SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

DIPLOMSKI RAD br. 539

Razvoj računalno zahtjevnih algoritama u hibridnoj paralelnoj okolini

Ivo Majić

Zagreb, lipanj 2013.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA
ODBOR ZA DIPLOMSKI RAD PROFILA

Zagreb, 7. ožujka 2013.

DIPLOMSKI ZADATAK br. 539

Pristupnik: **Ivo Majić**
Studij: Računarstvo
Profil: Računarska znanost

Zadatak: **Razvoj računalno zahtjevnih algoritama u hibridnoj paralelnoj okolini**

Opis zadatka:

Opisati posebnosti razvoja algoritama u hibridnoj paralelnoj okolini uz pomoć standarda OpenCL. Odrediti svojstva algoritama koja daju mogućnost paralelizacije u hibridnoj okolini. Ispitati učinkovitost paralelnih inačica kriptografskih funkcija i problema optimizacije kombinatoričkih funkcija. Ostvariti paralelni hibridni evolucijski algoritam za rješavanje problema kontinuirane optimizacije. Ispitati mogućnosti ugradnje paralelnog podsustava u postojeći okvir za evolucijsko računanje. Radu priložiti algoritme, izvorne tekstove programa i rezultate uz potrebna objašnjenja i dokumentaciju. Citirati korištenu literaturu i navesti dobivenu pomoć.

Zadatak uručen pristupniku: 15. ožujka 2013.
Rok za predaju rada: 28. lipnja 2013.

Mentor:

Prof.dr.sc. Domagoj Jakobović

Djelovođa:

Doc.dr.sc. Tomislav Hrkać

Predsjednik odbora za diplomski rad profila:

Prof.dr.sc. Siniša Srbljić

SADRŽAJ

Popis tablica **vi**

Popis slika **vii**

Popis algoritama i tekstova programa **ix**

1 Uvod **1**

2 Arhitektura grafičkih kartica **2**

2.1 Sklopovski model 2

2.2 Memorijski model 3

2.3 SIMT model izvršavanja 5

3 Open Computing Language **7**

3.1 OpenCL arhitektura 7

3.1.1 Platformski model 8

3.1.2 Programski model 8

3.1.3 Memorijski model 9

3.1.4 Izvršni model 10

3.2 Programski jezik OpenCL C 14

3.3 Java OpenCL 16

4 Simetrični kriptografski algoritmi **17**

4.1 Načini kriptiranja 18

4.2 Data Encryption Standard 20

4.2.1 Opis implementacije 21

4.2.2 Rezultati 22

4.3 Advanced Encryption Standard 26

4.3.1 Opis implementacije 29

4.3.2 Rezultati 30

5 Evolucijski algoritmi **33**
5.1 Genetski algoritam 34
5.2 Optimizacija kombinatoričkih funkcija 37
5.2.1 Opis implementacije 38
5.2.2 Rezultati 39
5.3 Hibridni Hooke-Jeeves genetski algoritam 42
5.3.1 Opis implementacije 43
5.3.2 Rezultati 44
5.4 Hibridni Taguchi genetski algoritam 48
5.4.1 Opis 1D implementacije 51
5.4.2 Opis 2D implementacije 52
5.4.3 Rezultati 54
5.5 Evolutionary Computation Framework 60
5.5.1 Rezultati 62

6 Zaključak **63**

Literatura **64**

A OpenCL kernel - Hooke-Jeeves istraži **68**

B OpenCL kernel - Taguchi križanje - 1 radna jedinica / 1 potomak **70**

C OpenCL kernel - Taguchi križanje - 32 radne jedinice / 1 potomak **73**

POPIS TABLICA

3.1 Razine pristupa OpenCL memorijskog modela 9
3.2 OpenCL C memorijski kvalifikatori 15
3.3 OpenCL C specifične naredbe 15

4.1 Testno računalo 22
4.2 NVIDIA GeForce 9600GT 22
4.3 Ovisnost broja krugova o veličini ključa (AES) 26

5.1 Kombinatoričke funkcije jedne varijable 37
5.2 Indeksi svojstava 38
5.3 Komponente - FIT1 41
5.4 Komponente - FIT3 41
5.5 Ispitne funkcije (COCO) 44
5.6 Hooke-Jeeves istraži komponente - sferna funkcija 47
5.7 Hooke-Jeeves istraži komponente - Rastrigin funkcija 47
5.8 Dvodimenzionalno ortogonalno polje $L_8(2^7)$ 49
5.9 Primjer: Roditelji $R1$ i $R2$ odabrani selekcijom 50
5.10 Primjer: Eksperimenti i pripadajuće vrijednosti SNR-a 50
5.11 Primjer: Vrijednosti efekata po razinama i optimalni potomak 51
5.12 Taguchi 1D komponente - sferna funkcija 56
5.13 Taguchi 1D komponente - Rastrigin funkcija 56
5.14 Taguchi 2D komponente - sferna funkcija 59
5.15 Taguchi 2D komponente - Rastrigin funkcija 59

POPIS SLIKA

2.1 Model protočnog multiprocesora 3
2.2 Memorijski model grafičkih kartica 4

3.1 OpenCL memorijski model 10
3.2 Primjer dvodimenzionalnog indeksnog prostora 13

4.1 ECB način kriptiranja (enkripcija i dekripcija) 18
4.2 CBC način kriptiranja (enkripcija i dekripcija) 18
4.3 CTR način kriptiranja (enkripcija i dekripcija) 19
4.4 CFB način kriptiranja (enkripcija i dekripcija) 19
4.5 DES algoritam 21
4.6 Propusnost, DES dekriptiranje, ECB način 23
4.7 Prosječno ubrzanje, DES dekriptiranje, ECB način 23
4.8 Propusnost, DES dekriptiranje, CTR način 24
4.9 Prosječno ubrzanje, DES dekriptiranje, CTR način 24
4.10 Propusnost, DES dekriptiranje, CBC način 24
4.11 Prosječno ubrzanje, DES dekriptiranje, CBC način 25
4.12 Trajanje dekriptiranja po komponentama, CTR način (DES) 25
4.13 128 bitni AES blok 26
4.14 Posmakni redove (AES) 28
4.15 Pomiješaj stupce (AES) 28
4.16 Propusnost, AES dekriptiranje, 256 bitni ključ, ECB način 30
4.17 Prosječno ubrzanje, AES dekriptiranje, 256 bitni ključ, ECB način 30
4.18 Propusnost, AES dekriptiranje, 256 bitni ključ, CBC način 31
4.19 Prosječno ubrzanje, AES dekriptiranje, 256 bitni ključ, CBC način 31
4.20 Propusnost, AES dekriptiranje, 256 bitni ključ, CTR način 31
4.21 Prosječno ubrzanje, AES dekriptiranje, 256 bitni ključ, CTR način 32
4.22 Trajanje dekriptiranja po komponentama, CTR način (AES) 32

5.1 Propusnost, evaluacija kombinatoričkih funkcija, FIT1 39
5.2 Prosječno ubrzanje, evaluacija kombinatoričkih funkcija, FIT1 39
5.3 Propusnost, evaluacija kombinatoričkih funkcija, FIT2 40
5.4 Prosječno ubrzanje, evaluacija kombinatoričkih funkcija, FIT2 40
5.5 Propusnost, evaluacija kombinatoričkih funkcija, FIT3 40
5.6 Prosječno ubrzanje, evaluacija kombinatoričkih funkcija, FIT3 41
5.7 Propusnost, Hooke-Jeeves istraži, sferna funkcija, 30 varijabli 45
5.8 Prosječno ubrzanje, Hooke-Jeeves istraži, sferna funkcija, 30 varijabli . . . 45
5.9 Propusnost, Hooke-Jeeves istraži, Rastrigin funkcija, 30 varijabli 46
5.10 Prosječno ubrzanje, Hooke-Jeeves istraži, Rastrigin funkcija, 30 varijabli . . 46
5.11 Propusnost, Hooke-Jeeves istraži, Weierstrass funkcija, 30 varijabli 46
5.12 Prosječno ubrzanje, Hooke-Jeeves istraži, Weierstrass funkcija, 30 varijabli 47
5.13 Hibridni Taguchi genetski algoritam (HTGA) 48
5.14 Jednodimenzionalna OpenCL inačica HTGA 52
5.15 Dvodimenzionalna OpenCL inačica HTGA 52
5.16 Propusnost, Taguchi križanje 1D, sferna funkcija, 30 varijabli 54
5.17 Prosječno ubrzanje, Taguchi križanje 1D, sferna funkcija, 30 varijabli . . . 54
5.18 Propusnost, Taguchi križanje 1D, Rastrigin funkcija, 30 varijabli 55
5.19 Prosječno ubrzanje, Taguchi križanje 1D, Rastrigin funkcija, 30 varijabli . . 55
5.20 Propusnost, Taguchi križanje 1D, Weierstrass funkcija, 30 varijabli 55
5.21 Prosječno ubrzanje, Taguchi križanje 1D, Weierstrass funkcija, 30 varijabli . 56
5.22 Propusnost, Taguchi križanje 2D, sferna funkcija, 30 varijabli 57
5.23 Prosječno ubrzanje, Taguchi križanje 2D, sferna funkcija, 30 varijabli . . . 57
5.24 Propusnost, Taguchi križanje 2D, Rastrigin funkcija, 30 varijabli 58
5.25 Prosječno ubrzanje, Taguchi križanje 2D, Rastrigin funkcija, 30 varijabli . . 58
5.26 Propusnost, Taguchi križanje 2D, Weierstrass funkcija, 30 varijabli 58
5.27 Prosječno ubrzanje, Taguchi križanje 2D, Weierstrass funkcija, 30 varijabli . 59
5.28 UML dijagram nadogradnje ECF razvojnog okruženja 60
5.29 Rezultati trajanja evaluacije (ECF) 62

POPIS ALGORITAMA I TEKSTOVA PROGRAMA

3.1 Tekst programa: Funkcija zbrajanja elemenata 2 cjelobrojna polja 14
3.2 Tekst programa: OpenCL jezgrena funkcija zbrajanja elemenata dvaju cjelobrojnih polja 14
4.1 Tekst programa: Prototip jezgrene funkcije kriptiranja algoritma DES . . . 21
4.2 Algoritam: Jezgrena funkcija kriptiranja algoritma DES (ECB) 22
4.3 Tekst programa: Prototip jezgrene funkcije kriptiranja algoritma AES . . . 29
4.4 Algoritam: Jezgrena funkcija kriptiranja algoritma AES (ECB) 29
5.1 Algoritam: Generacijski genetski algoritam 35
5.2 Tekst programa: Prototip jezgrene funk. evaluacije kombinatoričkih funkcija 38
5.3 Algoritam: Hibridni Hooke-Jeeves genetski algoritam, HHJGA 42
5.4 Algoritam: Funkcija `istrazi` Hooke-Jeeves algoritma 43
5.5 Tekst programa: Prototip jezgrene funkcije Hooke-Jeeves istraži 44
5.6 Tekst programa: Prototip jezgrene funkcije Taguchi križanja 51
5.7 Tekst programa: Prototip ECF jezgrene funkcije evaluacije 61

1. Uvod

Tijekom posljednjih nekoliko godina, mnoga istraživanja bave se tematikom učinkovite prilagodbe i implementacije postojećih algoritama za izvođenje na grafičkim karticama (engl. *GPU - Graphics Processing Unit*). Arhitektura grafičke kartice dizajnirana je s ciljem efikasnog iskorištavanja paralelizma na razini podataka, koji su do prije nekoliko godina bili pretežito grafičke prirode. Veliki utjecaj na ubrzani rast ovog područja istraživanja imao je i razvoj raznih radnih okruženja od strane proizvođača grafičkih kartica (NVIDIA, AMD), ali i nezavisnih organizacija (Khronos Group). Razvoj takvih okruženja omogućio je da grafičke kartice počnu izvršavati zadatke koji su dotad bili namijenjeni isključivo za procesore opće namjene (engl. *CPU - Central Processing Unit*).

Grafičke kartice srednjeg cjenovnog razreda u prosjeku imaju između 64 i 128 procesnih elemenata, dok ih one višeg cjenovnog razreda i specijalizirane namjene (poput NVIDIA Tesla serije) imaju i po nekoliko stotina. U algoritme koji su u ovom radu ispitani spadaju danas često korišteni kriptografski algoritmi, te algoritmi koji su bazirani na evolucijskom računanju. Cilj ovog rada je pokazati da se čak i sa standardnom grafičkom karticom mogu postići osjetna ubrzanja u radu ovih algoritama.

U prvom poglavlju dan je opis arhitekture grafičke kartice te način na koji je u njoj izveden podatkovni paralelizam. Zatim je u idućem poglavlju opisano radno okruženje OpenCL (engl. *Open Computing Language*), koji je korišten prilikom implementacije algoritama. U trećem poglavlju opisan je rad simetričnih kriptografskih algoritama te su dani rezultati njihovog izvođenja na grafičkoj kartici. U posljednjem poglavlju opisano je nekoliko algoritama koji se temelje na evolucijskom računanju te dobiveni rezultati ubrzanja njihovog izvođenja.

2. Arhitektura grafičkih kartica

Grafički procesori (engl. *Graphics Processor Unit*) su specijalizirana vrsta računalnih procesora koji su izvorno razvijeni za efikasnu obradu 3D podataka korištenih u računalnoj grafici. Osnovna razlika između procesora opće namjene ili CPU (engl. *Central Processing Unit*) te grafičkog procesora ili GPU (engl. *Graphics Processing Unit*) je u tome da je CPU dizajniran da što brže izvrši skup instrukcija, dok je GPU sposoban isti zadani skup instrukcija izvršiti paralelno nad više podataka.

