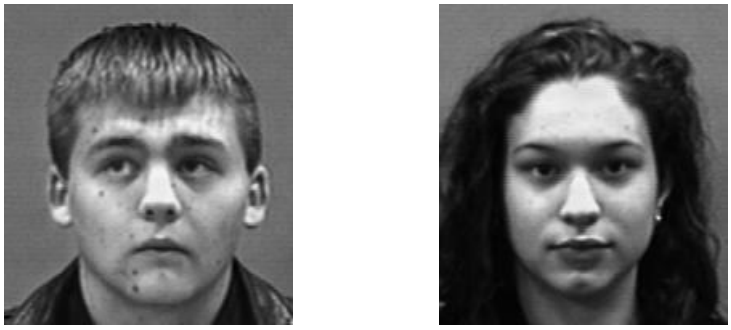
Slike lica u bazi imaju sljedeća svojstva:

- Pozadina je jednoliko zelene boje
- Varijacije u položaju glave su vrlo malene
- Lica su centrirana
- Nema varijacija u osvjetljenju
- Postoje značajne varijacije u izrazima lica

***Slika 25*: Primjeri slika lica**

Prije provođenja *PCA* metoda svaka od slika se pretvara u sivu sliku. Svaki slikovni element originalne slike ima pridružena tri intenziteta u intervalu [0, 255] za crvenu, zelenu i plavu boju, a na temelju kojih se prema formuli (3) dobiva jedinstveni intenzitet sive slike.

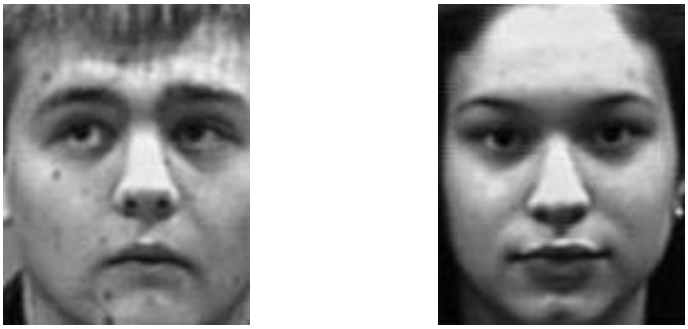
$$I = 0.2989 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{3}$$



***Slika 26*: Dobivene sive slike**

Koeficijenti za određene RGB komponente su ovakvi zbog toga što je ljudsko oko najosjetljivije na zelenu boju, zatim crvenu pa plavu [2].

Nakon pretvaranja originalne slike u sivu sliku potrebno je lokalizirati lice zbog smanjenja utjecaja pozadine na raspoznavanje. Određuje se područje interesa dimenzija 100 × 130 piksela s gornjim lijevim vrhom u točki (40, 50) i ono sadrži lice [2].



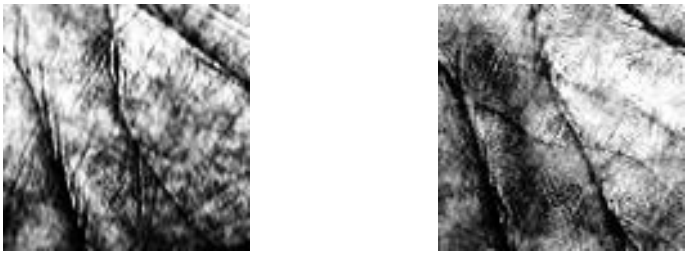
***Slika 27*: Područje interesa koje sadrži lice**

Sljedeći korak jest skaliranje izrezanog područja interesa sa spomenutih dimenzija na manje dimenzije 64 × 64 piksela.

***Slika 28*: Skalirana područja interesa na 64 × 64 piksela**

## 6.2 Baza slika otisaka dlanova

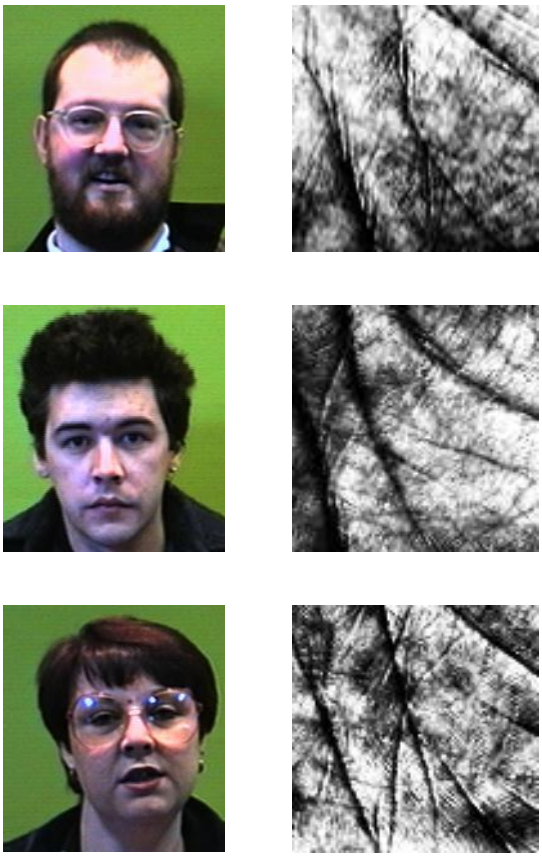
Baza dlanova koja se u ovom radu koristi jest baza PolyU II [21]. Baza sadrži slike otisaka dlanova 386 osoba, no za potrebe ovog rada korištene su slike dlanova prve 152 osobe zbog izgradnje bimodalnog biometrijskog sustava koji se temelji na fuziji značajki lica i dlanova. Proocesiranje slika je uključivalo izrezivanje područja interesa i ujednačavanje histograma zbog razlika u osvjetljenju jer su slike snimane u različitim sesijama [2]. Procesirane slika su dimenzija 96 × 96 piksela pa ih je za potrebe ovog rada još bilo potrbno skalirati na 64 × 64 piksela.



***Slika 29*: Procesirane slika otisaka dlanova**

## 6.3 Himerička baza lica i dlanova

Himerička baza lica i dlanova služi za testiranje bimodalnog biometrijskog sustava za identifikaciju osoba. Dobivena je tako da su sve slike lica uparene sa slikama dlanova, i to na način da jednoj slici lica pripada točno jedna slika dlana, istim redom kojim se pojavljuju u bazama.

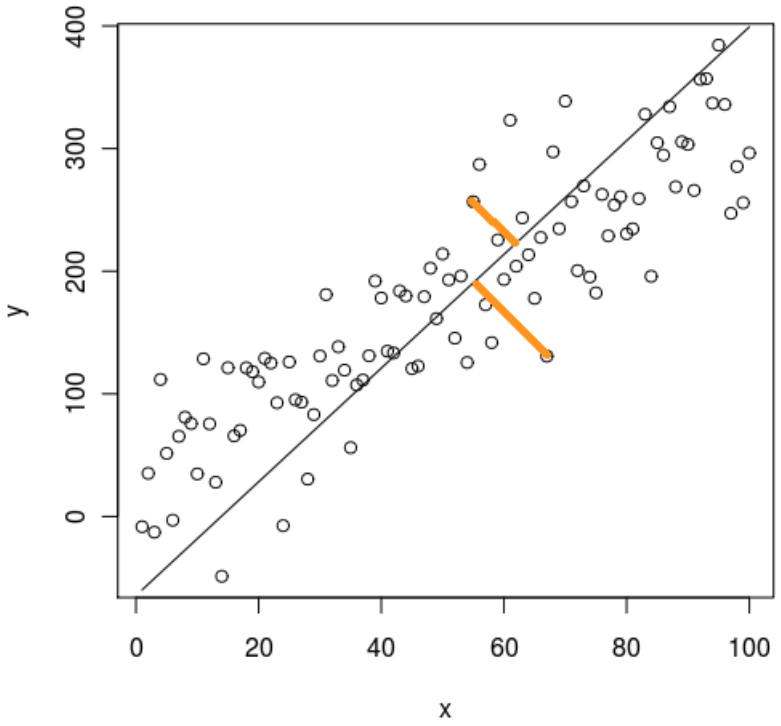


***Slika 30*: Primjeri parova lica i otiska dlanova za prve tri osobe u bazi**

Obje baze su organizirane na način da sadrže po 152 direktorija koji su imenovani brojevima od 1 do 152 radi lakšeg baratanja oznakama kod implementacije.

# 7 Analiza glavnih komponenti i njezine inačice

Analiza glavnih komponenti (eng. *Principal Component Analysis, PCA*), poznata i pod nazivom Karhunen – Loève transformacija, je metoda kojoj je cilj pronaći potprostor izvornog prostora koji je optimalan za reprezentaciju danog skupa uzoraka u smislu da je udaljenost tako kodiranih uzoraka kada ih se vrati u originalni prostor i originalnih uzoraka minimalna. Vektori koji razapinju takav potprostor nazivaju se glavne komponente (eng. *Principal Components*). *Slika 31* prikazuje jednostavan primjer metode *PCA* u 2D prostoru. Pravac označava prvu glavnu komponentu, odnosno jedini vektor koji predstavlja bazu potprostora u koji će se točke projicirati.




***Slika 31:*** **Primjer izvornih podataka, prve glavne komponente (pravac) i točke u koje će se originalni podaci projicirati (označeno narančasto)**

Iako ova metoda ne pronalazi nužno one komponente koje su optimalne za raspoznavanje, već one za reprezentaciju, rezultati dobiveni uz pomoć metode *PCA*

pokazuju se dobri u raznim primjenama. Metoda *PCA* je vrlo dobra u slučajevima kada imamo uzorke velike dimenzionalnosti jer se njome smanjuje njihova dimenzionalnost, ali se zadržavaju svojstva prepoznatljivosti uzoraka. Smanjenjem dimenzionalnosti uzoraka memorijski zahtjevi sustava padaju, a brzina raste.

## 7.1 Analiza glavnih komponenti za skup slikovnih uzoraka

U ovom potpoglavlju će biti dan općeniti *PCA* algoritam za skup slikovnih uzoraka. Pretpostavimo da imamo *N* uzoraka. Svaki od *N* uzoraka sadrži *n* slikovnih elemanta. Svaki uzorak se prikazuje kao vektor stupac $\mathbf{s_i}, \mathrm{i} \in \{1, \dots, \mathrm{N}\}$. Prvi korak jest računanje srednje vrijednosi svih vektora kao

$$\mathbf{m} = \frac{1}{\mathrm{N}} \sum_{\mathrm{i}=1}^{\mathrm{N}} \mathbf{s_i} \tag{4}$$

nakon čega treba oduzeti tu srednju vrijednost od svakog uzorka

$$\mathbf{a_i} = \mathbf{s_i} - \mathbf{m}, \qquad i \in \{1, \dots, N\} \tag{5}$$