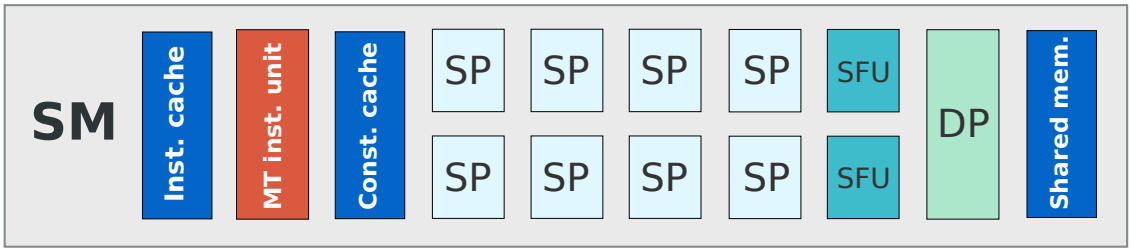
Jednu od najranijih klasifikacija računalnih sustava prema načinu obrade podataka osmislio je Michael J. Flynn [9]. Prema Flynnovoj klasifikaciji von Neumanov model računala spada u SISD (engl. *Single Instruction Single Data*) arhitekturu, dok grafička kartica spada u SIMD (engl. *Single Instruction Multiple data*) arhitekturu. U nastavku je detaljnije opisan sklopovski i memorijski model modernih grafičkih kartica.

2.1. Sklopovski model

Grafička kartica se u pravilu sastoji od nekoliko protočnih multiprocesora (engl. *SM - Stream Multiprocessors*). Multiprocesori su međusobno neovisni te su modelirani prema SIMD (engl. *Single Instruction Multiple Data*) arhitekturi. Svaki pojedini multiprocesor se sastoji od:

- 8 protočnih procesora (engl. *SP - Steam Processors*)
- 2 jedinice za specijalne funkcije (sin, cos, ...) (engl. *SFU - Special Functions Unit*)
- instrukcijska priručna memorija (engl. *instruction cache*)
- konstantna priručna memorija (engl. *constant cache*)
- višedretvena instrukcijska jedinica (engl. *Multithreaded Instruction Unit*)
- jedinica za operacije brojevima s pomičnim zarezom (engl. *Double Precision Unit*)
- dijeljena memorija (engl. *shared memory*)

Svaki od 8 protočnih procesora unutar jednog multiprocesora tijekom jednog takta izvršava istu instrukciju ali nad različitim podacima [17]. Pojednostavljeni prikaz protočnog multiprocesora prikazan je na slici 2.1



Slika 2.1: Model protočnog multiprocesora

U literaturi se protočni multiprocesori često nazivaju i računskim jedinicama (engl. *compute unit*), dok se protočni procesori sadržani u sklopu njih zovu procesnim elementima (engl. *processing element*). Proizvođači grafičkih kartica poput tvrtke NVIDIA imaju pak zasebno nazivlje, te se protočni procesori nazivaju CUDA jezgrama (engl. *CUDA cores*).

2.2. Memorijski model

Kod modernih grafičkih kartica razlikujemo nekoliko vrsta memorije koje se uglavnom razlikuju po brzini pristupa te količini dostupnog podatkovnog prostora.

Globalna memorija (engl. *global memory*) se odnosi na radnu memoriju grafičke kartice, koja se skraćeno zove VRAM (engl. *Video Random Access Memory*). Veličina VRAM memorije može iznositi od nekoliko stotina megabajta do nekoliko gigabajta. Radna memorija se najčešće koristi za spremanje podataka koji čekaju obradu na grafičkoj kartici, te spremanje rezultata obrade. VRAM je najčešće implementiran izvan grafičkog procesora (engl. *off-chip*) te iz tog razloga ujedno ima i najsporiju brzinu pristupa. Pristup globalnoj memoriji imaju svi protočni multiprocesori.

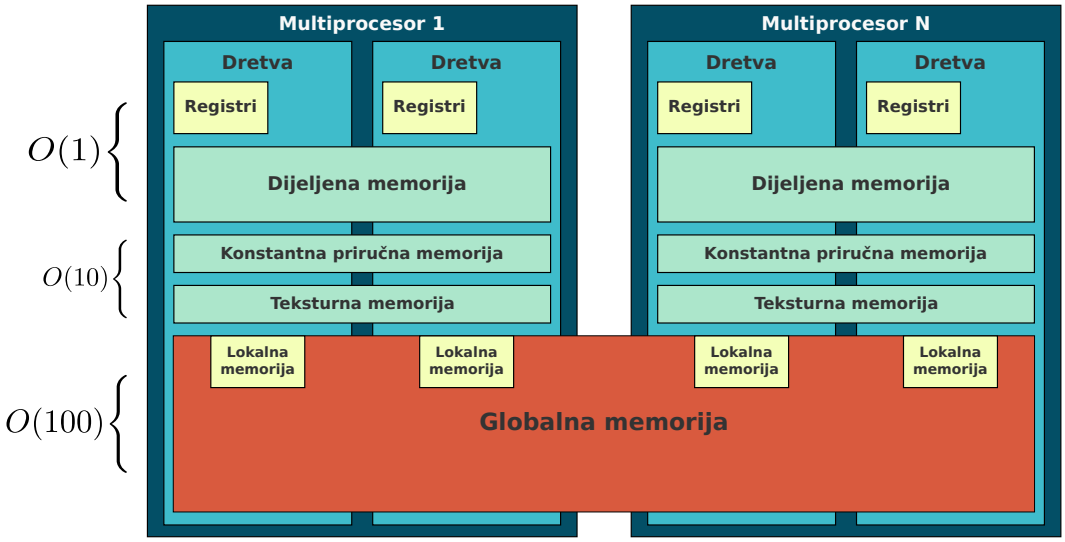
Dijeljena memorija (engl. *shared memory*) je vlastita memorija koju sadrži svaki protočni multiprocesor (računska jedinica), te joj zajednički mogu pristupati svi protočni procesori (procesni elementi) unutar istog multiprocesora. Takva memorija služi kao brza memorija kojom protočni procesori unutar istog multiprocesora mogu dijeliti podatke puno brže nego koristeći globalnu memoriju. Pristup dijeljenoj memoriji može biti i do 100x brži nego pristup globalnoj memoriji.

Privatna memorija je vlastita memorija koju sadrži svaki pojedini protočni procesor (procesni element) unutar jednog multiprocesora. Protočni procesori ne mogu pristupati privatnoj memoriji drugih protočnih procesora, već samo vlastitoj privatnoj memoriji. Prilikom programiranja u nekom od radnih okruženja, privatna memorija se koristi kao stog za lokalne varijable. Pod pojmom privatna memorija se misli na primarni skup 32-bitnih registara koji se nalaze u svakom protočnom procesoru, ali i dio globalne memorije (lokalna memorija) koji se koristi u slučaju popunjenosti registara.

Konstantna memorija (engl. *constant cache*) je jedna od bržih vrsta memorije u grafičkoj kartici, ali ujedno ima i najveća ograničenja. Konstantna memorija je memorija namijenjena samo čitanju (engl. *read-only*), te se podaci na nju moraju pretpohraniti tako da ih se deklarira konstantnima. Konstanta memorija je posebno efikasna ako bar pola dretvi osnovne skupine (engl. *half warp*) u istom trenutku žele pristupiti podatku na istoj memorijskoj lokaciji, njezina brzina je tada usporediva s brzinom pristupa registrima tzv. operacija prijenosa (engl. *broadcast operation*). Veličina konstantne memorije je u pravilu ograničena na 64KB.

Teksturna memorija (engl. *texture cache*) je posebna vrsta memorije koja omogućava brz pristup podacima namijenjenim korištenju u računalnoj grafici, poput 2D i 3D objekata. Slično kao i kod konstantne memorije, i kod teksturne memorije je moguće samo čitanje podataka (engl. *read-only*).

Na slici 2.2 prikazana je memorijska struktura grafičkih kartica te međusobni položaj različitih vrsta memorije.



Slika 2.2: Memorijski model grafičkih kartica

2.3. SIMT model izvršavanja

SIMT (engl. *Single Instruction Multiple Threads*) model izvršavanja sličan je SIMD (engl. *Single Instruction Multiple Data*) modelu gdje jednu instrukciju izvršava više procesnih elemenata. Pojam je prvi put uvela NVIDIA, kako bi dodatno definirala nekoliko specifičnosti grafičkih kartica. SIMT model dodatno definira način izvršavanja i ponašanje dretvi prilikom uvjetnog grananja i pristupa istim memorijskim lokacijama. Iako se prilikom programiranja mogu zanemariti SIMT pravila, njihovo poštivanje često može dovesti do značajnog povećanja performansi.

Prilikom izvršavanja programskih instrukcija protočni multiprocesor stvara, raspoređuje i izvršava dretve (engl. *threads*) u skupinama od 32 paralelne dretve koje se zovu **osnovne skupine** (engl. *warps*). Pojedinačne dretve koje čine jednu osnovnu skupinu započinju izvršavati iste instrukcije, ali svaka ima vlastito programsko brojilo i registre stanja. To im omogućuje uvjetno grananje i neovisno izvršavanje programskih instrukcija. Prilikom pokretanja više dretvi, multiprocesor ih prvo rasporedi u više osnovnih skupina, te raspoređivač (engl. *warp scheduler*) zatim svaku od skupina šalje na izvršavanje. Dretve su unutar osnovne skupine raspoređene slijedno, u rastućem poretku prema jedinstvenom identifikatoru dretve [10].

Osnovna skupina svaki takt izvršava jednu strojnu instrukciju, pa je najveća efikasnost postignuta kada sve 32 dretve unutar skupine izvršavaju istu instrukciju. Ako neka od dretvi uđe u uvjetno grananje, raspoređivač će slijedno izvršiti svaku granu, gaseći pritom dretve koje nisu ušle u grananje. Nakon što su sve grane izvršene, dretve unutar skupine ponovo nastavljaju paralelno izvršavanje. Ako se želi postići što veću efikasnost, treba minimizirati uvjetno grananje dretvi unutar iste osnovne skupine. Kada dretve unutar iste osnovne skupine žele pisati na istu memorijsku lokaciju u globalnoj ili lokalnoj memoriji, broj slijednih operacija pisanja na tu lokaciju određen je računskom sposobnošću (engl. *CC - Computing Capabilitiy*) grafičke kartice. Nije moguće odrediti koja će dretva u skupini posljednja pisati na zajedničku memorijsku lokaciju [10] [5].

U odnosu na procesore opće namjene koji imaju pristup brzoj priručnoj memoriji (engl. *L2 cache*), grafički procesori brzinu izvođenja postižu pametnim skrivanjem pristupa memoriji. Primjerice, ako prilikom izvođenja neka osnovna skupina zahtjeva pristup globalnoj memoriji, raspoređivač će na izvođenje poslati drugu dostupnu skupinu. Pristup globalnoj memoriji će se zatim obaviti paralelno u pozadini. Iz ovog razloga grafička kartica mora imati dovoljno posla (dretvi), kako bi raspoređivač imao što veći broj osnovnih skupina na izbor [5].

Postupak izmijene trenutno aktivne osnovne skupine (engl. *context switch*) nema vremensku cijenu jer je kontekst svake pojedine osnovne skupine pohranjen na multiprocesoru. Ovo ima za posljedicu da je broj dretvi koje je moguće pokrenuti na multiprocesoru ovisan o količini registara i dijeljene memorije koju koristi jezgrena funkcija, te količini dostupnih registara i dijeljene memorije multiprocesora. Ovo također ima za posljedicu da broj dretvi koje želimo pokrenuti na grafičkoj kartici mora biti unaprijed poznat zbog izračuna i pripreme potrebnih resursa za kontekste.

3. Open Computing Language

OpenCL (engl. *Open Computing Language*) je radno okruženje pogodno za programiranje u hibridnim paralelnom okolinama. Radno okruženje se sastoji od jezika OpenCL C, programskog prevodioca (engl. *compiler*) te izvršne okoline (engl. *runtime environment*) potrebne za pokretanje programa.

Razvijeno radno okruženje omogućava efikasno programiranje CPU + GPU heterogenih sustava, gdje CPU kontrolira izvođenje, dok GPU provodi zadane računske operacije (engl. *GPGPU, General Purpose GPU*). Međutim, OpenCL nije ograničen samo na heterogene sustave te se može koristiti i u višejezgrenim homogenim sustavima (poput nove Intel Core serije procesora opće namjene). U takvim sustavima jedna jezgra kontrolira izvođenje, dok ostale provode zadane računske operacije [6].

OpenCL specifikacija razvijena je od strane Khronos grupe, koja je poznata po održavanju OpenGL specifikacije. Grupa se sastoji od predstavnika više multinacionalnih kompanija (AMD, NVIDIA, Intel, IBM, Texas Instruments, ... itd.) čije se poslovanje temelji na proizvodnji višejezgrenih procesora i/ili programske podrške koja koristi višejezgrenu funkcionalnost. Krajnji cilj specifikacije je mogućnost programiranje bilo koje kombinacije procesora, koristeći jedan jedinstveni standardizirani jezik [18].

3.1. OpenCL arhitektura

OpenCL arhitektura se sastoji od nekoliko modela:

- Platformski model
- Programski model
- Memorijski model
- Izvršni model

3.1.1. Platformski model

OpenCL platformski model razlikuje dva entiteta:

Domaćin (engl. *host*) čiji je zadatak pokretanje i kontrola izvršavanja zadataka, pokrenutih na jednom ili više računskih uređaja (engl. *compute device*) koji su spojeni na domaćina. Ne postoje pravila oko načina spajanja OpenCL uređaja s domaćinom. U slučaj grafičke kartice to je najčešće PCI Express veza, ali u slučaju više CPU uređaja može biti i mrežna veza (engl. *ethernet*) koristeći TCP/IP protokol za prijenos podataka.

Računski uređaj (engl. *compute device*) je uređaj na kojem su pokrenuti računski zadaci, najčešće je to GPU ili CPU uređaj. Od svakog OpenCL računskog uređaja očekuje se dostupnost nekoliko računskih jedinica, koje unutar sebe sadržavaju više procesnih elemenata. Primjerice ako se radi o grafičkoj kartici to bi se preslikalo na sljedeći način:

- **OpenCL računski uređaj** - grafička kartica (GPU)
- **računska jedinica** - protočni multiprocesor
- **procesni element** - protočni procesor

3.1.2. Programski model

OpenCL podržava dva programska modela, paralelizaciju na razini podataka i paralelizaciju na razini zadatka. U modelu podatkovne paralelizacije, niz istih instrukcija se paralelno izvodi nad različitim podacima. Instrukcije se pritom paralelno izvode na svim računskim jedinicama. Kod modela paralelizacije na razini zadataka, različit skup instrukcija se pokreće na različitim računskim jedinicama. Kod ovog modela paralelizam se postiže korištenjem vektorskih tipova podataka koji su dostupni na OpenCL uređaju te stavljajući u red izvršavanja više zadataka.

Prilikom podatkovne paralelizacije svakoj dretvi je potreban jedinstveni identifikator kako bi mogla pristupati različitim podacima. OpenCL omogućava ovu funkcionalnost preko koncepta indeksnog prostora. OpenCL će dodijeliti identifikator grupi dretvi (engl. *workgroup ID*) koja se izvršava na pojedinoj računskoj jedinici, te dodatni identifikator svakoj dretvi unutar grupe koja se izvršava na procesnim elementima (engl. *work item ID*). Programer prilikom pokretanja dretvi mora definirati ukupan broj dretvi (engl. *global work size*) te broj dretvi unutar pojedine radne grupe (engl. *local work size*). Konačni broj radnih grupa se može dobiti tako da se ukupan broj dretvi podijeli s veličinom jedne radne grupe.

3.1.3. Memorijski model

OpenCL memorijski model definira pet vrsta memorije. Radne jedinice se izvršavaju na procesnim elementima te svaki ima svoju privatnu memoriju. Radna grupa se odvija računskoj jedinici koja ima svoju lokalnu memoriju kojoj mogu pristupati sve procesne jedinice koje sadrži. Globalnoj memoriji pristup imaju sve procesne jedinice [15]. Detaljan opis je dan u nastavku:

Memorija domaćina (engl. *host memory*) je područje memorije koji je vidljivo samo domaćinu. Fizički ova memorija bi odgovarala radnoj memoriji domaćina (primjerice kod procesora opće namjene to bi bila radna memorija (engl. *RAM - Random Access Memory*) na matičnoj ploči).

Globalna memorija je memorija u koju sve dretve, neovisno u kojoj se radnoj grupi nalaze, mogu pisati i čitati iz bilo koje lokacije. Fizički globalna memorija odgovara radnoj memoriji OpenCL uređaja. Kod procesora opće namjene (CPU) to bi bila radna memorija RAM (engl. *Random Access Memory*) na matičnoj ploči, dok bi kod grafičke kartice (GPU) to bio VRAM (engl. *Video Random Access Memory*).

Konstantna memorija je područje globalne memorije koje se ne mijenja tijekom izvršavanja jezgrenih funkcija (engl. *kernel functions*). Zauzimanje i postavljanje podataka u konstantu memoriju odgovornost je domaćina. Pojedine radne jedinice (engl. *work item*) mogu samo čitati iz konstantne memorije.

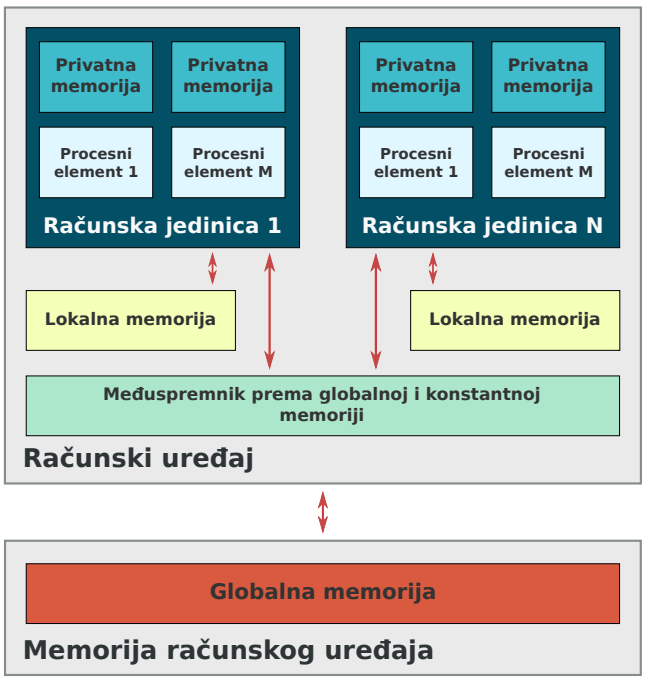
Lokalna memorija je memorija kojoj mogu pristupati sve dretve unutar iste radne grupe. Fizički lokalna memorija kod grafičke kartice odgovara dijeljenoj memoriji koju sadrži svaka računska jedinica. Ako OpenCL uređaj nema fizički zasebnu dijeljenu memoriju, koristit će se globalna memorija.

Privatna memorija je memorija namijenjena svakoj radnoj jedinici pojedinačno. Svaki procesni element ima pristup svojoj privatnoj memoriji. Varijable definirane unutar privatne memorije jedne radne jedinice, nisu vidljive drugim radnim jedinicama.

Tablica 3.1: Razine pristupa OpenCL memorijskog modela

Vrsta memorije	Pristup	Doseg	Može zauzeti
Globalna	R/W	Sve dretve + domaćin	Domaćin
Konstantna	R	Sve dretve + domaćin	Domaćin
Lokalna	R/W	Radna grupa dretvi	Radna grupa dretvi
Privatna	R/W	1 dretva	Dretva

Slika 3.1 prikazuje OpenCL memorijski model.



Slika 3.1: OpenCL memorijski model

OpenCL memorijski model je općeniti model i nije vezan niti za jednu konkretnu implementaciju. Od upravljačkih programa (engl. *driver*) konkretnih računskih uređaja se očekuje određivanje koja fizička memorija odgovara kojoj memoriji u OpenCL modelu.

3.1.4. Izvršni model

Izvršavanje OpenCL aplikacije odvija se u dvije faze: pokretanjem aplikacije na domaćinu, koja zatim pokreće skup jedne ili više jezgrenih funkcija (engl. *kernel function*) na OpenCL uređaju. Jezgrene funkcije su uglavnom jednostavne funkcije koje obavljaju operacije nad ulaznim podacima. OpenCL izvršni model definira postupak kojim se jezgrena funkcija pokreće na OpenCL uređaju. U nastavku će detaljnije biti objašnjeno nekoliko elemenata čije je razumijevanje nužno za shvaćanje načina na koji OpenCL izvršava jezgrene funkcije.

Kontekst

Računski zahtjevni dio posla OpenCL aplikacije odvija se na OpenCL uređaju. Međutim, jezgrene funkcije su inicijalno definirane na domaćinu, te se za njih stvara odgovarajući **kontekst** (engl. *context*) [15]. Kontekst definira okruženje u kojem će jezgrene funkcije biti inicijalizirane i izvršavane, te održava sljedeće resurse:

OpenCL uređaje (engl. *OpenCL devices*) domaćin može imati pristup više različitih OpenCL uređaja. Primjerice, moderniji procesori opće namjene imaju podršku za OpenCL, te su na većini današnjih modernih računala dostupna dva OpenCL uređaja - procesor opće namjene te jedna ili više grafičkih kartica.

Programske objekte koji sadrže izvorni i izvršni tekst programa jezgrenih funkcija. OpenCL jezgrene funkcije moraju se moći izvršavati na različitim uređajima koji imaju podršku za OpenCL. Kako bi se postigla takva visoka razina kompatibilnosti, izvorni programski kod se prilikom svakog pokretanja aplikacije na domaćinu iznova prevodi u izvršni oblik. Tek prilikom pokretanja aplikacije je moguće odrediti koji OpenCL uređaj će biti korišten za pokretanje jezgrenih funkcija, i u skladu s time pripremiti programski kod za izvršavanje na tom uređaju. Jedan programski objekt može sadržavati više jezgrenih funkcija.

Memorijske objekte koji omogućavaju alociranje memorijskog prostora koji će prilikom izvršavanja koristiti jezgrene funkcije. U heterogenim platformama kakva je OpenCL postoji više različitih memorijskih prostora, ovisno o korištenom OpenCL uređaju i njegovoj memorijskoj arhitekturi. Memorijski objekti omogućavaju prijenos podataka između memorije domaćina (engl. *host memory*) te različitih adresnih prostora dostupnih na uređaju.

Jezgrene funkcije koje su namijenjene izvršavanju na OpenCL uređajima. Izvorni i izvršni programski kod jezgrenih funkcija pohranjen je u programskim objektima.

Jezgrena funkcija

Jezgrena funkcija je definirana na domaćinu, te aplikacija na domaćinu pokreće naredbu za slanje jezgrene funkcije na izvršavanje na OpenCL uređaj. Kada domaćin pokrene takvu naredbu, OpenCL izvršna okolina (engl. *runtime environment*) stvara indeksni prostor odgovarajuće zadane veličine. Pojedinu instancu jezgrene funkcije nazivamo **radnom jedinicom** (engl. *work unit*), koja je jednoznačno definirana svojim položajem u indeksnom prostoru **globalnim identifikatorom** (engl. *global ID*).

Naredba kojom je jezgrena funkcija poslana na izvršavanje stvara skup radnih jedinica, gdje svaka radna jedinica izvršava isti slijed instrukcija definiran jezgrenom funkcijom. Iako radne jedinice izvršavaju isti slijed instrukcija, ponašanje pojedine radne jedinice može odudarati od ostalih ako instrukcije u sebi sadrže uvjetna grananja koja ovise o jedinstvenim identifikatorima radne jedinice.

Radne jedinice su organizirane u **radne grupe** (engl. *work groups*). Radne grupe pružaju krupnozrnatu raspodjelu indeksnog prostora, te ukupnom veličinom odgovaraju veličini indeksnog prostora. Drugim riječima, sve radne grupe su jednake veličine po pojedinim dimenzijama, te ta veličina jednoliko dijeli globalni indeksni prostor po pojedinim dimenzijama. Radnim grupama je dodijeljen jedinstveni **identifikator grupe** (engl. *work-group ID*) iste dimenzionalnosti kao i globalni indeksni prostor.

Radnim jedinicama također je pridružen jedinstveni **lokalni identifikator** (engl. *local ID*) unutar radne grupe. Prostorni položaj radne jedinice se zatim može jednoznačno odrediti koristeći globalni identifikator ili kombinacijom lokalnog identifikatora i identifikatora radne grupe kojoj pripada. Radne jedinice unutar iste radne grupe istodobno se izvršavaju na jednoj računskoj jedinici OpenCL uređaja.

Indeksni prostor

Indeksni prostor je N-dimenzionalan (engl. *NDRange*), gdje N može poprimiti vrijednosti 1, 2 ili 3. Unutar OpenCL aplikacije na domaćinu, NDRange je definiran cjelobrojnim poljem duljine N u kojem su definirane veličine pojedinih dimenzija. Ovaj koncept je najlakše pokazati na primjeru dvodimenzionalnog indeksnog prostora, pri tome koristeći sljedeće oznake:

$\mathbf{G_x}$ **i** $\mathbf{G_y}$ oznake definiraju veličinu pojedinih dimenzija u globalnom indeksnom prostoru označenih sa x i y. Položaj pojedine radne jedinice unutar globalnog indeksnog prostora dan je koordinatama (g_x, g_y). Koordinate mogu poprimiti vrijednosti iz intervala $[0 \dots G_x - 1, 0 \dots G_y - 1]$

$\mathbf{W_x}$ **i** $\mathbf{W_y}$ oznake definiraju broj radnih grupa po pojedinim dimenzijama, te su dimenzije označene sa x i y. Koordinate pojedine radne grupe označavaju se kao (w_x, w_y).

$\mathbf{L_x}$ **i** $\mathbf{L_y}$ oznake definiraju veličinu pojedinih dimenzija unutar jedne radne grupe. Kako OpenCL zahtjeva da ukupan broj radnih grupa jednoliko dijeli globalni indeksni prostor, sve radne grupe su iste veličine po svim dimenzijama. Taj indeksni prostor unutar radne grupe se naziva lokalni indeksni prostor. Koordinate pojedine radne jedinice unutar radne grupe označavaju se kao (l_x, l_y).

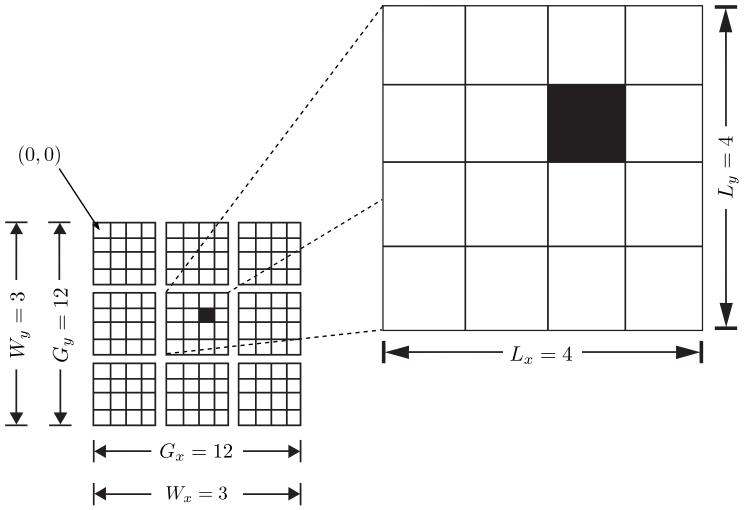
Na slici 3.2 prikazan je primjer takvog dvodimenzionalnog indeksnog prostora. U primjeru zacrnjeni kvadrat predstavlja radnu jedinicu s globalnim koordinatama $(g_x, g_y) = (6, 5)$, koordinatama radne grupe $(w_x, w_y) = (1, 1)$ te lokalnim koordinatama unutar radne grupe $(l_x, l_y) = (2, 1)$.