Sljedeći korak je postavljanje vektora $\mathbf{a_i}$ u matricu $\mathbf{A}$

$$\mathbf{A} = [\mathbf{a_1 a_2} \dots \mathbf{a_N}] \tag{6}$$

Matrica $\mathbf{A}$ je dimenzija *n* × *N.* Da bi se izračunao optimalan potprostor moramo pronaći svojstvene vektore $\mathbf{v}_\mathrm{k}$ kovarijacijse matrice koju dobivamo kao

$$\mathbf{C} = \mathbf{A}\mathbf{A}^T \tag{7}$$

Matrica $\mathbf{C}$ je dimenzija $n \times n$ što je u praksi često vrlo veliko (npr. ako se računa PCA za skup slika dimenzija 64 × 64, matrica $\mathbf{C}$ će imati dimenzije 4096 × 4096 pa bi računanje svojstvenih vektora ovakve matrice bilo vrlo dugotrajno). S obzirom da često vrijedi $N < n$, odnosno broj slika na raspolaganju je manji od broja slikovnih elementa svake slike, svojstveni vektori matrice $\mathbf{C}$ se ne računaju izravno već ih je moguće izračunati preko svojstvenih vektora zamjenske matrice $\mathbf{C}'$

$$\mathbf{C}' = \mathbf{A}^T\mathbf{A} \tag{8}$$

Ukoliko vrijedi $N < n$ broj svojstvenih vrijednosti različitih od 0 će biti $N - 1$, a tolika će biti i maksimalna dimenzionalnost novonastalog potprostora. Matrica $\mathbf{C}'$ je dimenzija $N \times N$. Nakon izračuna svojstvenih vektora $\mathbf{v}_k'$ i pripadajućih svojstvenih vrijednosti $\boldsymbol{\lambda}_k'$ matrice $\mathbf{C}'$, mogu se izračunati svojstveni vektori i pripadajuće svojstvene vrijednosti matrice $\mathbf{C}$

$$\boldsymbol{\lambda}_{\mathrm{k}} = \boldsymbol{\lambda}_k', \qquad k = 1,2,\dots,N \tag{9}$$

$$\mathbf{v}_{\mathrm{k}} = \mathbf{A}\mathbf{v}_k', \qquad k = 1,2,\dots,N \tag{10}$$

Potprostor u koji ćemo projicirati uzorke biti će razapet vektorima $\mathbf{v}_{\mathrm{k}}$ s najvećim pripadajućim svojstvenim vrijednostima $\boldsymbol{\lambda}_{\mathrm{k}}$. Svaki uzorak se projicira u odabrani potprostor kao

$$\eta_k = \mathbf{v}_k^T\mathbf{a}_s, \qquad k \in \{1,\dots,N_{PCA}\} \tag{11}$$

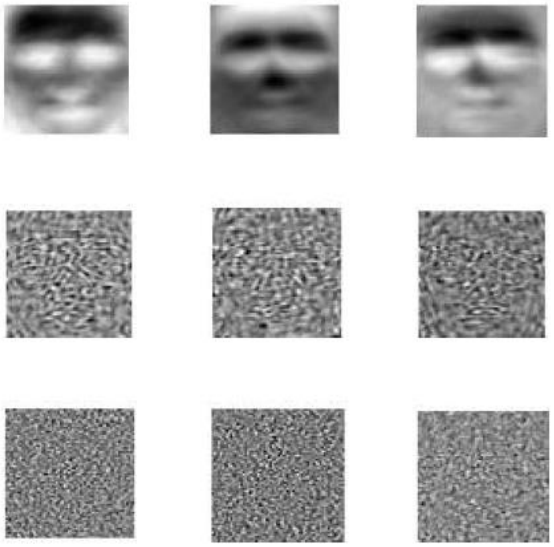
gdje je $N_{PCA}$ dimenzionalost potprostora značajki, a $\mathbf{a}_s = \mathbf{s_s} - \mathbf{m}$, pri čemu je $\mathbf{s_s}$ predstavlja uzorak (sliku) koji želimo prikazati u potprostoru manje dimenzionalnosti od originalnog. Komponente $\eta_k, k \in \{1, \dots, N_{PCA}\}$ predstavljaju vektor koji reprezentira uzorak u dobivenom potprostoru. Taj vektor će se koristiti kao reprezentacija slike u daljnjem postupku raspoznavanja.

Pripadnost nepoznatog uzorka $\mathbf{x}$ nekom od razreda određuje se tako da se on prvo projicira u dobiveni potprostor prema formuli (14), a zatim se prema nekom klasifikacijskom algoritmu određuje razred kojemu pripada. U ovom radu korišten je 1 – NN algoritam.

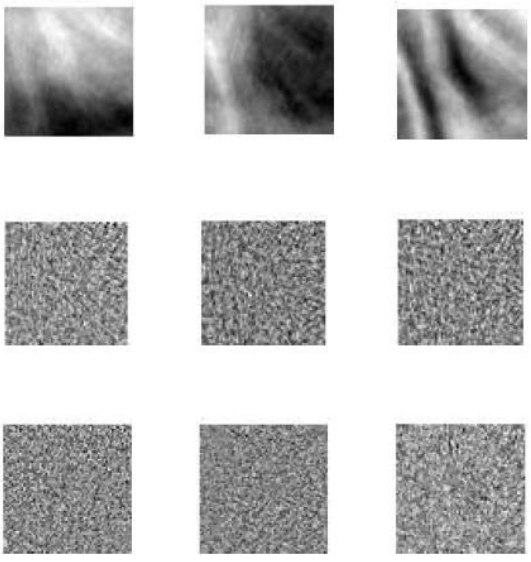
$$\mathbf{x'}_k = \mathbf{v}_k^T \cdot (\mathbf{x} - \mathbf{m}), \qquad k \in \{1, \dots, N_{PCA}\} \tag{12}$$

*Slika 32*, 33 i 34 prikazuju primjere srednjih uzoraka i svojstvenih vektora koji proizlaze kao rezultat osnovne inačice metode PCA.

***Slika 32*: Primjer srednjeg lica i srednjeg dlana**



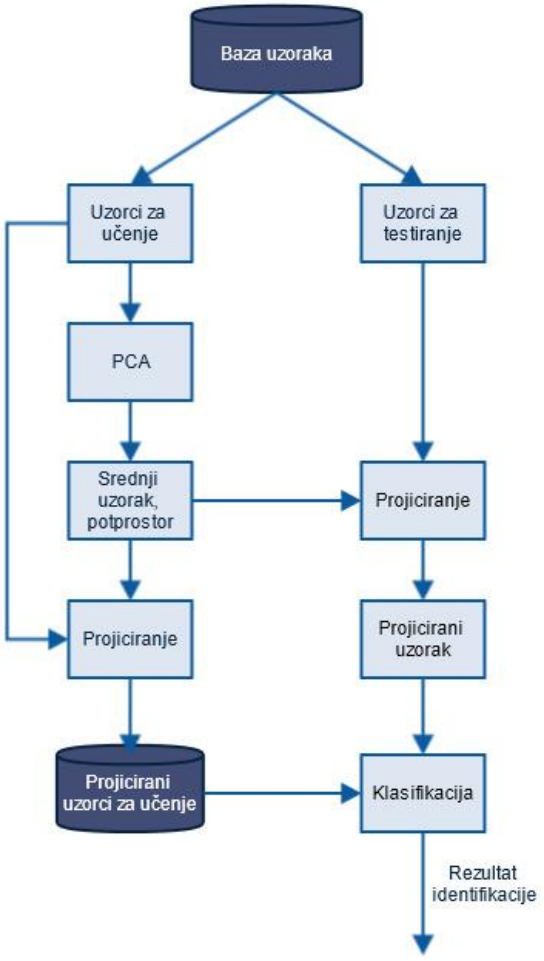
***Slika 33*: Primjeri svojstvenih vektora (svojstvenih lica). Slike prvog retka prikazuju svojstvene vektore s tri najveće pripadajuće svojstvene vrijednosti. Srednji redak prikazuje svojstvena lica koja odgovaraju trima svojstvenim vrijednostim u sredini, a zadnji redak one s tri najmanje svojstvene vrijednosti. Ukupan broj svojstvenih vektora u ovom primjeru je 755.**



***Slika 34:* Primjer svojstvenih vektora (svojstvenih lica). Slike prvog retka prikazuju svojstvene vektore s tri najveće pripadajuće svojstvene vrijednosti. Srednji redak prikazuje svojstvena lica koja odgovaraju trima svojstvenim vrijednostim u sredini, a zadnji redak one s tri najmanje svojstvene vrijednosti. Ukupan broj svojstvenih vektora u ovom primjeru je 755.**

## 7.2 Inačice metode PCA

Kod osnovne varijante metode PCA opisane u prošlom poglavlju sve slike, neovisno kojem razredu pripadaju, prije projekcije se nalaze u zajedničkom *n* dimenzionalnom prostoru, a nakon projekcije u zajedničkom $N_{PCA}$ dimenzionalnom potprostoru.




***Slika 35*: Dijagram sustava za provođenje osnovne metode PCA**

Originalnu metodu je moguće modificirati na način da se opisani postupak provodi posebno za svaki razred.

### 7.2.1 Inačica I

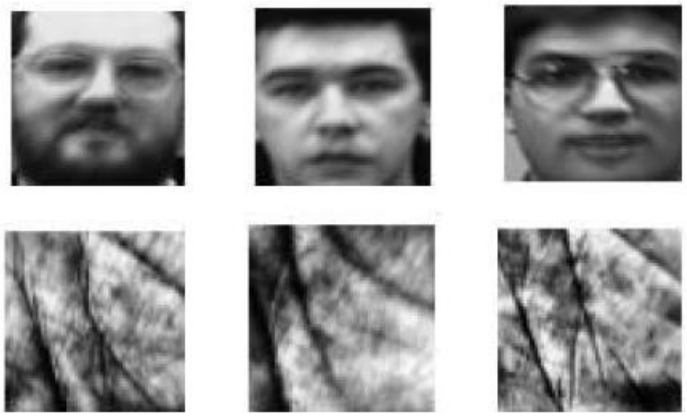
Kod inačice I metoda PCA se provodi nad svakim od *P* razreda posebno. Rezultat postupka jest ukupno *P* srednjih lica i *P* dobivenih potprostora (prostora lica i dlanova), odnosno po jedno srednje lice i jedan prostor lica ili dlanova koji se koriste za projekciju uzoraka iz toga razreda. Obrada nepoznatog uzorka je nešto drugačija u odnosu na originalnu metodu. Nepoznati uzorak $\mathbf{x}$ se projicira u svaki od *P* potprostora prema formuli (15).

$$\mathbf{x'}_{i,k} = \mathbf{v}_{i,k}^{T} \cdot (\mathbf{x} - \mathbf{m_i}), \qquad k \in \{1, \dots, N_{PCA}\}, i \in \{1, \dots, P\} \tag{13}$$

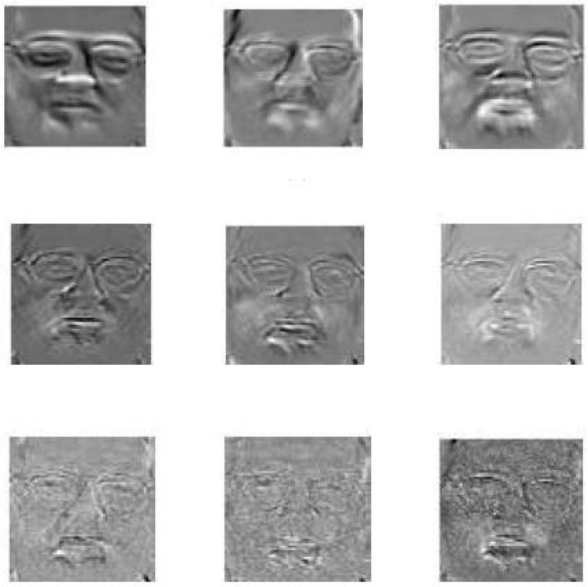
Nakon projiciranja uzorci $\mathbf{x}_i'$, $i \in \{1, \dots, P\}$ se uspoređuju s projiciranim uzorcima *i*-tog razreda te se prema nekom klasifikacijskom algoritmu određuje kojem razredu oni pripadaju.

U slučaju inačice I potrebno je primjetiti da je najveća moguća dimenzionalnost svakog od *P* potprostora *M,* gdje je *M* broj uzoraka u svakom od *P* razreda, uz pretpostavku da svaki razred sadrži jednak uzoraka.