U ovakvom dvodimenzionalnom primjeru koordinate radnih grupa po pojedinim dimenzijama mogu se računati izrazima 3.1:

$$\begin{aligned} L_x &= G_x / W_x \\ L_y &= G_y / W_y \end{aligned} \tag{3.1}$$

Globalni identifikatori radne jedinice po pojedinim dimenzijama se mogu računati koristeći izraze 3.2:

$$\begin{aligned} g_x &= w_x * L_x + l_x \\ g_y &= w_y * L_y + l_y \end{aligned} \tag{3.2}$$



Slika 3.2: Primjer dvodimenzionalnog indeksnog prostora

3.2. Programski jezik OpenCL C

Programski jezik OpenCL C temelji se na C99 specifikaciji programskog jezika C, uz nekoliko ograničenja i dodataka jeziku kako bi se olakšalo korištenje paralelne programske paradigme. Kako je opisano u potpoglavlju 3.1.4, OpenCL definira višedimenzionalni globalni indeksni prostor čija veličina odgovara ukupnom broju radnih jedinica. Na primjeru će biti prikazano kako se oblikuje jezgrena funkcija koja obavlja paralelno zbrajanje elemenata polja cijelih brojeva. Slijedna verzija takve funkcije pisane u programskom jeziku C prikazana je programskim odsječkom 3.1.

```
void zbroji (int duljina, const int *a, const int *b, int *rezultat) {

    for (int index=0; index < duljina; index++) {
        rezultat[index] = a[index] + b[index];
    }

}
```

Tekst programa 3.1: Funkcija zbrajanja elemenata 2 cjelobrojna polja

Programskim odsječkom 3.2 prikazana je ekvivalentna OpenCL jezgrena funkcija.

```
__kernel void paralelno_zbroji (
        __global const int *a,
        __global const int *b,
        __global int *rezultat) {

    int id = get_global_id(0);
    rezultat[id] = a[id] + b[id];

}
```

Tekst programa 3.2: OpenCL jezgrena funkcija zbrajanja elemenata dvaju cjelobrojnih polja

Funkcija `paralelno_zbroji` deklarirana je kao OpenCL jezgrena funkcija koristeći kvalifikator `__kernel`. U ovom primjeru korišten je jednodimenzionalni globalni indeksni prostor, čija veličina odgovara broju elemenata polja koji ćemo označiti sa n. Jezgrena funkcija se paralelno izvršava na n radnih jedinica, koje pritom koriste globalni identifikator za pronalazak elementa polja za koji su zadužene. Naredba `get_global_id(0)` vraća jedinstveni jednodimenzionalni globalni identifikator pojedine radne jedinice.

OpenCL C specifikacija ima nekoliko dodataka u odnosu na C99 specifikaciju:

Vektorski tipovi podataka koji omogućavaju zapis vektora duljine do 16 elemenata istog skalarnog tipa podataka (`int4`, `float16`, ...itd.). Vektorski tipovi podataka se za-

tim mogu koristiti poput skalarnih tipova podataka. Ugrađene funkcije dostupne unutar OpenCL C programskog jezika omogućuju efikasno provođenje SIMD operacija nad vektorskim tipovima.

Kvalifikatori adresnog prostora omogućuju pristup i dodjeljivanje memorijskog prostora različitih vrsta memorije dostupnih u hijerarhiji OpenCL modela (tablica 3.2). Kvalifikator `__private` je unaprijed zadan ako niti jedan drugi kvalifikator nije naveden prilikom deklaracije varijable. OpenCL memorijski model detaljnije je objašnjen u potpoglavlju 3.1.3.

Tablica 3.2: OpenCL C memorijski kvalifikatori

Kvalifikator	Vrsta memorije
`__global`	Globalna memorija
`__local`	Lokalna (dijeljena) memorija
`__private`	Privatna memorija
`__constant`	Konstantna memorija

OpenCL specifične naredbe su naredbe kojima se primjerice, može pristupati pojedinim jedinstvenim identifkatorima u indeksnom prostoru ili provoditi sinkronizacija između radnih jednica (tablica 3.3).

Tablica 3.3: OpenCL C specifične naredbe

Naredba	Opis
`get_global_id(DIM)`	Dohvaća globalni identifikator za dimneziju DIM
`get_local_id(DIM)`	Dohvaća lokalni identifikator za dimenziju DIM
`get_group_id(DIM)`	Dohvaća identifikator grupe za dimneziju DIM
`barrier(FLAG)`	Sinkronizacija radne grupe preko pristupa memoriji, $FLAG$ može biti $CLK\_LOCAL\_MEM\_FENCE$ ili $CLK\_GLOBAL\_MEM\_FENCE$

Ugrađene funkcije su razne matematičke (`sqrt`, `sin`, `cos`, ... itd.) i geometrijske funkcije dostupne unutar OpenCL C specifikacije. Određeni OpenCL uređaji poput grafičkih kartica imaju i na sklopovskoj razini implementirane razne matematičke funkcije koje se izvršavaju u jednom taktu, ali imaju nešto manju točnost. Pristup takvim funkcijama moguć je prefiksom `native`, primjerice `native_sin` [21].

Osim ovih dodataka postoji nekoliko ograničenja, od kojih su najvažnija navedena u nastavku:

- argumenti jezgrenih funkcija smiju koristiti samo kvalifikatore `__global`, `__constant` i `__local`
- argumenti jezgrene funkcije ne smiju biti dvostruki pokazivači
- jezgrena funkcija mora imati povratnu vrijednost tipa `void`
- nije dopušteno korištenje varijabilnih polja unutar jezgrene funkcije
- nisu podržane rekurzije

3.3. Java OpenCL

OpenCL programske biblioteke izvorno su pisane za programski jezik C, te su naknadno razvijene poveznice (engl. *bindings*) za druge programske jezike poput Pythona, Jave, C++ ... itd. U ovom radu su prilikom programske implementacije korištene JOCL (engl. *Java OpenCL*) poveznice namijenjene korištenju u sklopu programskog jezika Java.

Dostupne su dvije inačice JOCL poveznica, koje se razlikuju po razini apstrakcije. Jednostavna inačica poveznica je vrlo slična originalnoj OpenCL API specifikaciji. OpenCL funkcije su dostupne kroz više statičkih metoda, te je programski kod jako sličan C programskom kodu zbog korištenja JNI-a (engl. *Java Native Interface*) prilikom pristupa C poveznicama.

JOGAMP JOCL poveznice su poveznice više razine apstrakcije koje pružaju objektni, više Javi slični, pristup OpenCL funkcionalnosti. Mnoge naredbe poput dohvaćanja informacija o OpenCL uređajima te specifikacijama uređaja (maksimalna veličina radne grupe, veličina lokalne memorije, ... itd.) su znatno jednostavnije za korištenje u ovoj inačici. Prilikom prijenosa podataka između OpenCL uređaja i JVM-a (engl. *Java Virtual Machine*) koriste se Java NIO biblioteka u cilju povećanja performansi.

4. Simetrični kriptografski algoritmi

Kada se govori o kriptografiji općenito se misli na zaštitu podataka pomoću matematičkih postupaka ili algoritama koji koriste kriptografske ključeve. S obzirom na vrste kriptografskih ključeva, današnje algoritme možemo podijeliti na simetrične i asimetrične kriptografske algoritme. U ovom će poglavlju biti opisana dva simetrična kriptografska algoritma, DES (engl. *Data Encryption Standard*) i AES (engl. *Advanced Encryption Standard*), te njihova implementacija na grafičkog procesoru koristeći standard OpenCL [1] [12] [16].

Osnovna karakteristika simetrične kriptografije je identičnost kriptografskih ključeva koji se koriste prilikom kriptiranja i dekriptiranja podataka. Postupci kriptiranja i dekriptiranja mogu se zapisati u obliku prikazanom formulama 4.1:

$$\begin{aligned} C &= E(P, K) \\ P &= D(C, K) \end{aligned} \tag{4.1}$$

gdje **P** predstavlja izvorni (jasni) tekst (engl. *plaintext*) dok **C** predstavlja kriptirani tekst (engl. *ciphertext*). Postupkom kriptiranja funkcijom **E** jasni tekst se prevodi u kriptirani tekst. Obrnuti postupak prevođenja kriptiranog teksta u jasni tekst naziva se dekriptiranjem i provodi se funkcijom **D**. Funkcije **E** i **D** mogu biti istovjetne, tj. funkcija dekriptiranja može biti inverzna funkciji kriptiranja kako je prikazano izrazom 4.2. Obje funkcije u radu koriste parametar ili ključ kriptiranja **K**, čija je duljina definirana algoritmom.

$$P = D(E(P, K), K) \tag{4.2}$$

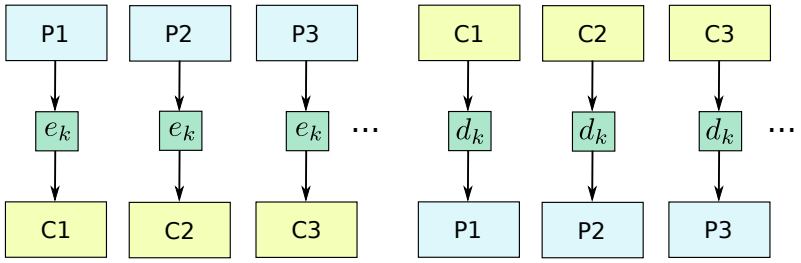
Algoritmi opisani u ovom poglavlju provode ove funkcije na razini jednog bloka podataka (engl. *block cipher*) čiju veličinu također zasebno definira svaki algoritam. Prilikom kriptiranja većeg broja blokova definirano je nekoliko načina kriptiranja, čija je svrha osigurati što sigurniji konačni rezultat kriptiranja [2]. Neki od tih načina kriptiranja koriste dodatni parametar zvan **inicijalizacijski vektor** (engl. *initialization vector, IV*). Svrha inicijalizacijskog vektora je osigurati uvijek različite kriptirane blokove, čak i ako su nastali od istog bloka jasnog teksta i istog ključa, te njegova veličina odgovara veličini bloka.

4.1. Načini kriptiranja

U nastavku su opisana četiri osnovna načina kriptiranja te mogućnosti njihove paralelizacije.

ECB

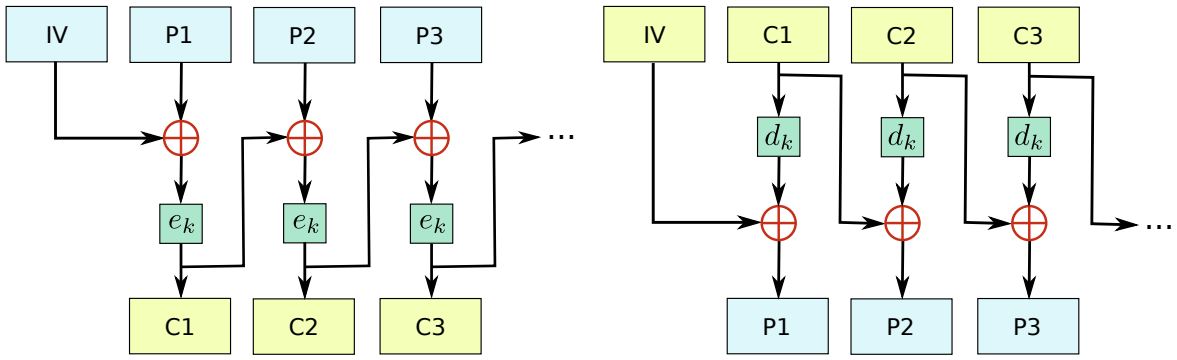
Elektronička bilježnica (engl. *Electronic Codebook, ECB*) je najjednostavniji način kriptiranja. Kod ECB načina se svaki blok zasebno kriptira, odnosno dekriptira. Ovaj način kriptiranja je jako pogodan za paralelizaciju jer su postupci kriptiranja (odnosno dekriptiranja) pojedinih blokova podataka međusobno neovisni.



Slika 4.1: ECB način kriptiranja (enkripcija i dekripcija)

CBC

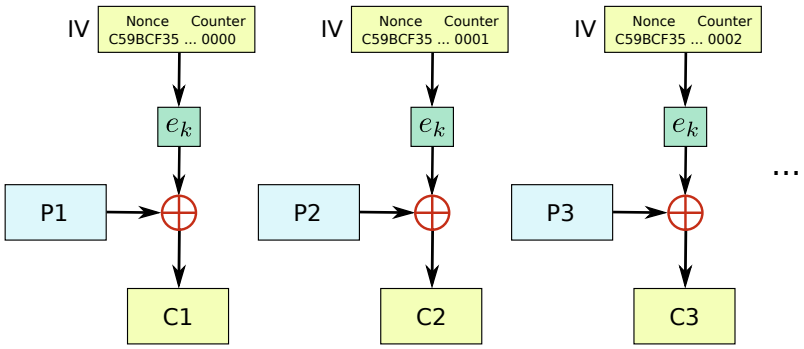
Ulančavanje (engl. *Cipher Block Chaining, CBC*) najpopularniji je način kriptiranja blokova jasnog teksta. Jasni tekst prvog bloka se zbraja (XOR) s inicijalizacijskim vektorom (engl. *initialization vector, IV*) te se šalje u kriptografski algoritam. Rezultirajući kriptirani tekst se zatim koristi kao IV sljedećeg bloka. Kako prilikom postupka kriptiranja svaki blok ovisi o rezultatu kriptiranja prethodnog bloka paralelizacija u tom smjeru nije moguća. Prilikom dekriptiranja je paralelizacija moguća, jer je za dekriptiranje jednog bloka potrebno poznavati samo kriptirani prethodni blok.



Slika 4.2: CBC način kriptiranja (enkripcija i dekripcija)

CTR

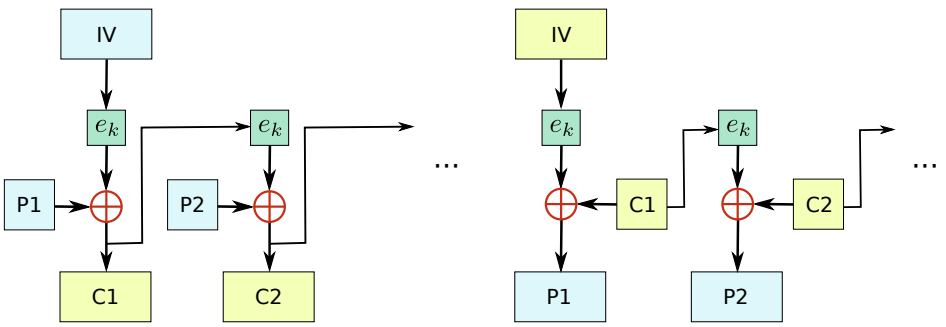
Brojač (engl. *Counter, CTR*) je način kriptiranja prilikom kojeg se inicijalizacijski vektor svakog sljedećeg bloka, dobiva uzastopnim uvećavanjem vrijednosti brojača sadržanog u sklopu početnog inicijalizacijskog vektora. Tako dobiveni IV se zatim šalje u kriptografski algoritam te se rezultat tog postupak zbraja (XOR) s jasnim tekstom. Ovo je od posebnog značaja prilikom paralelne obrade više blokova koristeći OpenCL, jer se kao brojač može koristiti jedinstveni **globalni identifikator** radne jedinice. Funkcija kriptiranja se koristi prilikom kriptiranja ali i dekriptiranja.



Slika 4.3: CTR način kriptiranja (enkripcija i dekripcija)

CFB

Povratna vrijednost (engl. *Cipher Feedback, CFB*) je način kriptiranja jako sličan CBC načinu. CFB dekriptiranje je u biti obrnuto CBC kriptiranje. Dekriptiranje je ujedno i jedini smjer koji je moguće izvesti paralelno nad više blokova. Prilikom CFB dekriptiranja se inicijalizacijski vektor prvog bloka **kriptira** kriptografskim algoritmom te rezultat zbraja (XOR) s kriptiranim tekstom. Kod preostalih blokova kao inicijalizacijski vektor koristi se kriptirani tekst prethodnog bloka. Funkcija kriptiranja se koristi prilikom kriptiranja ali i dekriptiranja.



Slika 4.4: CFB način kriptiranja (enkripcija i dekripcija)

4.2. Data Encryption Standard

DES (engl. *Data Encryption Standard*) je simetrični kriptografski sustav razvijen sredinom 70-tih u IBM-u, te je kasnije prihvaćen kao federalni standard u SAD-u do kraja devedesetih godina (kada ga je zamijenio AES). DES kriptosustav je u biti kriptografski algoritam **Lucifer**. Algoritam je razvio IBM-ov inženjer Horst Feistel, te je jezgra algoritma prema njemu dobila ime - **Feistelova mreža** [2].

DES je sustav koji pretvara **64 bitne** blokova jasnog teksta u 64 bitne blokove kriptiranog teksta. Duljina kriptografskog ključa je **64 bita**, od koji 8 bitova otpada na provjeru pariteta, pa je efektivna duljina ključa 56 bita.

Prije početka kriptiranja 64 bitni kriptografski ključ mora se proširiti na 16 48 bitnih potključeva ($K_1 \dots K_{16}$). Na početku algoritma se nad 64 bitnim ulaznim blokom P provodi inicijalna permutacija (IP) te se dobiva 64 bitni blok $IP(P)$. Tako dobiveni blok zatim ulazi u Feistelovu mrežu koja se sastoji od 16 krugova (engl. *rounds*). Prvo se rezultat permutacije podijeli na dvije polovine L_0 i R_0 svaka duljine 32 bita, kako je prikazano izrazom 4.3.

$$L_0 R_0 = IP(P) \tag{4.3}$$

Zatim se prolazi kroz 16 krugova Feistelove mreže u kojima se, uz $i = 1, 2, \dots 16$, obavljaju operacije opisane izrazima 4.4.

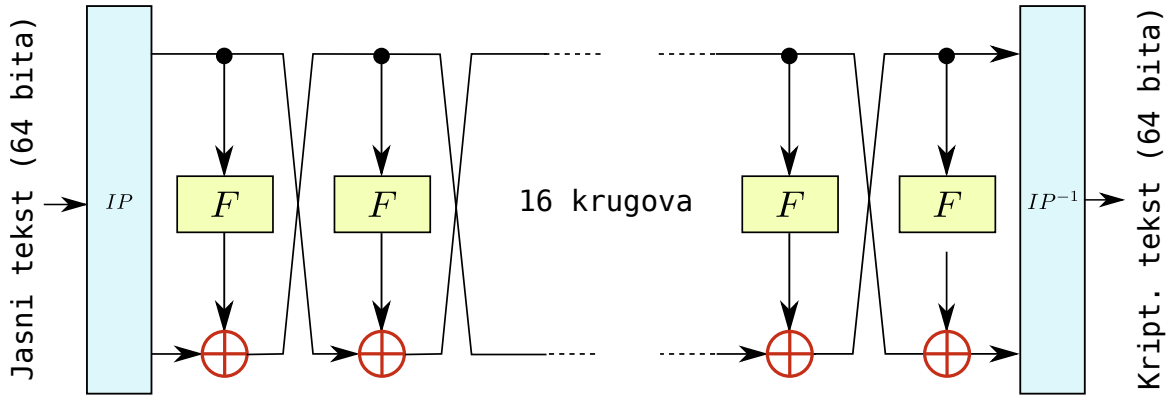
$$\begin{aligned} L_i &= R_{i-1} \\ Ri &= L_{i-1} \oplus F(R_{i-1}, K_i) \end{aligned} \tag{4.4}$$

Funkcija $F(R_{i-1}, K_i)$ predstavlja Feistelovu funkciju koja obavlja preslagivanje bitova u podbloku R_{i-1} ovisno o potključu K_i. Na kraju 16 kruga se podblokovi spajaju, te se nad rezultirajućim blokom provodi inverzna inicijalna permutacija (IP^{-1}), čime se dobiva konačni kriptirani blok kako je prikazano izrazom 4.5.

$$C = IP^{-1}(R_{16} L_{16}) \tag{4.5}$$

Feistelova funkcija (F) prilikom obrade polubloka prvo proširuje polublok na 48 bita koristeći permutacijsku tablicu E. Tako prošireni polublok se zatim zbraja (XOR) sa potključem te prolazi kroz proces supstitucije bitova koristeći 8 S-kutija. S-kutija je supstitucijska tablica koji uvodi nelinearnost u DES algoritam. Supstituiraju se blokovi od po šest bitova,

pri čemu krajnji lijevi i desni bit u bloku određuju red u tablici, a preostala četiri središnja bita određuju stupac. Nakon supstitucije se polublok ponovo smanjuje na 32 bita koristeći permutacijsku tablicu P. Proces se ponavlja kroz 16 krugova kako je prikazano na slici 4.5.



Slika 4.5: DES algoritam

4.2.1. Opis implementacije

Implementacija OpenCL jezgrene funkcije se temelji na paralelizaciji na razini jednog bloka. Više blokova se paralelno kriptira, odnosno dekriptira, te pritom jedna radna jedinica obrađuje jedan blok. Prototip jezgrene funkcije koja izvršava DES algoritam nad jednim blokom prikazan je odsječkom 4.1.

```
__kernel void cryptoDES (
    __global const ulong *input,
    __global ulong *output,
    __constant const ulong *roundKeys,
    const uint blockCount,
    const ulong iv
)
```

Tekst programa 4.1: Prototip jezgrene funkcije kriptiranja algoritma DES

Prilikom kriptiranja argument `input` predstavlja pokazivač na polje blokove jasnog teksta koji se nalaze u **globalnoj memoriji**. Pritom se koristi `ulong` tip podatka koji svojom veličinom od 64 bita odgovara veličini jednog bloka teksta DES algoritma. Također putem argumenta `output`, održavamo pokazivač na poziciju u globalnoj memoriji gdje će biti zapisani rezultirajući kriptirani blokovi. Argument `blockCount` daje informaciju o broju blokova koje treba obraditi, dok argument `iv` predstavlja inicijalizacijski vektor koji je korišten u nekim načinima kriptiranja.

Argument `roundKeys` je pokazivač na polje od 16 potključeva koji su smješteni u konstantnu memoriju. Konstantna memorija je odabrana jer će sve radne jedinice u istom trenutku dohvaćati pojedini potključ, što će rezultirati prijenosnom (engl. *broadcast*) operacijom. S-kutije, te permutacijske tablice (IP, IP^{-1}, E, P) su također smještene u konstantnoj memoriji. Pseudokod jezgrene funkcije kriptiranja ECB načinom prikazan je algoritmom 4.2.

Algoritam 4.2 Jezgrena funkcija kriptiranja algoritma DES (ECB)

$gId \leftarrow dohvati\_globalni\_ID()$
if $gID \geq blockCount$ **then**
    return
end if
$blok = input[gId]$
$kriptiraniBlok \leftarrow algoritamDES(blok)$
$result[gId] \leftarrow kriptiraniBlok$

Na početku jezgrene funkcije se prvo dohvaća globalni identifikator radne jedinice. Identifikator služi kao indeks kojim se iz globalne memorije dohvaća odgovarajući podatkovni blok, te ga se sprema u privatnu memoriju radne jedinice [19]. Nakon što je nad blokom proveden DES algoritam, blok se zapisuje natrag u globalnu memoriju na poziciju određenu globalnim identifikatorom.

4.2.2. Rezultati

U ovom potpoglavlju prikazani su rezultati usporedbe ubrzanja implementacije DES algoritam koristeći standard OpenCL (koristeći Java OpenCL poveznice), te implementacije unutar Java Cipher paketa, koji je dio standardne Java distribucije.

Mjerenja su provedena na računalu sa specifikacijama prikazanim tablicom 4.1, te je računalo opremljeno grafičkom karticom čije su specifikacije prikazane tablicom 4.2. Specifikacije grafičke kartice su pritom izravno očitane s uređaja koristeći OpenCL.

Tablica 4.1: Testno računalo

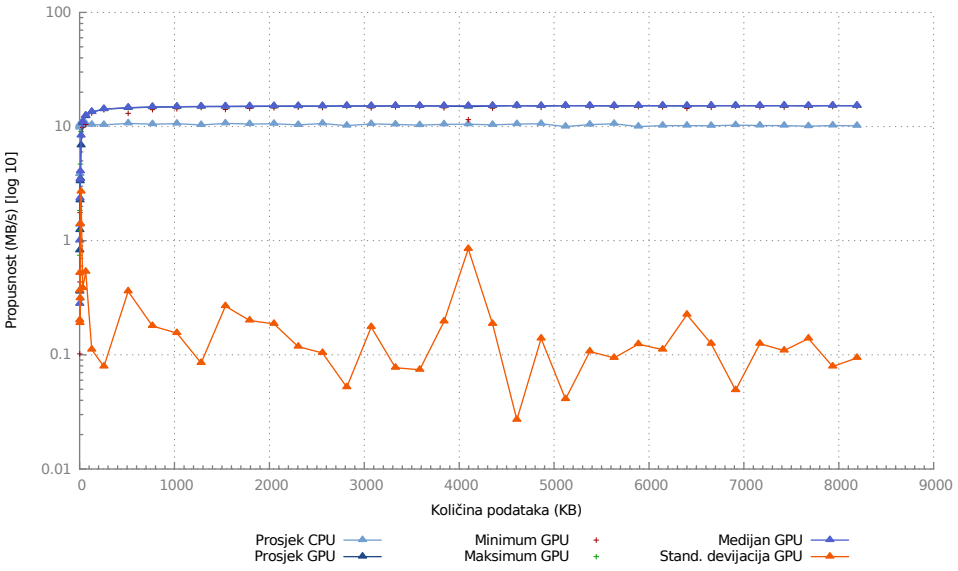
Procesor	AMD Athlon64 3200+
Takt procesora	2.2 GHz
Broj jezgri	1 jezgra
Memorija	2 GB DDR2

Tablica 4.2: NVIDIA GeForce 9600GT

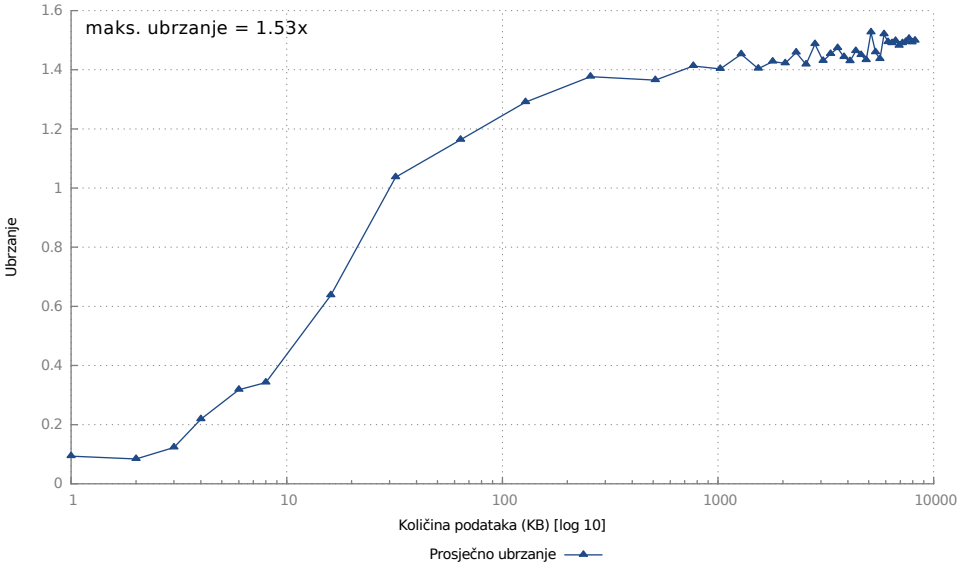
Protočnih multiprocesora	8
Protočnih procesora	64
Takt procesora	1675 MHz
Globalna memorija	512 MB GDDR3
Dijeljena memorija	16 KB
Takt memorije	900 MHz

U grafovima prikazanim u nastavku mjerena je propusnost (izražena u megabajtima u sekundi) te prosječno ubrzanje dekriptiranja. Mjerenja su provedena dekriptirajući blokove podataka veličine 256 kilobajta do 8 MB megabajta, u koracima od 256 kilobajta. Na grafovima je također istaknuto maksimalno prosječno ubrzanje koje je postignuto. Mjerenja su provedena za ECB, CTR i CBC način (CFB način je preskočen jer je u biti obrnuto CBC enkriptiranje, te su rezultati vrlo slični).

Grafovi 4.6 i 4.7 prikazuju rezultate dekriptiranja koristeći ECB način dekriptiranja.