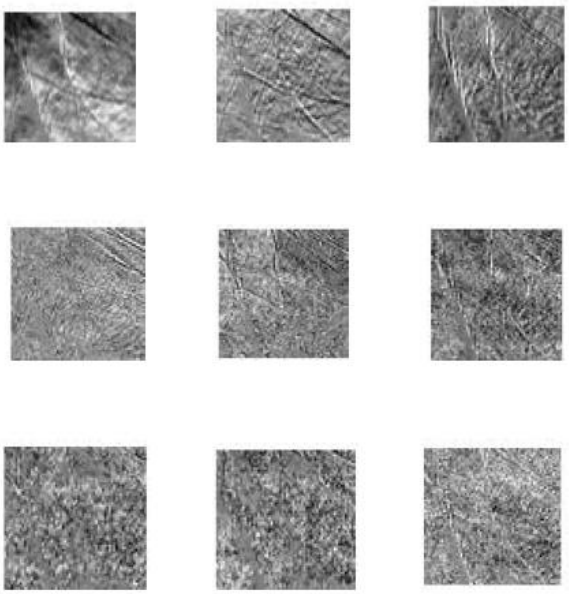
Slike 35, 36 i 37 prikazuju primjere srednjih uzoraka i svojstvenih vektora koji proizlaze kao rezultat osnovne inačice I metode PCA.



***Slika 36*: Srednja lica i dlanovi**



***Slika 37:*** **Primjer svojstvenih vektora (svojstvenih lica). Slike prvog retka prikazuju svojstvene vektore s tri najveće pripadajuće svojstvene vrijednosti. Srednji redak prikazuje svojstvena lica koja odgovaraju trima svojstvenim vrijednostim u sredini, a zadnji redak one s tri najmanje svojstvene vrijednosti. Ukupan broj svojstvenih vektora u ovom primjeru je 20.**



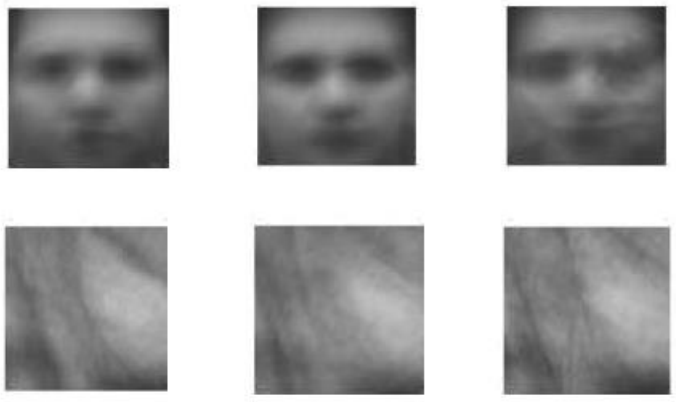
***Slika 38:*** **Primjer svojstvenih vektora (svojstvenih dlanova). Slike prvog retka prikazuju svojstvene vektore s tri najveće pripadajuće svojstvene vrijednosti. Srednji redak prikazuje svojstvena lica koja odgovaraju trima svojstvenim vrijednostim u sredini, a zadnji redak one s tri najmanje svojstvene vrijednosti. Ukupan broj svojstvenih vektora u ovom primjeru je 20.**

### 7.2.2 Inačica II

Kao i u inačici I skup uzoraka za učenje podijeljen je po razredima. Razlika u odnosu na inačicu I je u tome što se svaki razred proširuje sa jednom nasumično odabranom slikom iz svakog razreda. Moguće je uzeti i proizvoljan broj slika iz drugih razreda, no zbog očuvanja specijaliziranosti skupa za učenje uzet će se samo jedna. Ovakvom modifikacijom svaki razred sada umjesto $M$ uzoraka sadrži $M + P - 1$ uzorak. U konkretnom slučaju svaki razred će sadržavati 20 slika koje su iz tog razreda i 151 sliku koja nije $(20 + 152 - 1)$. Razlog umjetnog proširivanja skupova za učenje je povećanje dimenzionalnosti potprostora u koji će se originalni uzorci projicirati. Pretpostavka je da će povećanje dimenzionalnosti potprostora pridonositi točnosti identifikacije. Ptrebno je napomenuti da se nakon provođenja metode PCA u odgovarajuće potprostore projiciraju samo oni uzorci koji originalno pripadaju tome razredu (svi ili neki njihov podskup), a ne i oni koji su korišteni za umjetno proširenje razreda, odnosno oni koji pripadaju drugim razredima.

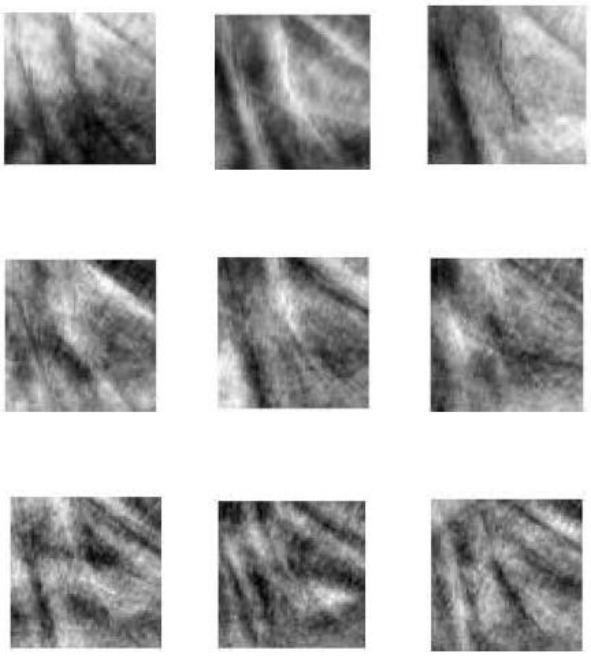
Obrada nepoznatog uzorka provodi se na jednak način kao i kod inačice I. Jedina razlika je maksimalna dimenzionalnost svakog od *P* potprostora – u inačici I ona iznosi $M$, a u inačici II $M + P - 1$.

Slike 38, 39 i 40 prikazuju primjere srednjih uzoraka i svojstvenih vektora koji proizlaze kao rezultat osnovne inačice I metode PCA.

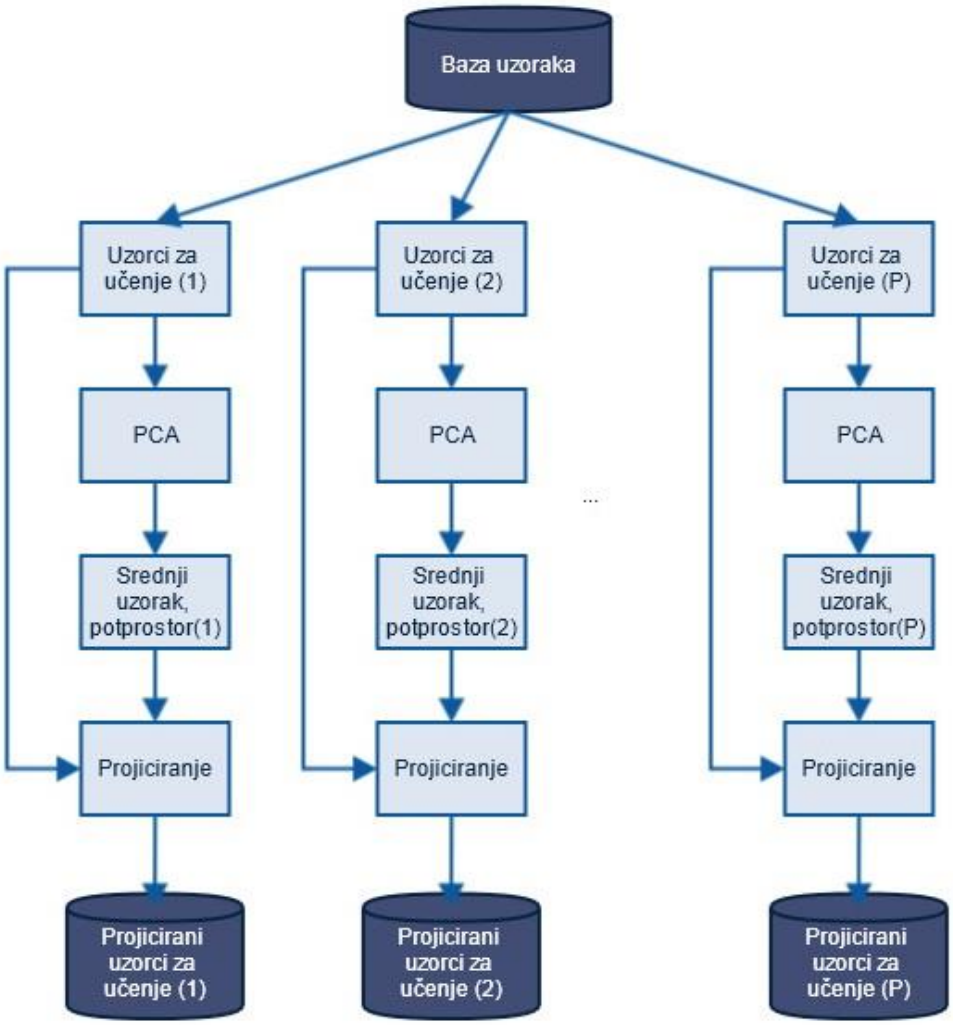


***Slika 39*: Srednja lica i dlanovi**

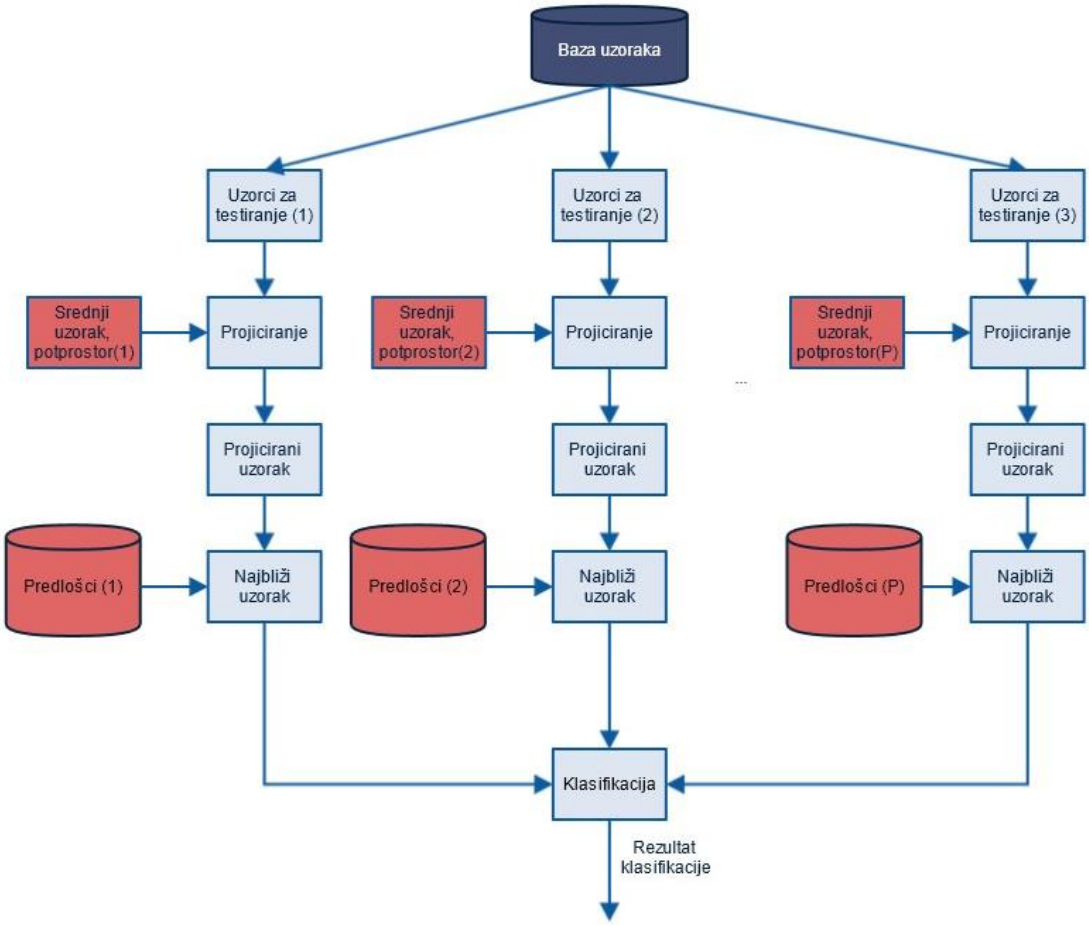
***Slika 40*: Primjer svojstvenih vektora (svojstvenih lica). Slike prvog retka prikazuju svojstvene vektore s tri najveće pripadajuće svojstvene vrijednosti. Srednji redak prikazuje svojstvena lica koja odgovaraju trima svojstvenim vrijednostim u sredini, a zadnji redak one s tri najmanje svojstvene vrijednosti. Ukupan broj korištenih svojstvenih vektora u ovom primjeru je 20.**