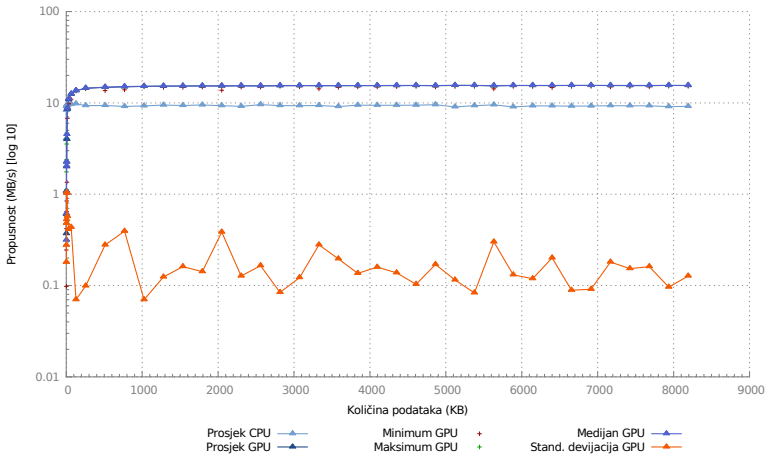


Slika 4.6: Propusnost, DES dekriptiranje, ECB način

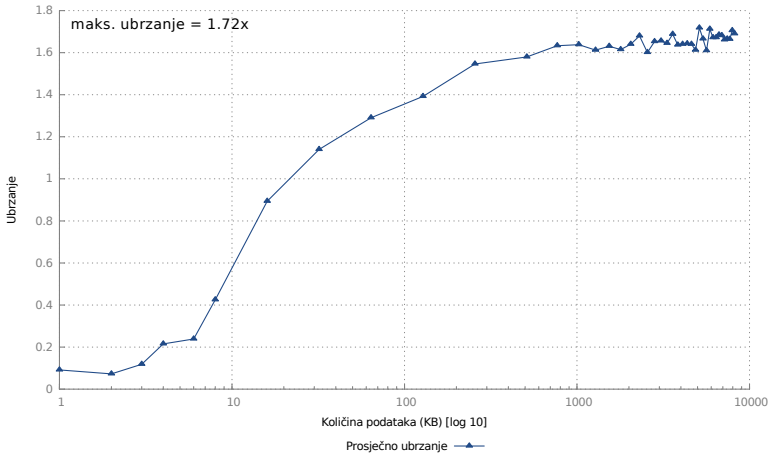


Slika 4.7: Prosječno ubrzanje, DES dekriptiranje, ECB način

Grafovi 4.8 i 4.9 prikazuju rezultate dekriptiranja koristeći CTR način dekriptiranja.

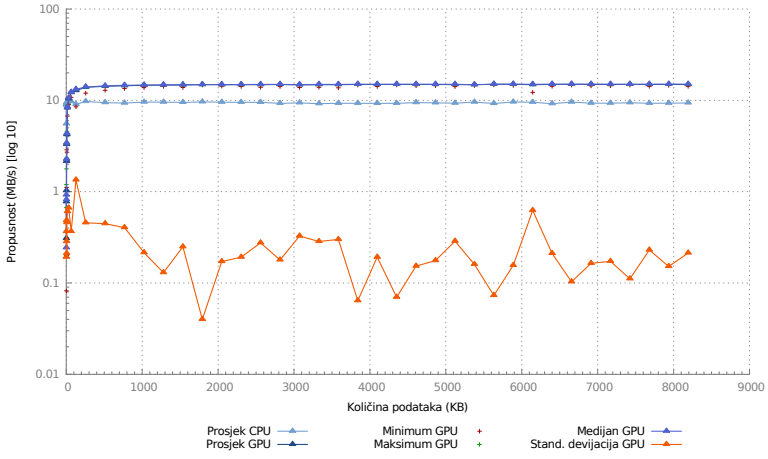


Slika 4.8: Propusnost, DES dekriptiranje, CTR način

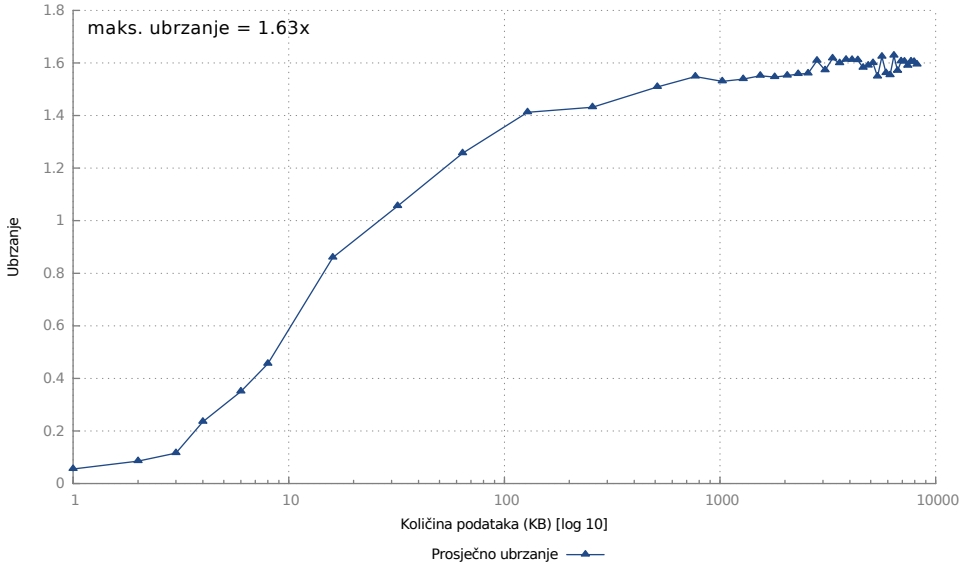


Slika 4.9: Prosječno ubrzanje, DES dekriptiranje, CTR način

Grafovi 4.10 i 4.11 prikazuju rezultate dekriptiranja koristeći CBC način dekriptiranja.

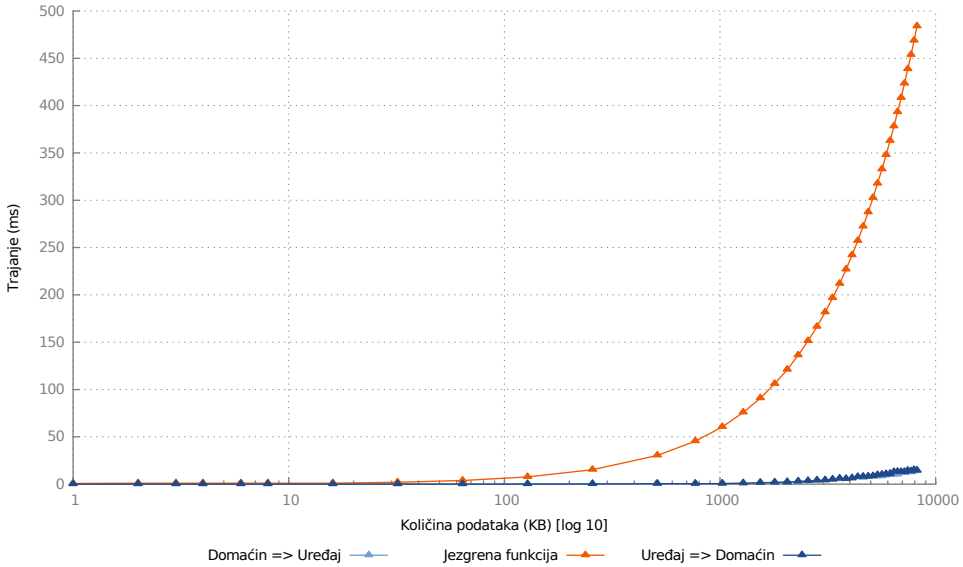


Slika 4.10: Propusnost, DES dekriptiranje, CBC način



Slika 4.11: Prosječno ubrzanje, DES dekriptiranje, CBC način

Kod OpenCL implementacije je osim ukupnog trajanja dekriptiranja mjereno i trajanje po komponentama. Graf 4.12 prikazuje trajanje prijenosa podataka s domaćina na grafičku karticu (OpenCL uređaj), trajanje izvođenja jezgrene funkcije te trajanje dohvaćanja rezultata dekriptiranja natrag u memoriju domaćina.



Slika 4.12: Trajanje dekriptiranja po komponentama, CTR način (DES)

Iz rezultata po komponentama je vidljivo da se najviše vremena potrošeno na izvođenje jezgrene funkcije. Ovi rezultati kao i nešto lošiji rezultati ukupnog ubrzanja od samo 70% su očekivani. Posljedica su činjenice da DES tokom izvođenja ima oko 1500 pristupa konstantnoj memoriji, zbog čestog korištenja nekoliko različitih permutacijskih tablica.

4.3. Advanced Encryption Standard

AES (engl. *Advanced Encryption Standard*) je simetrični kriptografski sustav, koji je odabran putem natječaja koji je 1997. raspisao američki Nacionalni institut za standarde i tehnologiju (engl. *National Institute of Standards and Technology, NIST*). Algoritam koji je 2000. odabran je algoritam **Rijndael**, čiji su autori kriptografi Joean Daemen i Vincent Rijemen.

U svojoj izvornoj verziji Rijandael je podržavao veličinu bloka veću od 128 bitova, ali je za AES to fiksirano na 128 bitova (16 okteta), dok ključ kriptiranja može biti veličine 128, 192 ili 256 bitova (odnosno 16, 24 ili 32 okteta). AES kriptiranje svodi se na uzastopnu primjenu određenih transformacija nad ulaznim blokom podataka. Transformacije se obavljaju u krugovima, gdje je broj krugova određen veličinom kriptografskog ključa. Tablica 4.3 prikazuje ovisnost broja krugova o veličini ključa [2].

Tablica 4.3: Ovisnost broja krugova o veličini ključa (AES)

Veličina ključa	Broj krugova
128 bita	10
192 bita	12
256 bita	14

Ključni element u radu AES algoritma je konačno **Galoisovo polje GF(2^8)** [2]. Elementi tog polja su polinomi oblika prikazanog izrazom 4.6.

$$a_7x^7 + a_6x^6 + \ldots + a_1x + a_0, a_i \in \{0, 1\} \tag{4.6}$$

Svaki oktet podataka $a_7a_6a_5a_4a_3a_2a_1a_0$ (niz od 8 bitova) predstavljen je odgovarajućim polinomom. Nad elementima konačnog polja GF(2^8) definirane su operacije zbrajanja i množenja. Zbrajanje se svodi na logičku binarnu operaciju isključivo ILI (engl. *XOR*). Množenje je binarno množenje polinoma modulo fiksni ireducibilni polinom $g(x) = x^8 + x^4 + x^3 + x + 1$.

a_{00}	a_{01}	a_{02}	a_{03}
a_{10}	a_{11}	a_{12}	a_{13}
a_{20}	a_{21}	a_{22}	a_{23}
a_{30}	a_{31}	a_{32}	a_{33}

Slika 4.13: 128 bitni AES blok

Prilikom kriptiranja se blok jasnog teksta smješta u pravokutni niz okteta dimenzija 4×4, kako je prikazano tablicom 4.13. Na sličan način tretira se i ključ, čiji blok ovisno o veličini

ključa može imati 4, 6, ili 8 stupaca (što odgovara veličinama ključa od 128, 192 i 256 bitova).

Postupak kriptiranja

Postupak kriptiranja odvija se u koracima:

1. Proširenje ključa kriptiranja (generiranje potključeva)
2. Inicijalni korak
   (a) Dodaj potključ (XOR)
3. Krugovi (ponavljaj $brojKrugova - 1$ puta)
   (a) Zamijeni oktete
   (b) Posmakni redove
   (c) Pomiješaj stupce
   (d) Dodaj potključ (XOR)
4. Posljednji korak
   (a) Zamijeni oktete
   (b) Posmakni redove
   (c) Dodaj potključ (XOR)

Proširenje ključa kriptiranja

Potključevi veličinom odgovaraju veličini bloka (16 okteta) i generiraju se iz izvornog ključa (koji može biti duljine 16, 24 ili 32 okteta). U svakom krugu kriptiranja koristi se različiti potključ. Kako veličina proširenog ključa ovisi o broju krugova, može se prikazati izrazom 4.7.

$$velicinaProsirenogKljuca = (brojKrugova + 1) * 16 \tag{4.7}$$

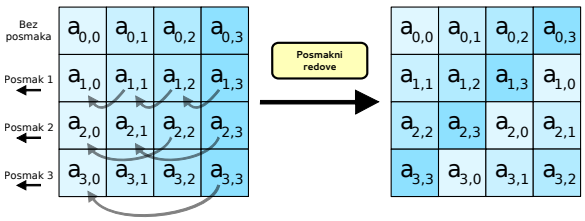
Prošireni ključ se generira na način da se izvorni ključ kopira na početak proširenog, a ostatak se gradi koristeći rotaciju bitova (kružni posmak), zamjenu okteta koristeći supstitucijske tablice te dodavanje konstanti [2].

Zamijeni oktete

Funkcija zamijeni oktete (engl. *sub bytes*) mijenja ulazni blok oktet po oktet koristeći supstitucijsku tablicu (S-kutija (engl. *S-box*)).

Posmakni redove

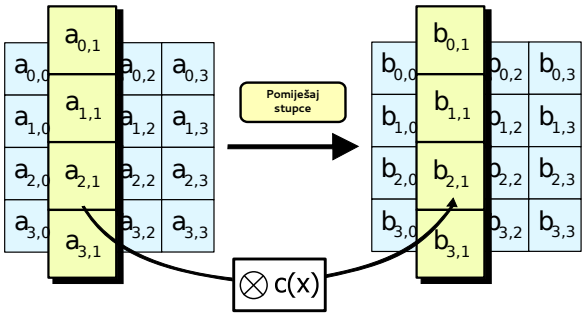
Funkcija posmakni redove (engl. *shift rows*) kružno posmiče oktete u lijevo za određeni pomak. Prvi redak ostaje nepromijenjen. Drugi redak se posmiče za jedan oktet, treći za dva okteta, a četvrti za tri okteta.



Slika 4.14: Posmakni redove (AES)

Pomiješaj stupce

Funkcija pomiješaj stupce (engl. *mix columns*) svaki stupac bloka promatra kao polinom u Galoisovom polju GF(2^8) te ga množi polinomom $c(x) = 3x^4 + x^2 + x + 2$ modulo fiksni ireducibilni polinom $g(x) = x^8 + x^4 + x^3 + x + 1$.



Slika 4.15: Pomiješaj stupce (AES)

Dekriptiranje je isti postupak kao kriptiranje osim što se koriste inverzni oblici navedenih funkcija. Također je potrebna inverzna supstitucijska tablica, odnosno inverzna S-tablica.

4.3.1. Opis implementacije

Kao i kod DES algoritma, AES jezgrena funkcija provodi algoritam nad jednim blokom podataka [22] [7] [13]. Prototip jezgrene funkcije prikazan je odsječkom 4.3.

```
__kernel void cryptoAES(
    __global const uchar *input,
    __global uchar *output
    __constant const uchar *roundKeys,
    const uint roundNum,
    const uint blockCount
)
```

Tekst programa 4.3: Prototip jezgrene funkcije kriptiranja algoritma AES

Argumenti `input` i `output` predstavljaju polja u globalnoj memoriji za čitanje jasnih blokova teksta i zapisivanje rezultirajućih kriptiranih blokova. Argument `roundKeys` je pokazivač na polje proširenih ključeva koje se nalazi u **konstantnoj memoriji**. Argumenti `roundNum` i `blockCount` nam govore koliko će krugova algoritam imati (ovisno o veličini ključa), odnosno koliko blokova treba obraditi.

Kako je AES algoritam čije transformacije rade s oktetima, odabran je tip podatka `uchar` čija je duljina 1 oktet. U konstantu memoriju su također smještene supstitucijska i inverzna supstitucijska tablica.

Algoritam 4.4 Jezgrena funkcija kriptiranja algoritma AES (ECB)

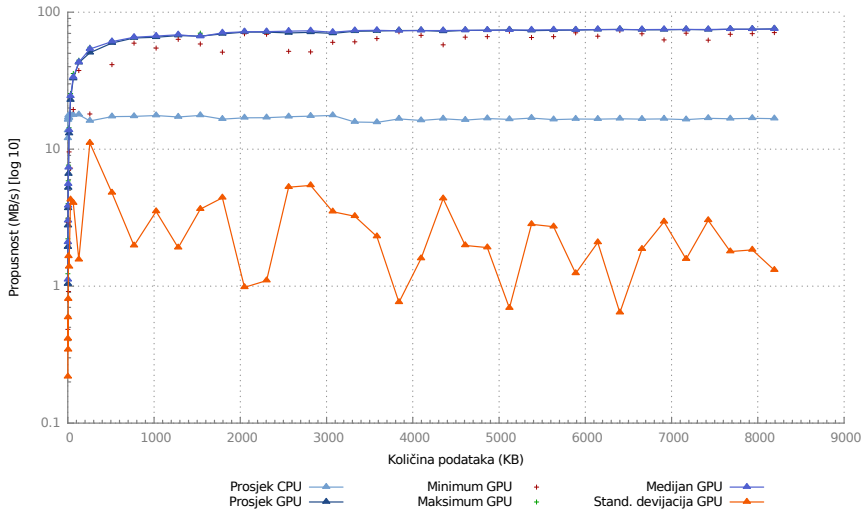
$gId \leftarrow dohvati\_globalni\_ID()$
if $gId \geq blockCount$ **then**
    return
end if
$startPos \leftarrow gId * 16$
for $index = 0$ **to** 15 **do**
    $block[index] \leftarrow input[startPos + index]$
end for
$kriptiraniBlok \leftarrow algoritamAES(blok)$
for $index = 0$ **to** 15 **do**
    $output[startPos + index] \leftarrow kriptiraniBlok[index]$
end for

Kao i kod prethodnog algoritma, prvo se provodi kopiranje bloka od 16 okteta iz globalne u privatnu memoriju radne jedinice te se blok zatim obrađuje. Nakon obrade rezultirajući blok treba prepisati natrag na odgovarajuće mjesto u globalnoj memoriji. Ovaj postupak je prikazan algoritmom 4.4.

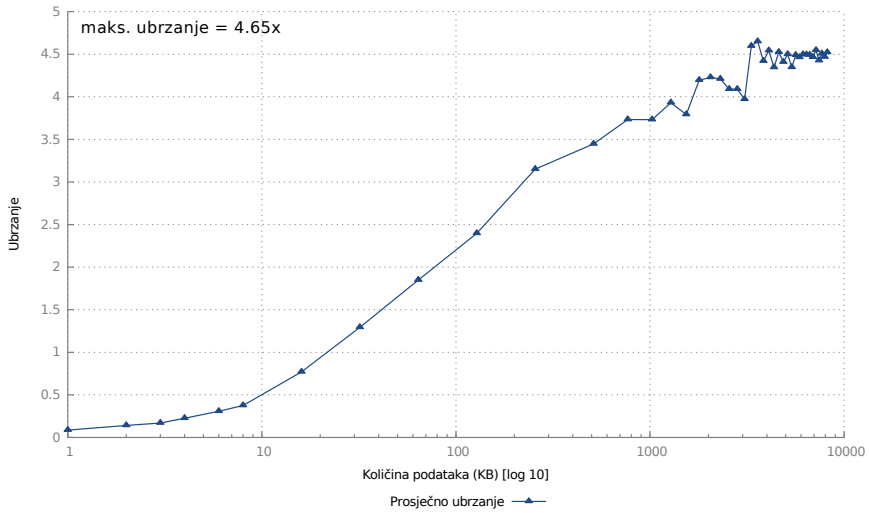
4.3.2. Rezultati

Slično kao i prilikom usporedbe ubrzanja OpenCL implementacije DES algoritma, i kod AES implementacije korištena slijedna implementacija algoritma unutar Java Cipher paketa. Korišteno je i isto testno računalo (tablica 4.1) i grafička kartica (tablica 4.2). U grafovima je mjerena propusnost (izražena u megabajtima u sekundi) i prosječno ubrzanje dekriptiranja. Mjerenja su provedena dekriptirajući blokove podataka veličine 256 kilobajta do 8 MB megabajta, u koracima od 256 kilobajta.

Grafovi 4.16 i 4.17 prikazuju rezultate dekriptiranja korištenjem ECB načina, uz 256 bitni ključ (14 krugova).

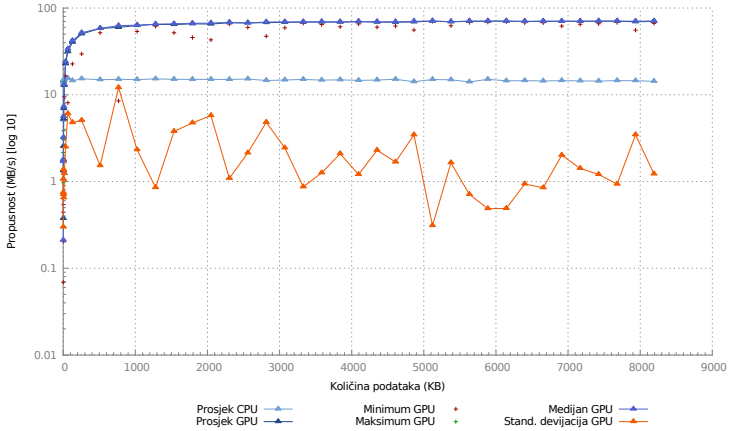


Slika 4.16: Propusnost, AES dekriptiranje, 256 bitni ključ, ECB način

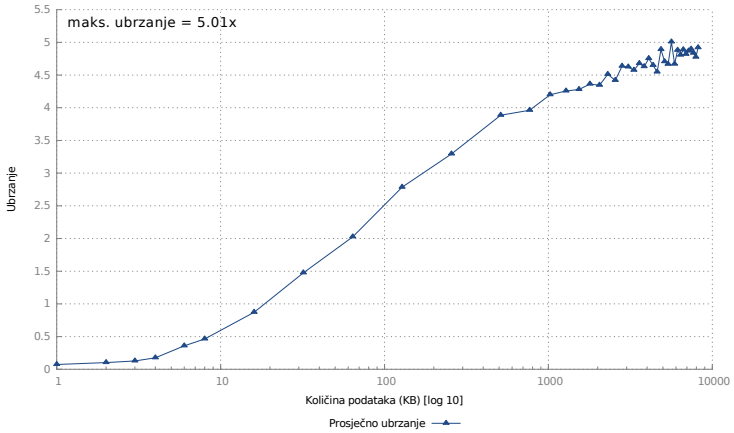


Slika 4.17: Prosječno ubrzanje, AES dekriptiranje, 256 bitni ključ, ECB način

Grafovi 4.18 i 4.19 prikazuju rezultate dekriptiranja korištenjem CBC načina, uz 256 bitni ključ (14 krugova).

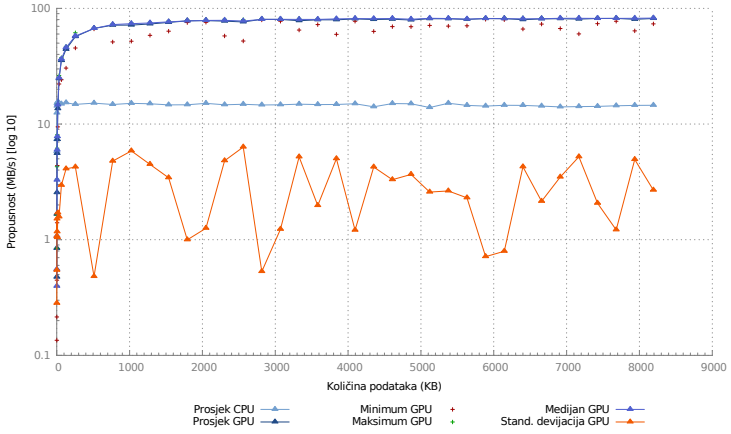


Slika 4.18: Propusnost, AES dekriptiranje, 256 bitni ključ, CBC način

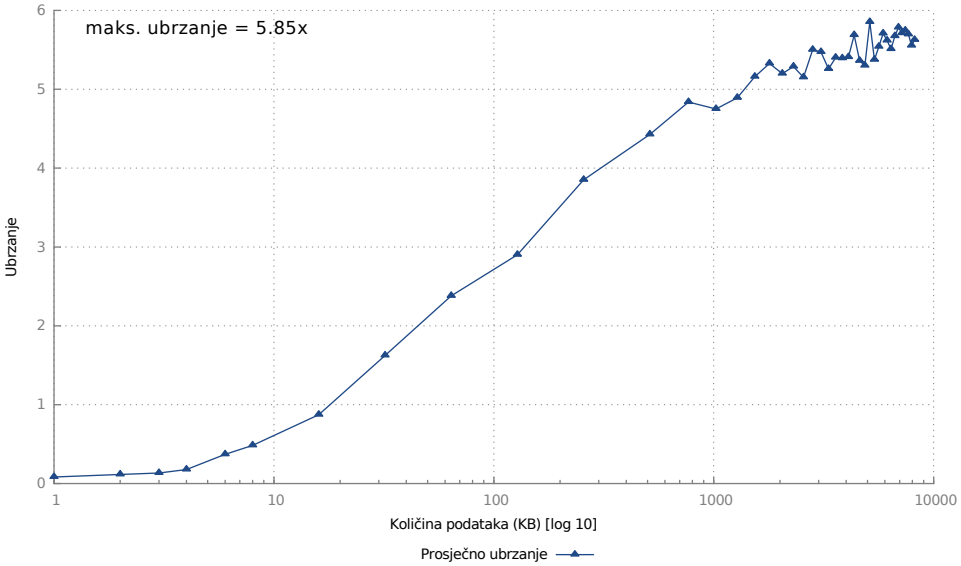


Slika 4.19: Prosječno ubrzanje, AES dekriptiranje, 256 bitni ključ, CBC način

Grafovi 4.20 i 4.21 prikazuju rezultate dekriptiranja korištenjem CTR načina, uz 256 bitni ključ (14 krugova).



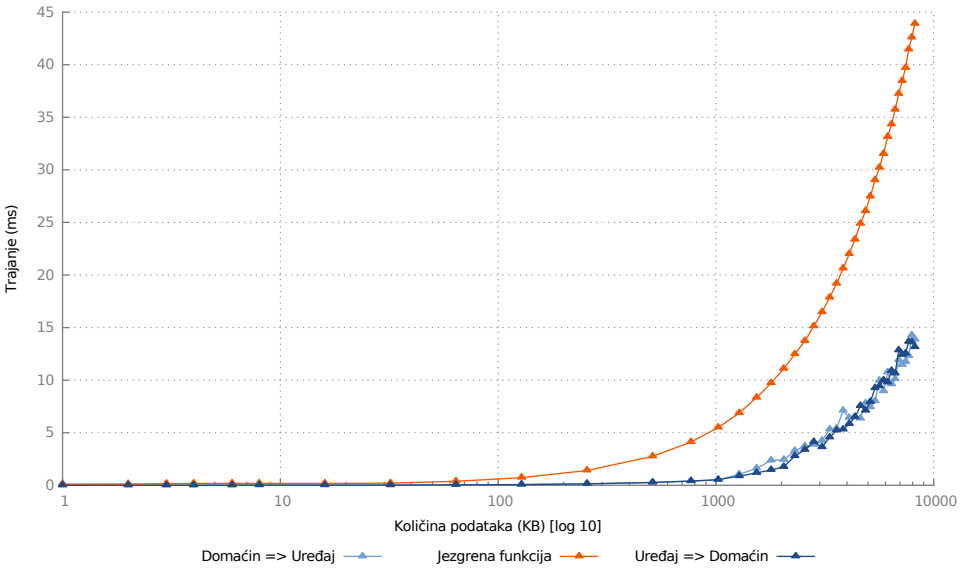
Slika 4.20: Propusnost, AES dekriptiranje, 256 bitni ključ, CTR način



Slika 4.21: Prosječno ubrzanje, AES dekriptiranje, 256 bitni ključ, CTR način

Rezultati pokazuju da je OpenCL implementacija od 5 do 6 puta brža u dekriptiranju podataka. Jedan od razloga tako dobrih rezultata je manji broj pristupa S-tablicama (a time i konstantnoj memoriji) tokom izvođenja algoritma. Drugi razlog je mogućnost efikasne implementacije algoritma jer koristi samo operacije na razini jednog okteta (za što se može koristiti tip podataka `uchar`).

Graf 4.22 prikazuje rezultate trajanja izvođenja po komponentama (CTR način).