***Slika 41*: Primjer svojstvenih vektora (svojstvenih dlanova). Slike prvog retka prikazuju svojstvene vektore s tri najveće pripadajuće svojstvene vrijednosti. Srednji redak prikazuje svojstvena lica koja odgovaraju trima svojstvenim vrijednostim u sredini, a zadnji redak one s tri najmanje svojstvene vrijednosti. Ukupan broj korištenih svojstvenih vektora u ovom primjeru je 20.**




***Slika 42*: Dijagram sustava za provođenje inačica metode PCA, faza učenja**




***Slika 43*: Dijagram sustava za provođenje inačica metode PCA, faza testiranja. Crvenom bojom su označene komponente koje su dobivene u fazi učenja. Predlošci predstavljaju projicirane uzorke za učenje.**

## 7.3 Klasifikacija nepoznatih uzoraka

Nakon što se obavi redukcija dimenzionalnosti nepoznatog uzorka, istog je potrebno i klasificirati, odnosno pridjeliti mu oznaku jednog od poznatih razreda. U ovom radu je korišten algoritam *k najbližih susjeda* (eng. *k Nearest Neighbors, k – NN*).

Spomenuti klasifikator je jedan od najjednostavnijih klasifikatora – za nepoznati uzorak odredi se njemu *k* najbližih susjeda te se on svrstava u onaj razred koji se među tih *k* susjeda pojavljuje najviše puta. Pri odabiru *k* je potrebno paziti da on bude neparan jer bi se u suprotnom mogla dogoditi situacija u kojoj bi postojala nejednoznačnost prilikom odabira razreda. Najjednostavniji slučaj *k – NN* klasifikacije je za *k = 1*. U tom slučaju se nepoznati uzorak svrstava u onaj razred u kojem se nalazi i njemu najbliži uzorak. Za određivanje *k* najbližih susjeda potrebno je odrediti koja će se mjera udaljenosti definirana u prostoru značajki koristiti. To mogu biti Manhattan ($L_1$) udaljenost, euklidska ($L_2$) udaljenost, Čebiševljeva ($L_\infty$) udaljenost, Mahalanobisova udaljenost itd.

Za dani primjer $\mathbf{x}$ s nepoznatom klasifikacijom i *k* uzoraka $\mathbf{x}_1, \mathbf{x}_2, \dots, \mathbf{x}_\mathrm{k}$ koji su najbliži $\mathbf{x}$ algoritam vraća

$$h(\mathbf{x}) = \underset{v \in \{0,1,\dots,K\}}{\operatorname{argmax}} \sum_{i=1}^{k} \delta(v, y_i) \tag{14}$$

gdje je $y_i$ oznaka razreda uzorka kojemu pripada susjed $\mathbf{x}_i$, $\delta(a,b) = 1$ ako $a = b$, 0 inače. $h(\mathbf{x})$ je najčešća vrijednost ciljne funkcije koja se pojavljuje među *k* primjera za učenje koji su najbliži nepoznatom uzorku $\mathbf{x}$.

K – NN algoritam spada u skupinu *lijenih* metoda. Takve metode odgađaju odluku o klasifikaciji sve do trenutka predočavanja nepoznatog uzorka, odnosno, umjesto procjene ciljne funkcije jednom za cijeli prostor one procjenjuju ciljnu funkciju samo lokalno, u okolini nepoznatog uzorka. Nedostatak takvog pristupa je visoka cijena klasificiranja nepoznatog uzorka jer je potrebno razmotriti sve njegove značajke. Upravo

taj nedostatak se rješava upotrebom metode *PCA* koja značajno reducira dimenzionalnost uzoraka.

# 8 Izgrađeni biometrijski sustavi

U ovom poglavlju će biti navedene osnovne karakteristike izgrađenih biometrijskih sustava. Unimodalni sustavi temelje se na licu i otisku dlana zasebno, a multimodalni na licu i otisku dlana zajedno. Eksperimenti vršeni nad unimodalnim sustavima će u nastavku biti označavani kao *eksperiment uN*, a eksperimenti vršeni nad multimodalnim sustavima kao *eksperiment mN*, gdje je N redni broj eksperimenta.

## 8.1 Unimodalni sustavi

Unimodalni biometrijski sustavi testirani su u tri različita eksperimenta. U prvom eksperimentu (eksperiment u1) testira se klasična metoda *PCA* za redukciju dimenzionalnosti i 1 – NN klasifikacijski algoritam za određivanje razreda nepoznatog uzorka. Ovaj eksperiment je služio samo kao prototip za testiranje metode *PCA* i dobivanje referentnih podatak o točnosti. U drugom eksperimentu (eksperiment u2) testira sa inačica I metode *PCA,* opisana u poglavlju 7.2.1. Treći eksperiment (eksperiment u3) testira inačicu II metode *PCA,* opisanu u poglavlju 7.2.2. Testiranje u sklopu eksperimenta 1 je bilo vršeno unakrsnom validacijom sa 4 preklopa (eng. *4 – folded cross validation*). U svakoj iteraciji (preklopu) je korišteno 5 slika za učenje i 15 za tesiranje. Eksperimenti 2 i 3 testiranje su obavljali na temelju *leave – one – out* metode gdje se po 19 slika iz pojedinog razreda koristilo za učenje, a jedna za testiranje. Dodatno, u eksperimentu 3 koristilo se i proširenje svakog razreda, kako je objašnjeno u poglavlju 7.2.2

Testirane su i paralelne inačice eksperimenata u2 i u3 s ciljem ocjene faktora ubrzanja paralelne implementacije u odnosu na serijsku. Paralelizirana je samo faza testiranja.

## 8.2 Multimodalni sustavi

Multimodalni biometrijski sustavi testirani su u 2 eksperimenta. Prvi eksperiment (eksperiment m2) testira sustav koji se temelji na inačici I metode *PCA*. Drugi eksperiment temelji se na inačici II metode *PCA* (eksperiment m3). U oba eksperimenta fuzija se obavlja na razini mjere podudaranja koja je opisana u poglavlju 5.2.3. *Slika 23* prikazuje dijagram takvog sustava. S obzirom da je izlaz iz svakog modula za podudaranje mjera različitosti, istu je potrebno normalizirati. Korištena je max – min normalizacijska tehnika koja za mjere različitosti glasi

$$n_i = \frac{\max(O_i) - r_i}{\max(O_i) - \min(O_i)}, i = 1,2 \dots, N \tag{15}$$

gdje je $r_i \in O_i$ mjera različitosti dobivena za pojedinu biometrijsku karakteristiku.

Pravila fuzije koje se koriste za dobivanje *TSM* je pravilo težinske sume

$$TSM = \sum_{i=1}^{N} w_i n_i \tag{16}$$

Kod težinske sume uobičajeno je da je suma težinskih faktora $w_i$ jednaka jedan. Težinski faktori se postavljaju tako da budu proporcionalni rezultatima rasponzavanja pripadajućih unimodalnih sustava. Ako su $p_1$ i $p_2$ točnosti unimodalnih sustava temeljenih na licu i dlanu tada će se faktor $w_1$ računati kao $\frac{p_1}{p_1+p_2}$, a faktor $w_2$ kao $\frac{p_2}{p_1+p_2}$. Za različite brojeve korištenih svojstvenih vektora potrebno je izračunati i nove težinske faktore.

# 9 Opis implementacije multimodalnih biometrijskih sustava

Osnova svakog implementiranog biometrijskog sustava je razred koji implementira skup funkcija koje su potrebne za provođenje metode *PCA*. Razred interno čuva srednji uzorak dobiven prilikom provođenja metode, svojstvene vektore (svojstvena lica ili dlanovi) koji razapinju dobiveni potprostor, skup predložaka (projicirani uzorci koji pripadaju tome razredu) i oznaku pripadajućeg razreda, odnosno osobe. Srednji uzorak i vektori koji čine potprostor koriste se za projekciju nepoznatog uzorka, a predlošci za računanje udaljenosti projiciranog uzorka od uzoraka koji pripadaju tome razredu. Metoda *PCA* provodi se prema algoritmu danom u poglavlju 7.1.

Radi lakšeg učitavanja slika unutar programa, stvorene su dvije datoteke, po jedna za lica i dlanove, koje u svakom retku sadrže putanju do slike i oznaku razreda kojem ta slika pripada.

Prije provođenja metode *PCA* sve slike iz baze se učitavaju, pretprocesiraju, pretvaraju u vektor – stupce i slažu u matrice (jedan razred, jedna matrica). Nakon toga se dijele na skup za učenje i skup za testiranje. Sustavi se testiraju u 20 iteracija *leave – one – out* metode unakrsne validacije.

Faza učenja je jednaka i za serijsku i paralelnu implementaciju.

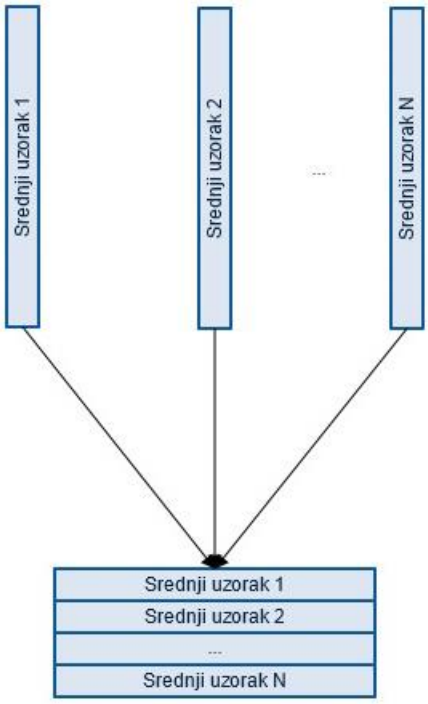
## 9.1 Paralelna implementacija faze identifikacije

Da bi se postigle najbolje performanse paralelne implementacije bilo kojeg paralelnog *CUDA* programa, preporuča se pratiti ove smjernice [22]:

1. Odrediti način paralelilazije – dekompozicija početnog problema u paralelne potprobleme
2. Minimizirati broj prijenosa podataka između *host* i *device* strana
3. Podesiti parametre konfiguracije izvršavanja kernela (broj dretvi i broj blokova dretvi) tako da se grafički procesor maksimalno iskoristi, odnosno da najveći dio vremena koristi za računanje
4. Pokušati globalnom memoriji pristupati sjedinjeno
5. Minimizirati redundante pristupe globalnoj memoriji kada god je to moguće
6. Izbjegavati divergenciju kernela kada god je to moguće

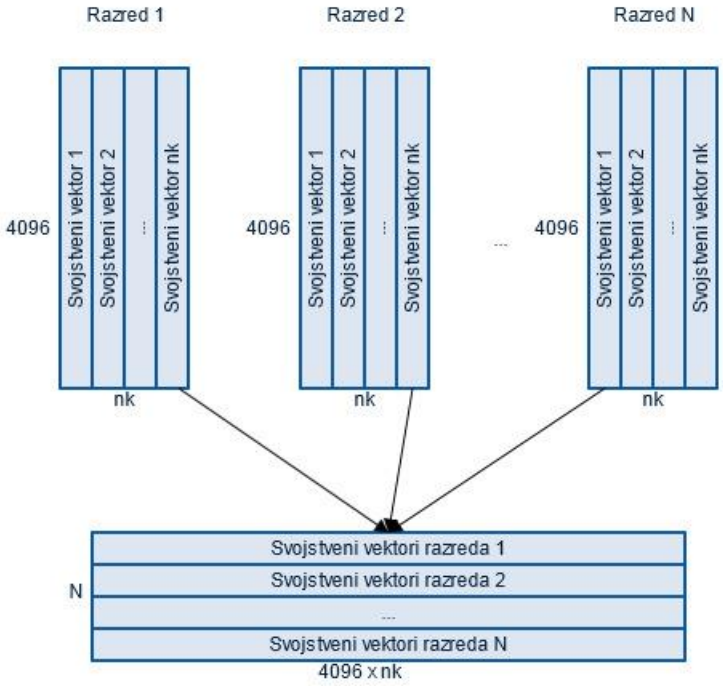
Prije provođenja identifikacije nepoznatog uzorka nužno je sve potrebne podatke preseliti u memoriju grafičkog procesora. Osim prijenosa nepoznatog uzorka to uključuje i prijenos svojstvenih vektora, srednjeg uzorka i predložaka za svaki razred. S obzirom na velik broj razreda u bazi pojedinačni prijenosi navedenih podataka bili bi vremenski vrlo zahtjevni pa je bilo potrebno pronaći način da se broj prijenosa smanji. Svi navedeni podaci organizirani su na slijedeći način:

- Svi srednji uzorci su smješteno po retcima u jednu matricu. Broj redaka je jednak broju razreda (152), a broj stupaca dimenzionalnosti uzorka (4096).



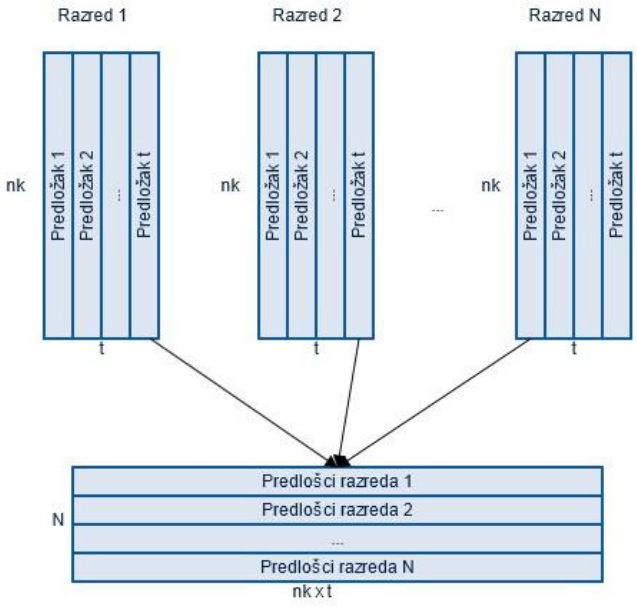

***Slika 44*: Organizacija srednjih uzoraka**

- Svojstveni vektori pojedinog razreda inicijalno se nalaze u matrici kao vektori stupci. Prije kopiranja reorganizirani su tako da tvore jedan redak nastao uzastopnim konkateniranjem redaka inicijalne matrice. Tako dobiveni retci svakog razreda se zatim slažu u jednu matricu koja ima broj redaka jednak broju razreda (152), a broj stupaca jednak je $n_k$ × 4096, gdje je $n_k$ broj svojstvenih vektora.




***Slika 45*: Organizacija svojstvenih vektora**

- Analogno organizaciji svojstvenih vektora organizirani su i predlošci. Dimenzija matrice u kojo su smješteni predlošci svih razread je dimenzija 152 × $(n_k \cdot t)$, gdje je $n_k$ broj korištenih svojstvenih vektora, a $t$ broj predložaka koji su u tome razredu.




***Slika 46*: Organizacija predložaka**

Ovakvom organizacijom podataka ukupan broj prijenosa sa *host* u *device* memoriju sveden je na samo četiri memorijske transakcije. Nakon što se svi potrebni podaci nalaze u memoriji grafičkog procesora moguće je provesti samu identifikaciju.

Kernel koji vrši projekciju nepoznatog uzorka ukupno se izvodi *N* puta, odnosno po jednom za svaki razred. Model takvog paralelnog izvođenja može se poistovjetiti dijagramom kojeg prikazuje slika 43. Niz instrukcija kernela koje izvodi jedna dretva moguće je poistovjetiti sa nizom instrukcija jedne iteracije *for* petlje. Zbog načina na koji su podaci organizirani svaka dretva do potrebnih podataka dolazi na vrlo jednostavan način – svaka dretva zna svoj indeks, a taj indeks koristi za pristup određenom retku svake od navedenih matrica koje sadrže srednje uzorke, svojstvene vektore ili predloške.

S obzirom na veliku količinu podataka s kojom svaka dretva barata, nije moguće iskoristiti brze memorije poput dijeljene memorije ili registara, već se svi podaci moraju dohvaćati iz spore globalne memorije. Dodatan problem je što su pristupi memoriji nesjedinjeni jer svaka dretva dohvaća cijele blokove podataka, a uzastopne dretve (gledajući njihove indekse) ne pristupaju uzastopnim memorijskim lokacijama. No, ovaj problem se može ublažiti na način da se iz memorije čita više podataka odjednom zapakiranih u neku strukturu. Primjer takve strukuture je vektorski tip *float4* ugrađen u *CUDA* C jezik. On interno sadrži četiri *float* varijable koje u memoriji zauzimaju 16 uzastopnih bajtova (po četiri za svaku varijablu). Korištenjem ove strukture broj iteracija *for* petlje, a samim time i broj pristupa globalnoj memoriji, smanjuje se 4 puta. Osim toga, moguće je koristiti i odmotavanje petlji (eng. *loop unrolling*), tehniku za transformaciju petlje koja se koristi za optimiziranje vremena izvođenja programskog koda, ali na štetu veličine izvršnog koda.

Nakon oduzimanja srednje vrijednosti od  uzorka on se projicira u potprostor (prostor lica ili dlanova) odgovarajućeg razreda. Projiciranje se svodi na $n_k$ skalarnih produkata između usrednjenog uzorka i svojstvenih vektora toga razreda, gdje je $n_k$ ukupan broj svojstvenih vektora. Prilikom projekcije je također korišteno odmotavanje (unutarnje) petlje koja računa skalrni produkt.

Slijedeći korak je računanje minimalne udaljenosti od projiciranog uzorka do svih predložaka. Minimalne udaljenosti se smještaju u polje nad kojim se primjeni paralelna

redukcija sa *min* operatorom te se pronađe minimalna udaljenost od svih izračunatih. Nakon toga se odredi njen indeks u polju minimalnih udaljenost. Taj indeks je zapravo oznaka razreda u koji se nepoznati uzorak klasificirao.

Kod multimodalnih sustava pozivi kernelu koji je zadužen za projekciju uzorka i računanje minimalne udaljenosti su serijalizirani, odnosno prvo se u paraleli obradi uzorak koji odgovara licu, a zatim uzorak koji odgovara dlanu.

# 10 Rezultati

U poglavlju 10.1 će biti opisani izvedeni eksperimenti, a u poglavlju 10.2 će se izložiti rezultati dobiveni u pojedinom eksperimentu te će biti uspoređena vremena izvođenja serijske i paralelne implementacije.

## 10.1 Opis provedenih eksperimenata

### 10.1.1 Testiranje klasišne metode *PCA* – eksperiment 1

Kao što je navedeno u poglavlju 8.1, eksperiment sa klasičnom metodom *PCA* se proveo da bi se dobili referentni podaci o točnosti unimodalnih sustava temeljenih na licu ili dlanu. U ovom eksperimentu metoda *PCA* je testirana *4 – folded* unakrsnom validacijom gdje se po pet slika lica, odnosno dlana koristilo za učenje, a petnaest za testiranje. Na temelju zapisa iz unaprijed uređene tekstualne datoteke koja sadrži putanje do slika iz baze i oznake razreda sve slike se učitavaju u program (slike jednog razreda se učitavaju u jedno polje) te se na skupove za učenje i testiranje dijele na slijedeći način:

- U prvoj od četiri iteracije unakrsne validacije učitane slike s indeksima 1, 5, 9, 13 i 17 se svrstavaju u skup za učenje, ostale u skup za testiranje
- U drugoj iteraciji slike s indeksima 2, 6, 10, 14 i 18 se svrstavaju u skup za učenje, ostale u skup za testiranje
- Analogno vrijedi i za treću i četvrtu iteraciju

Ovakvom raspodjelom svaka slika će se točno jednom naći u skupu za učenje.

### 10.1.2 Testiranje inačice I metode *PCA* – eksperiment 2

U ovom eksperimentu metoda *PCA* se provodila zasebno na svakom skupu lica ili otisaka dlanova koji pripadaju jednoj osobi. Sustav je testiran u dvadeset iteracija, gdje se

u svakoj iteraciji koristilo devetnaest slika za učenje, a po jedna za testiranje. U svakoj iteraciji se druga slika koristi za testiranje. U ovom slučaju eksperiment se provodi nad specijaliziranom skupu od samo devetnaest slika. Ovaj eksperiment je proveden za unimodalne i multimodalne sustave.

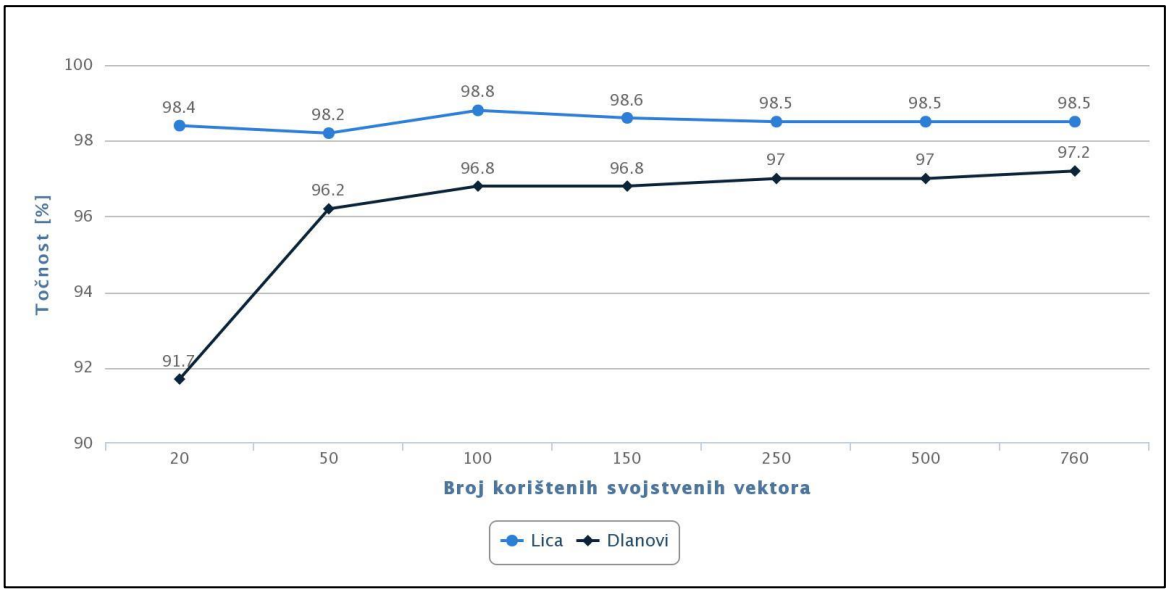
### 10.1.3 Testiranje inačice II metode *PCA* – eksperiment 3

Ovaj eksperiment je vrlo sličan prethodnome. Jedina razlika jest broj slika koji se koristi za učenje, kao što je opisano u poglavlju 7.2.2. Cilj umjetnog proširivanja skupa uzoraka za učenje na opisani način je povećanje broja svojstvenih vektora, odnosno dimenzionalnosti potprostora u koji će se uzorci projicirati jer je pretpostavka da će u prostoru većih dimenzija uzorci različitih razreda biti više razmaknuti pa će i klasifikacija biti točnija. Ovaj eksperiment je proveden za unimodalne i multimodalne sustave.

## 10.2 Analiza rezultata

### 10.2.1 Rezultati testiranja klasične metode *PCA* – eksperiment 1

Ovaj eksperiment proveden je za više brojeva svojstvenih vektora (20, 50, 100, 150, 250, 500, 759). Maksimalan broj korištenih svojstvenih vektora je 152 × 5 - 1 = 759. Sa grafa 2 je vidljivo da se visoka točnost postiže već i za mali broj korištenih svojstvenih vektora.




***Graf 2*: Ovisnost točnosti raspoznavanja o broju korištenih svojstvenih vektora (eksperiment 1)**

### 10.2.2 Rezultati testiranja unimodalnog sustava - inačica I metode *PCA* (eksperiment u2)

U tablici 4 prikazani su rezultati eksperimenta 2 za unimodalne sustave. Vidljivo je da su rezultati lošiji od rezultata dobivenima sa klasičnom metodom *PCA* (eksperiment 1). Broj korištenih svojstvenih vektora u ovom eksperimentu je 18 (kao što je navedeno u poglavlju 7.1 – ako je broj uzoraka *N* manji od dimenzionalnosti *n,* tada postoji *N – 1* svojstvena vrijednost veća od 0, a samim time i *N – 1* relevantan svojstveni vektor).

***Tablica 4*: Rezultati *leave - one - out* metode za eksperiment 2**

| Biometrijska karakteristika | Točnost |
|---|---|
| **Lice** | 82.25 % |
| **Dlan** | 73.37 % |

Tablica 5 prikazuje prosječno trajanje identifikacije jednog nepoznatog uzorka za serijsku i paralelnu implementaciju.

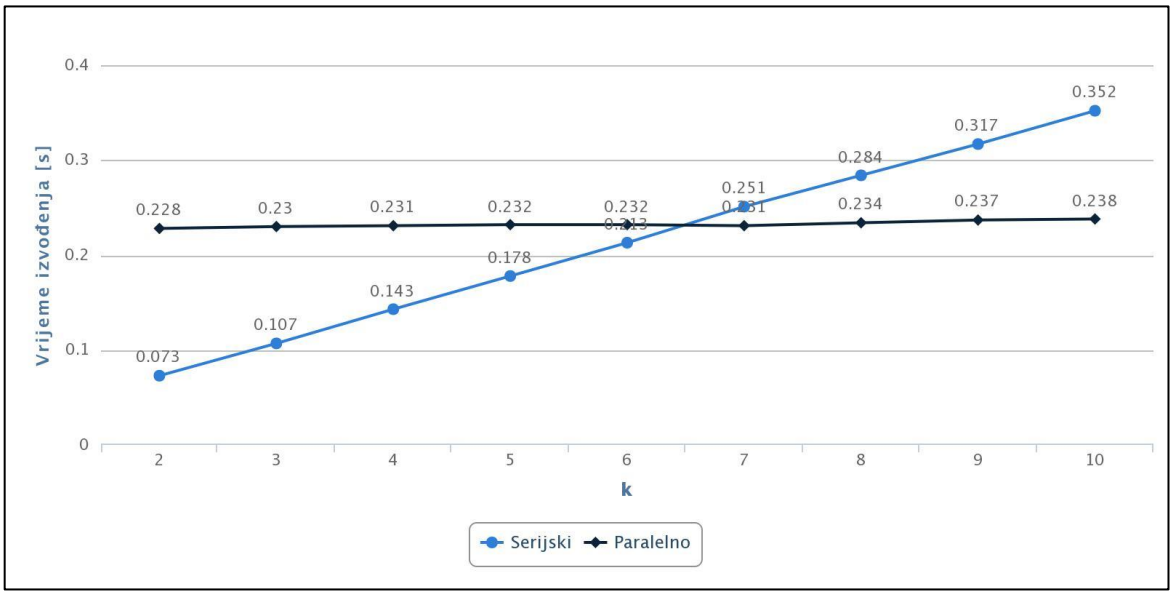
***Tablica 5*: Prosječna vremena trajanja identifikacije jednog nepoznatog uzorka**

| Serijska implementacija | Paralelna implementacija |
|---|---|
| 0.044 s | 0.228 s |

Prvotna pretpostavka je bila da će paralelna implementacija značajno ubrzati postupak identifkacije jer se nepoznati uzorak istovremeno projicira u svaki razred, no više je razloga zbog kojih rezultati nisu u skladu sa pretpostavkom. Zbog odabranog pristupa rješavanju problema (opisanog u poglavlju 9.1) istovremeno se izvršavaju samo 152 dretve (koliki je i broj razreda u koji se nepoznati uzorak projicira), što ne zauzima niti polovicu dostupnih jezgri grafičkog procesora (ukupno ih je 384) pa se čekanje na memorijske operacije ne može prekriti korisnim računanjem. 152 dretve je premalo da bi se paralelizacijom postiglo značajno ubrzanje i obično se preporuča program podesiti tako da se istovremeno izvodi na tisuće dretvi [7, 13], poput primjera paralelnog množenja matrica iz poglavlja 4.3.2.2. Problem čekanja na memorijske operacije još više dolazi do izražaja jer svi podaci leže u sporoj globalnoj memoriji jer brze memorije poput dijeljene ili pak registara nije moguće iskoristiti zbog njihovog ograničenog kapaciteta. Najbitniji razlog zbog ovakvih rezultata jesu male baze uzoraka, odnosno, malen broj različitih razreda, a što ujedno povlači i malen broj dretvi. Pretpostavka je da ukoliko bi se baza uzoraka povećala, povećao bi se i broj dretvi, grafički procesor bi bio bolje iskorišten (u smislu da se više vremena troši na računanje nego na memorijske operacije), a vrijeme izvršavanja ne bi raslo linearno kao što bi bio slučaj sa serijskom implementacijom. Zbog ove pretpostavke izvršen je još jedan eksperiment u kojem je broj razreda u koji se svaki uzorak projicira umjetno povećan *k* puta, i to na slijedeći način:

- Kod serijske implementacije projekcija i traženje minimalne udaljenosti nepoznatog uzorka obavlja se *k* puta. To je ekvivalentno povećanju broja razreda za *k* puta.
- Kod paralelne implementacije prilikom pokretanja kernela u konfiguraciji izvršavanja broj blokova dretvi koje će izvoditi kernel poveća se za *k* puta. To je također ekvivalentno povećanju broja razreda za *k* puta.

Graf 3 prikazuje vrijeme trajanja identifikacije jednog uzorka u slučaju da se početni broj razreda (152) poveća *k* puta. Vidljivo je da vrijeme izvođenja serijske implementacije linearno raste, što je u skladu s očekivanjima. Vrijeme izvođenja kod paralelne implementacije je gotovo konstatno, s tek neznatnim varijacijama. Zbog tehničkih ograničenja (maksimalni broj blokova i dretvi) i vrijeme izvođenja paralelne implementacije bi u nekom trenutku počelo rasti, no za dovoljno velik *k* paralelni algoritam bi još uvijek bio brži od serijskog. Za $k = 7$ paralelni program postaje brži od serijskog.




***Graf 3*: Vrijeme izvođenja u ovisnosti o faktoru povećanja broja razreda (eksperiment u2)**

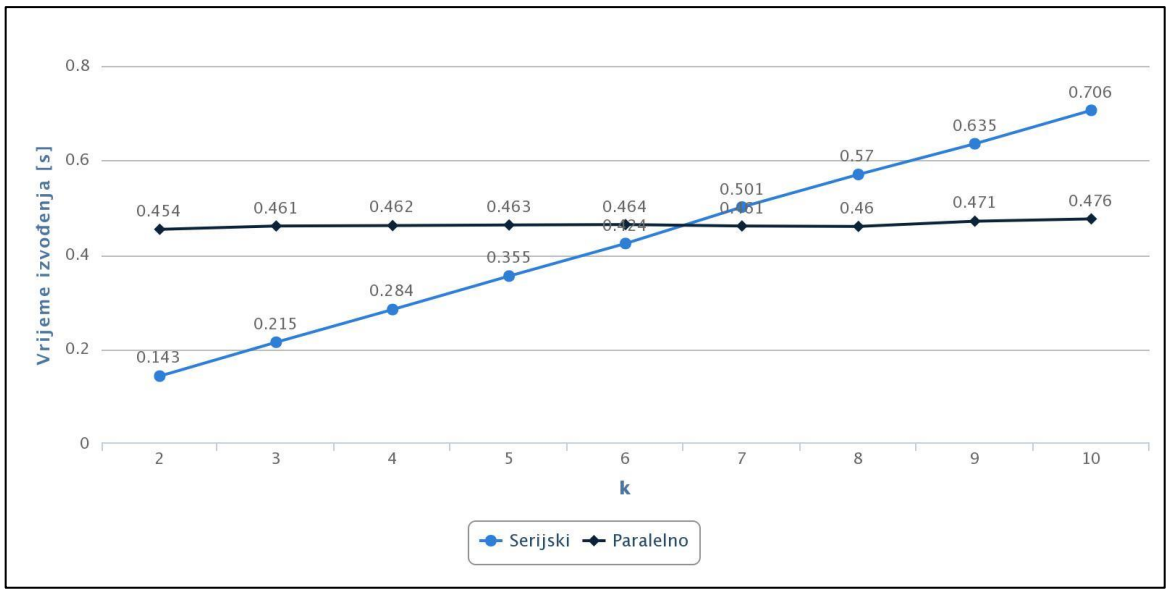
### 10.2.3 Rezultati testiranja mutlimodalnog sustava - inačica I metode *PCA* (eksperiment m2)

Multimodalni biometrijski sustav koji kombinira biometrijske karakteristika lica i dlana temeljen na inačici I metode *PCA* postiže prosječnu točnost od 94.7%, što je bolji rezultat od rezultata dobivenih prilikom tesitranja unimodalnih sustava temeljenih na istoj inačici metode *PCA*.

Kod serijske implementacije prvo se obrađuje uzorak koji predstavlja lice nepoznate osobe, a zatim uzorak dlana. Sukladno tome vrijeme potrebno za identifikaciju nepoznate osobe je u prosjeku dvostruko više od vremena potrebnog za identifikaciju jednog uzorka kod istovjetnog unimodalnog sustava (navedeno u tablici 5).

Kod paralelne implementacije pozivi odgovarajućim kernelima koji obrađuju uzorak lica, a zatim dlana su serijalizirani. Vrijeme za identifikaciju nepoznate osobe na temelju lica i dlana je također u prosjeku dvostruko veće od onoga navedenog u tablici 5 za paralelnu implementaciju.

Ukoliko bi se broj razreda umjetno povećavao kao u eksperimentu u2 dobili bi se slijedeći rezultati, prikazani grafom 4. Za $k = 7$ paralelni program postaje brži od serijskog, kao i kod unimodalnog sustava.




***Graf 4*: Vrijeme izvođenja u ovisnosti o faktoru povećanja broja razreda (eksperiment m2)**

### 10.2.4 Rezultati testiranja unimodlanog sustava - inačica II metode *PCA* (eksperiment u3)

Tablica 6 prikazuje rezultate eksperimenta u3 za različite brojeve korištenih svojstvenih vektora. Rezultati dobvieni u ovom eksperimentu su bolji od rezultata dobivenih u eksperimentu u2, što je u skladu s pretpostavkom da će umjetno proširivanje svakog razreda u svrhu povećanja broja svojstvenih vektora povećati točnost klasifikacije.

***Tablica 6*: Rezultati *leave - one - out* metode za eksperiment u3**

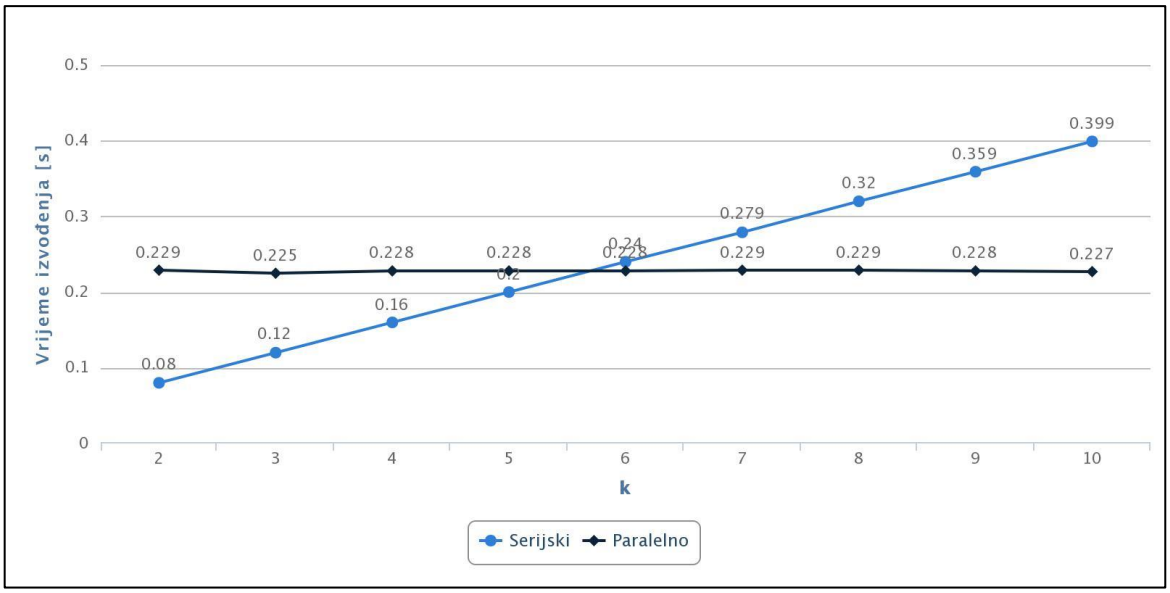
| | **Biometrijska karakteristika** | |
|---|---|---|
| **Broj svojstvenih vektora** | **Lice** | **Dlan** |
| 20 | 98.5 % | 99.87 % |
| 50 | 99.87 % | 99.97 % |
| 100 | 99.80 % | 99.97 % |
| 170 | 99.30 % | 99.80 % |

Tablica 7 prikazuje potrebno vrijeme za identifikaciju osobe za različit broj korištenih svojstvenih vektora za serijsku i paralelnu implementaciju. Kao i za eksperiment u2 vidljivo je da je serijska implementacija brža od paralelne ako se ispituje nad originalnom bazom od 152 razreda.

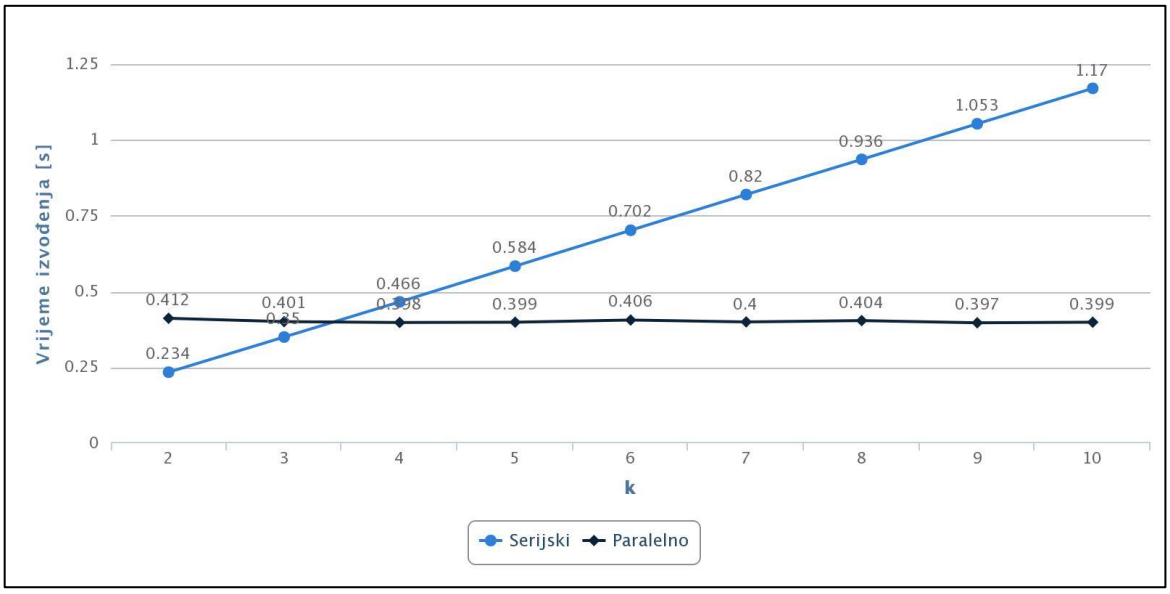
***Tablica 7*: Prosječna vremena trajanja identifikacije osobe na temelju kombinacije lica i dlana**

| | **Vrijeme** | |
|---|---|---|
| **Broj svojstvenih vektora** | **Serijska implementacija** | **Paralelna implementacija** |
| 20 | 0.041 s | 0.22 s |
| 50 | 0.110 s | 0.42 s |
| 100 | 0.191 s | 0.73 s |
| 170 | 0.432 s | 1.18 s |

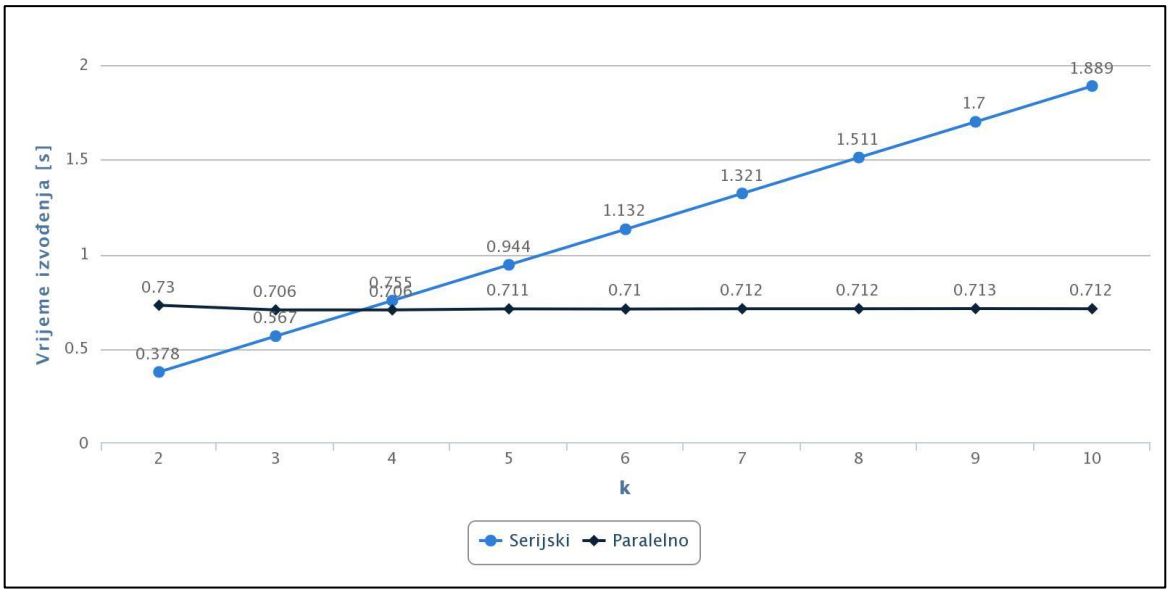
Ukoliko bi se broj razreda umjetno povećavao na način opisan u poglavlju 10.2.2, dobili bi se rezultati prikazani grafovima 5, 6, 7 i 8.



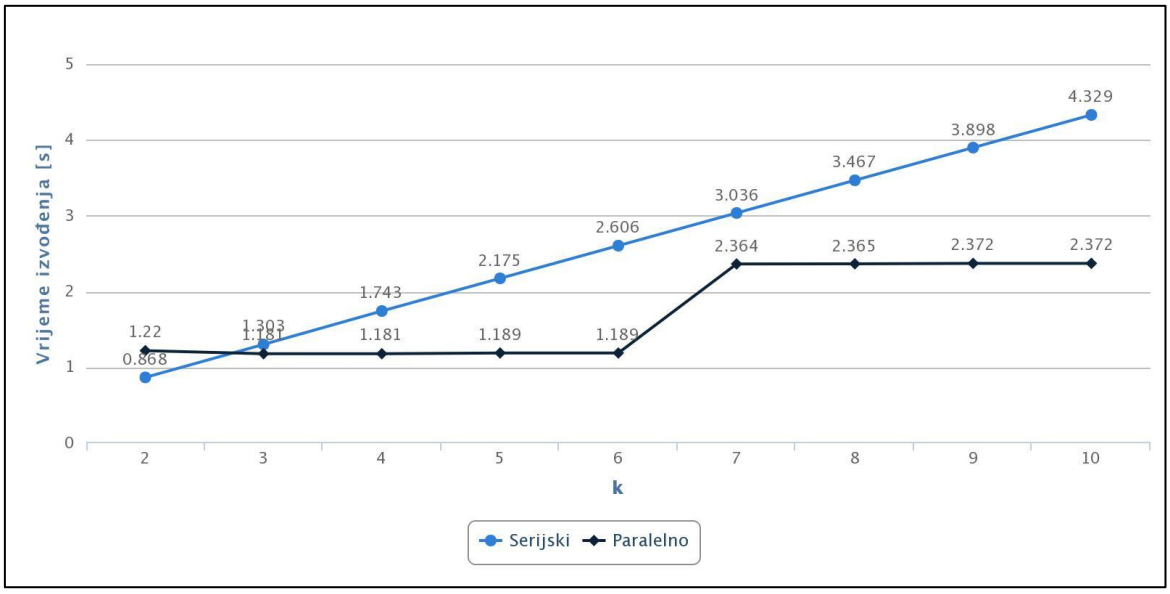

***Graf 5*: Vrijeme izvođenja u ovisnosti o faktoru povećanja broja razreda za eksperiment u3. Korišteno je 20 svojstvenih vektora.**




***Graf 6*: Vrijeme izvođenja u ovisnosti o faktoru povećanja broja razreda za eksperiment u3. Korišteno je 50 svojstvenih vektora.**




***Graf 7*: Vrijeme izvođenja u ovisnosti o faktoru povećanja broja razreda za eksperiment u3. Korišteno je 100 svojstvenih vektora.**




***Graf 8:* Vrijeme izvođenja u ovisnosti o faktoru povećanja broja razreda za eksperiment u3. Korišteno je 170 svojstvenih vektora.**

U skladu s očekivanjima za dovoljno velik *k* za svaki broj korištenih svojstvenih vektora paralelni program će biti brži od serijskog. Zanimljivo je promotriti graf 8 na kojem se vidi i pad u brzini paralelnog programa. Nakon pada brzine ona ostaje približno konstatna.

### 10.2.5 Rezultati testiranja multimodalnog sustava – inačica II metode *PCA* (eksperiment m3)

Tablica 8 prikazuje rezultate eksperimenta m3 za različite brojeve korištenih svojstvenih vektora. Ovakav sustav postiže maksimalnu točnost za sve testirane brojeve svojstvenih vektora. I u ovom slučaju je vidljivo da sustav koji se temelji na inačici II metode *PCA* postiže veću točnost od onoga temeljenog na inačici I (koji postiže točnost od 94.7 %).

***Tablica 8:*** **Rezultati *leave - one - out* metode za eksperiment m3**

| Broj svojstvenih vektora | Točnost |
|---|---|
| 20 | 100 % |
| 50 | 100 % |
| 100 | 100 % |
| 170 | 100 % |

Vremena izvođenja za eksperiment m3 prikazana su tablici 9. Iz tablice je vidljivo da su vremena izvođenja prosječno dvostruko veća u odnosu na eksperiment u3, kao što je i slučaj sa vremenima izvođenja eksperimenata m2 i u2.

***Tablica 9:*** **Prosječna vremena trajanja identifikacije osobe na temelju kombinacije lica i dlana**

| | Vrijeme | |
|---|---|---|
| **Broj svojstvenih vektora** | **Serijska implementacija** | **Paralelna implementacija** |
| 20 | 0.83 s | 0.43 s |
| 50 | 0.210 s | 0.82 s |
| 100 | 0.402 s | 1.41 s |
| 170 | 0.854 s | 2.32 s |

# 11 Zaključak

U ovom radu se pokušalo smanjiti vrijeme potrebno za identifikaciju nepoznate osobe uz pomoć paralelnih implementacija *PCA* temeljenih biometrijskih sustava koji za identifikaciju koriste značajke lica, dlana ili fuziju značajki lica i dlana na razini mjere podudaranja. Testirane su serijska implementacija klasične metode *PCA*, serijska i paralelna inačica metode *PCA* u kojoj se metoda *PCA* provodi nad svakim razredom posebno (za unimodalan i bimodalan sustav) te serijska i paralelna implementacija inačice metode *PCA* kod koje se svaki razred umjetno proširuje sa uzorcima iz drugih razreda (također su testirani unimodalan i bimodalan sustav).

Dobiveni rezultati pokazuju da se najbolji rezultati identifikacije dobivaju za inačicu metode *PCA* koja svaki razred proširuje sa uzorcima iz drugih razreda, i to za sve eksperimente. Najlošije rezultate daje inačica metode *PCA* kod koje se postupak provodi samo nad slikama koje pripadaju nekom razredu. Iz toga se može zaključiti da je veličina skupa za učenje bitnija od njegove specijaliziranosti. Dobiveni rezultati pokazuju da multimodalni sustavi koji kombiniraju značajke lica i dlana daju bolje rezultate od unimodalnih sustava koji se temelje samo na licu ili samo na dlanu, što je sukladno pretpostavci.

Paralelne inačice faza testiranja implementirane su u *CUDA* C jeziku i izvode se na grafičkom procesoru. Iz rezultata paralelizacije, odnosno dobivenih vremena izvođenja paralelnih inačica može se zaključiti da je opisane postupke isplativo paralelizirati na grafičkom procesoru, ali u slučaju da je broj razreda, a samim time i broj paralelnih dretvi, nekoliko puta veći od korištenog broja razreda. Zbog veličine korištenih baza serijska implementacija je u svakom eksperimentu bila brža od paralelne, no pokazlo se da ukoliko se broj razreda u bazi umjetno poveća za *k* puta da će vrijeme potrebno za identifikaciju nepoznate osobe kod serijske implementacije rasti linearno s faktorom *k,* a kod paralelne će ono biti približno konstantno do neke određene granice, odnosno dogoditi će se trenutak u kojem će paralelna implementacija postati brža od serijske.

# Literatura


[1] Štefić, D. Raspoznavanje slika lica i dlanova uporabom inačice metode analize glavnih komponenti. Diplomski rad. Fakultet elektrotehnike i računarstva, 2012.

[2] Vasiljević, I. Biometrija. Seminar. Fakultet elektrotehnike i računarstva, 2007.

[3] Biometrics, 20.3.2013., *Biometrics.* http://en.wikipedia.org/wiki/Biometrics

[4] Supercomputers, 6.4.2013., *Supercomputers* https://en.wikipedia.org/wiki/Supercomputer

[5] Top500, 6.4.2013., *Top500* http://www.top500.org/

[6] History of supercomputing, 6.4.2013., *History of supercomputing* https://en.wikipedia.org/wiki/History_of_supercomputing

[7] Cook, S. A Developer's Guide to Parallel Computing with GPUs: Supercomputing. Elsevier Inc, 2013

[8] Mahapharta, The Processor-Memory bottleneck: Problems and Solutions, 25.11.2008, *The Processor-Memory bottleneck: Problems and Solutions*, http://epic.hpi.uni-potsdam.de/pub/Home/TrendsAndConceptsII2010/HW_Trends_The_Processor-Memory_bottleneck___Problems_and_Solutions..pdf 11.4.2013

[9] Connection Machine, 12.4.2013, *Connection machine,* *http://en.wikipedia.org/wiki/Connection_Machine*

[10] Cray History, 12.4.2013, *Cray History, http://www.cray.com/About/History.aspx*

[11] Cell (microprocessor), 12.4.2013, *Cell (microprocessor)*, http://en.wikipedia.org/wiki/Cell_(microprocessor)

[12] Cell microprocessor, 12.4.2013, *Cell,* http://pc.watch.impress.co.jp/docs/2005/0208/kaigai153.htm

[13] Farber, R. CUDA Application Design and Development, Waltham, USA, Elsevier Inc, 2011

[14] Parallel Computing: Background, 4.5.2013, *Parallel Computing: Background,* http://www.intel.com/pressroom/kits/upcrc/parallelcomputing_backgrounder.pdf

[15] Budin, L. Operacijski sustavi. 1. Izdanje. Zagreb: Element, 2010.

[16] CUDA C Programming Guide, 15.3.2013. *CUDA C Pogramming Guide*, http://docs.nvidia.com/cuda/cuda-c-programming-guide/

[17] Fratrić, I. Biometrijska verifikacija osoba temeljena na značajkama dlana i lica dobivenim iz videosekvenci. Doktorska disertacija. Fakultet elektrotehnike i računarstva 2011.

[18] Ross, A. Multimodal Biometrics: An Overview. Proc. of 12th European Singal Processing Conference (EUSIPCO). Vienna, Austria, 2004. Str 1221 - 1224

[19] Jain, A. An introduction to Biometric Recognition, IEEE Transactions on Circuits and Systems for Video Technology; Special Issue on Image and Video Biometrics, Vol. 14, No. 1, January 2004

[20] Spacek, L. A Collection of Facial Images: Faces94, 16.2.2007, *A Collection of Facial Images: Faces94,* http://cswww.essex.ac.uk/mv/allfaces/faces94.html, 2012.

[21] The Hong Kong Polytechnic University (PolyU) Palmprint Databace, 2003, Biometrics Research Center, http://www4.comp.polyu.edu.hk/~biometrics/, 2012.

[22] Tuning CUDA Applications for Kepler, 15.3.2013. *Tuning CUDA Applications for Kepler* http://docs.nvidia.com/cuda/kepler-tuning-guide/index.html

# Programska implementacija PCA temeljenog biometrijskog sustava u CUDA okruženju

## Sažetak

U ovom se radu provela usporedba serijske i paralelne implementacije biometrijskih sustava temeljenih na inačicama metode *PCA*. *PCA* metoda vrši transformaciju uzoraka u prostor dimenzija manjih od dimenzija originalnog prostora tako da je srednja kvadratna pogreška između originalnog uzorka i transformiranog uzorka jednaka sumi varijanci komponenti koje su eliminirane iz originalnog uzorka. Metoda *PCA* se obično provodi nad cijelim skupom uzoraka za učenje te se kao rezultat dobiva jedan srednji uzorak i skup ortonormalnih vektora koji čine bazu potprostora u koji će se uzorci projicirati. Nepoznati uzorak se nakon transformacije klasificira *1 – NN* algoritmom. Osim originlane metode testirane su i dvije inačice. U prvom slučaju se metoda *PCA* provodi nad svakim razredom posebno, a u drugom slučaju se svaki razred proširi sa slikama iz drugih razreda te se *PCA* provodi nad svakim razredom zasebno kao i u prvom slučaju. Sustavi bazirani na prvoj inačici postižu lošije rezultate od sustava baziranih na originalnoj metodi dok sustavi bazirani na drugoj inačici postižu bolje rezultate. Inačice metode *PCA* testiraju se na bazama lica i otisaka dlanova, a zatim i na himeričkoj bazi koja se temelji na fuziji lica i dlana. Faza identifikacije nepoznate osobe implementirana je i serijski i paralelno na grafičkom procesoru u CUDA C jeziku.

**Ključne riječi:** Serijska implementacija, paralelna implementacija, biometrijski sustav, PCA, 1 – NN, raspoznavanje lica, raspoznavanje otisaka dlanova, fuzija, CUDA

# CUDA implementation of a PCA based biometric system

## Abstract

Within this thesis serial and parallel implementations of a biometric systems based on two variants of a *PCA* method are compared. *PCA* method transforms high dimensional samples into the subspace of lower dimensions in such way that mean squared error between original sample and transformed sample is equal to the sum of variances of components that are eliminated from original sample. *PCA* method is usually performed on whole set of train samples. Results are mean sample and set of orthonormal vectors which form a basis of a subspace in which train samples will be projected. After transformation a live template is classified using a *1 – NN* algorithm. Beside original method, two more variants of a *PCA* method were tested. In first case *PCA* method is performed on each class of samples separately. In second case each class of samples is extended with images from another classes before *PCA* method is performed on each class separately, like in first case. Systems based on the first version of a *PCA* method gives poor results in comparison with the original method, while systems based on second version outperformes systems based on original method. Implemented systems are tested with a face images, palmprint images and finally, with a fusion of face and palmprint images. Identification phase of an unknown person is implemented both serial and parallel. Parallel version is implemented in CUDA C language and is intended to run on a CUDA capable graphics processor.

**Keywords:** Serial implementation, parallel implementation, biometric system, PCA, 1 – NN, face recognition, palmprint recognition, fusion, CUDA

# Dodatak A: Programska implementacija

Za programsku implementaciju korišteno je računalo sa *Intel i7* procesorom (2.3 GHz) i 4GB RAM. Grafička kartica je Nvidia GeForce GTX 600M. Grafički procesor ukupno ima 384 *CUDA* jezgri, odnosno dva *stream* multiprocesora sa po 192 *stream* procesora od kojih svaki radi na taktu od 835 MHz.

Za operacije nad slikama (učitavanje, izrezivanje područja interesa, skaliranje, pretvorba u vektor), operacije nad matricama i računanje svojstvenih vrijednosti i vektora korištena je *OpenCV* biblioteka (v2.4.9.).

Kompletna programska implementacija načinjena je na *Windows 7* operativnom sustavu unutar *Visual Studio 2010* razvojnog okruženja. Za pokretanje svakog od programa potrebno je imati instaliran *CUDA Toolkit v5.0* kao i *OpenCV* biblioteku sa uključenom podrškom za *CUDA* grafičke procesore.

Kompletna implementacija nalazi se na CD – u priloženom uz rad. Unutar *Visual Studio soulutiona* se nalaze slijedeći projekti:

- Eksperiment 1 – serijska implementacija standardne metode *PCA*
- Eksperiment 2 – serijska implementacija inačice I metode *PCA*
- Eksperiment 3 – serijska implementacija inačice II metode *PCA*
- Fuzija 2 – serijska implementacija multimodalnog biometrijskog sustava temeljenog na inačici I metode *PCA*
- Fuzija 3 - serijska implementacija multimodalnog biometrijskog sustava temeljenog na inačici II metode *PCA*
- GPU - Eksperiment 2 – paralelna implementacija inačice I metode *PCA*
- GPU - Eksperiment 3 – paralelna implementacija inačice II metode *PCA*
- GPU - Fuzija 2 – paralelna implementacija multimodalnog biometrijskog sustava temeljenog na inačici I metode *PCA*
- GPU - Fuzija 3 - paralelna implementacija multimodalnog biometrijskog sustava temeljenog na inačici II metode *PCA*