Slika 4.22: Trajanje dekriptiranja po komponentama, CTR način (AES)

5. Evolucijski algoritmi

Evolucijski algoritmi su podgrana algoritama evolucijskog računanja koji, korištenjem procesa opisanih Darwinovom teorijom evolucije, pokušavaju riješiti kompleksne optimizacijske probleme. Unutar evolucijskog algoritma više umjetnih jedinki (engl. *artificial inidividual*) pretražuje prostor problema. Jedinke se konstantno međusobno natječu u nalaženju optimalnih područja (ili lokalnih optimuma) unutar prostora pretraživanja. Ideja evolucijskih algoritama je da će jedna od tih jedinki naposljetku pronaći najbolje rješenje (ili globalni optimum) zadanog optimizacijskog problema. Jedinke su uobičajeno prikazane kao numerički vektori fiksne duljine. Svaka jedinka predstavlja jedno moguće rješenje zadanog problema, te evolucijski algoritmi upravljaju populacijama takvih jedinki.

Općeniti evolucijski algoritam počinje sa inicijalnom populacijom fiksne veličine μ koja se sastoji slučajno stvorenih jedinki. Svakoj jedinki unutar populacije zatim je dodijeljena mjera dobrote, koristeći funkciju dobrote. Funkcija dobrote određuje koliko je kvalitetno rješenje zadanog problema koje predstavlja jedinka. Jedinke unutar populacije s višom mjerom dobrote predstavljaju bolja rješenja zadanog problema, dok jedinke s nižom mjerom dobrote predstavljaju lošija rješenja. Nakon ove inicijalne faze započinje iterativni proces koji čini jezgru evolucijskog algoritma.

Mehanizam rada evolucijskih algoritama inspiriran je biološkim procesima (operatorima) poput razmnožavanja, migracije, selekcije, križanja i mutacije. Korištenjem operatora selekcije i križanja, μ jedinki unutar populacije stvara λ potomaka. Potomcima su pridružene mjere dobrote, te se stvara nova populacija veličine μ koja se sastoji od najboljih jedinki iz populacije roditelja i potomaka. Tako stvorena populacija zamjenjuje prethodnu populaciju, te se ciklus iterativno ponavlja.

U evolucijske algoritme danas uobičajeno ubrajamo evolucijske strategije, evolucijsko programiranje, genetski algoritam te genetsko programiranje [4]. U većini praktičnih primjena evolucijskih algoritama, računalna zahtjevnost njihovog izvođenja zna predstavljati prepreku pri njihovom korištenju. Računalna zahtjevnost prvenstveno proizlazi iz potrebe

za velikim brojem izračuna funkcije dobrote. U ovom poglavlju će se na primjeru genetskog algoritma (i nekoliko njegovih hibridnih inačica) pokušati prikazati postupke kojima se može ubrzati izvođenje računalno zahtjevnijih elementa takvih algoritama koristeći standard OpenCL.

5.1. Genetski algoritam

Genetski algoritam svoju funkcionalnost temelji na Darwinovoj teorije [4] evolucije vrsta koja se temelji na 5 pretpostavki:

- **Plodnost vrsta** - Potomaka uvijek ima više nego je potrebno.
- **Veličina populacije** - Populacija je uvijek približno stalne veličine.
- **Količina hrane** - Količina hrane je ograničena.
- **Varijacija jedinki** - Vrste koje se seksualno razmnožavaju nemaju identičnih jedinki nego postoje varijacije.
- **Nasljeđe** - Najveći dio varijacije se prenosi nasljeđem

Uzimajući u obzir ove pretpostavke jasno je da sve jedinke neće moći preživjeti jer je hrana ograničeni resurs. Opstati će samo najjače jedinke s najboljim karakteristikama koje će se zatim razmnožavati te prenijeti svoje karakteristike na potomke. Preslikavanjem ovog obrasca ponašanja na optimizacijske probleme dobivamo učinkovit način rješavanja problema čije bi rješavanje klasičnim metodama bilo dugotrajno. Postoje dva načina izvođenja algoritma:

- **Eliminacijski (engl. *steady-state*)**
- **Generacijski**

U nastavku (algoritam 5.1) je prikazana te objašnjena generacijska izvedba algoritma. Generacijski algoritam u svakoj iteraciji stvara novu populaciju, u odnosu na eliminacijski algoritam koji provodi evoluciju nad stalnom populacijom jedinki. Generacijski genetski algoritam iz trenutne populacije operatorima selekcije, križanja i mutacije gradi novu pomoćnu populaciju sastavljenu isključivo od potomaka. Kada se izgradi nova populacija koja veličinom odgovara staroj, stara populacija roditelja se odbacuje, a novoizgrađena populacija potomaka postaje trenutna [4].

Moguće su i izvedbe kod kojih se gradi populacija potomaka koja je nekoliko puta veća od populacije roditelja te se iz te populacije odabiru jedinke koje će ići u sljedeću generaciju. U nastavku je objašnjeno nekoliko načina prikaza rješenja te nekoliko različitih varijacija operatora selekcije, križanja i mutacije koji se koriste u genetskim algoritmima.

Algoritam 5.1 Generacijski genetski algoritam

$P = stvori\_inicijalnu\_populaciju(VEL\_POP)$
while $nije\_zadovoljen\_uvjet\_zaustavljanja$ **do**
  $evaluiraj(P)$
  $nova\_populacija\ P' = \emptyset$
  while $velicina(P') < VEL\_POP$ **do**
    $odaberi\ R1\ i\ R2\ iz\ P$
    $\{D1,\ D2\} = krizaj(R1,\ R2)$
    $mutiraj\ D1,\ mutiraj\ D2$
    $dodaj\ D1\ i\ D2\ u\ P'$
  end while
  $P = P'$
end while

Prikaz rješenja

Postoji više različitih načina prikaza rješenja u evolucijskim algoritmima (niz bitova, vektor brojeva, permutacije, stabla, ... itd.). U ovom poglavlju korištena su dva načina:

Prikaz nizom bitova (engl. *bit string genotype*) je najjednostavniji prikaz rješenja. Svako rješenje je predstavljeno kao niz nula i jedinica fiksne duljine.

Pritom se niz bitova može tretirati kao prirodni način predstavljanja rješenja. Primjerice, neka je s O označen skup objekata $\{o_1, o_2, \ldots o_n\}$. Funkcija $f(o)$ za svaki podskup $o \in O$ vraća mjeru kvalitete tog podskupa. Zadatak algoritma je naći optimalni podskup skupa O tj. podskup o za koji je vrijednost funkcije $f(o)$ maksimalna. Rješenje ovog problema se može prikazati nizom od n bitova, pri čemu nula na poziciji i označava da objekt o_i nije uključen u podskup, a jedinica na poziciji i bi označavala da je objekt o_i uključen u podskup [4].

Ovaj prikaz se može koristiti i u numeričkim problemima jedne ili više varijabli. U tom slučaju treba definirati postupak **dekodiranja** koji će niz bitova prevesti u realne brojeve.

Prikaz poljem brojeva (engl. *floating point genotype*) je praktičan kod problema kontinuirane optimizacije funkcija više varijabli. Ukoliko funkcija $f(x_1, x_2, \ldots, x_n)$ koja se optimira ima n varijabli, tada se kao prikaz rješenja može koristiti vektor od n realnih brojeva.

Operatori križanja i mutiranja

U nastavku je opisano nekoliko najčešće korištenih operatora križanja i mutiranja:

Binarna mutacija uz zadanu vrijednost mutacije bita p_m (ukoliko se koristi prikaz nizom bitova). Mutacija se provodi na način da se invertira vrijednost bita. Može se primijeniti i na prikaz poljem brojeva, pri čemu se mutacija provodi na način da se primjerice, generira slučajan broj na zadanom mjestu u polju (uz vjerojatnost p_m).

Križanje s jednom točkom prekida slučajnim mehanizmom odabire točku prekida zajedničku za oba roditelja ($R1$ i $R2$). Zatim se stvaraju dva nova rješenja - potomci $D1$ i $D2$: prvi dobiva prvi dio genetskog materijala roditelja $R1$ i drugi dio materijala roditelja $R2$, a drugi potomak dobiva prvi dio genetskog materijala $R2$ i drugi dio materijala $R1$. Postupak je primjenjiv na oba načina prikaza rješenja opisana ranije.

Križanje s t točaka prekida je općenitiji slučaj križanja s jednom točkom prekida, gdje t može biti između 1 i $k - 1$ (gdje je k duljina zapisa rješenja). Ova metoda stvara potomke tako da izmjenično uzima segmente genetskog materijala oba roditelja.

Operator selekcije

Selekcija je postupak kojim se odabiru jedinke iz populacije (najčešće u svrhu razmnožavanja). Ovdje su opisane dvije najčešće korištena operatora selekcije:

Proporcionalna selekcija (engl. *Roulette wheel selection*) radi na način da se sve jedinke postave na kolo, pritom bolje jedinke imaju veću površinu kola. Kako prilikom vrtnje kola boljim jedinkama pripada veći dio kola, one će imati i veću vjerojatnost odabira tj. vjerojatnost odabira će biti proporcionalna dobroti jedinke.

Turnirska selekcija (engl. *Tournament selection*) je selekcija kod koje se iz populacije slučajnim odabirom izvlači N jedinki, iz tog uzorka odabire se ona jedinka koja ima najveću dobrotu. Turnirska selekcija koja iz populacije izvlači slučajnu uzorak od k rješenja naziva se k-turnirska selekcija.

Generacijska izvedba genetskog algoritma više odgovara u ovom poglavlju predloženom OpenCL paralelnom modelu. Kako u svakoj iteraciji algoritma nastaje nova populacija jedinki kojima treba evaluirati dobrotu (engl. *fitness*), proces evaluacije može se paralelizirati na način da svaka OpenCL radna jedinica evaluira dobrotu jedne jedinke. Ovaj princip će biti prikazan na problemu optimizacije kombinatoričkih funkcija koji je objašnjen u nastavku.

5.2. Optimizacija kombinatoričkih funkcija

Kombinatorička ili Booleova funkcija (ili n-arna logička operacija) je bilo koja funkcija $F : B^n \rightarrow B$, gdje je $B = \{0, 1\}$. Dakle vrijedi izraz 5.1

$$(x_1, x_2, \ldots, x_n) \in B^n \rightarrow F(x_1, x_2, \ldots, x_n) \in B \tag{5.1}$$

gdje su $x_1, x_2, \ldots, x_n$ varijable kombinatoričke funkcije. Kombinatoričke funkcije je najjednostavnije prikazati tablicom istinitosti (engl. *truth table*).

Definicija 1 *Broj svih kombinatoričkih funkcija od n varijabli iznosi* 2^{2^n}.

Primjerice, kombinatoričkih funkcija jedne varijable ima $2^{2^1} = 4$, a dane su tablicom 5.1.

Tablica 5.1: Kombinatoričke funkcije jedne varijable

x	F_1	F_2	F_3	F_4
1	0	1	0	1
0	0	0	1	1

Kombinatoričke funkcije imaju široku primjenu u simetričnoj kriptografiji. Simetrični kriptografski algoritmi se mogu podijeliti na algoritme koji rade na razini bloka podataka (engl. *block cipher*) i algoritme koji rade na razini toka podataka (engl. *stream chipher*).

Osnivač moderne teorije informacija Claude Shannon definirao je dva principa koja bi trebala osigurati siguran kriptosustav - princip **konfuzije** (engl. *confusion principle*) i princip **difuzije** (engl. *difusion principle*). Princip difuzije služi tome da se proširi utjecaj svakog bita jasnog teksta i ključa na što više bitova kriptiranog teksta. Princip konfuzije pak služi tome da vezu između ključa i kriptiranog teksta učini što složenijom, te se dobiva nelinearnim transformacijama. U algoritama koji rade na razini bloka podataka, konfuzija se dobiva iz S-tablica koje se mogu promatrati kao vektorske kombinatoričke funkcije. Kod algoritama koji rade na razini toka podataka koriste se **kombinatoričke funkcije** za postizanje nelinearnosti.

Posebno su interesantne takve kombinatoričke funkcije koje imaju dobra kriptografska svojstva. Ideja uporabe evolucijskih algoritama za pronalaženje kombinatoričkih funkcija optimalnih kriptografskih svojstava ispitana je u nekolicini radova [8] [3] [14]. Dobrotu pojedine jedinke pritom je određivala pogodnost njezinih kriptografskih svojstava. U ovom radu ispitat će se potencijalno ubrzanje procesa evaluacije populacije jedinki, ukoliko se evaluacija izvršava paralelno nad više jedinki. Kriptografska svojstva čija je kombinacija korištena u ocjeni dobrote jedinki preuzeta su iz rada [8], te su nabrojana u nastavku.

- Balansiranost (BALANS)
- Nelinearnost (NL)
- Korelacijski imunitet (KI)
- Walshov spektar (WALSH)
- Algebarski stupanj (DEG)
- Karakteristika propagacije (KP)
- Suma kvadrata indikator (SKI)
- Apsolutni indikator (AI)

5.2.1. Opis implementacije

Za prikaz jedinke korišteno je polje cijelih brojeva koji mogu poprimiti vrijednost iz skupa $\{0, 1\}$, te predstavlja tablicu istinitosti za pojedinu jedinku. Za kombinatoričku funkciju od n varijabli duljina zapisa jedinke je 2^n bitova. Prilikom slanja populacije na evaluaciju koristeći OpenCL, prvo je potrebno spojiti zapise jedinki (polja cijelih brojeva) u jedan veliki zapis. Radne jedinice će zatim koristeći svoj globalni identifikator moći odrediti koji dio zapisa se odnosi na jedinku koju trebaju obraditi. Prototip jezgrene funkcije koja provodi evaluaciju jedne jedinke prikazana je pod 5.2.

```
__kernel void boolFitness(
    __global const int *input,
    __global int *output,
    __constant int *fitnessMask,
    const int populationSize
)
```

Tekst programa 5.2: Prototip jezgrene funkcije evaluacije kombinatoričkih funkcija

Argument `input` je cjelobrojno polje u globalnoj memoriji duljine $2^n \times brojJedinki$, gdje je n broj varijabli kombinatoričke funkcije koju optimiramo. Argument `output` je cjelobrojno polje duljine $8 \times brojJedinki$ u koje će biti zapisane konačne vrijednosti dobrote pojedine jedinke po svojstvima (ukupno je 8 svojstava). Funkcijske implementacije pojedinih svojstava preuzete su iz rada [8]. Argument `fitnessMask` je cjelobrojno polje u konstantnoj memoriji duljine 8 čiji su elementi iz skupa $\{0, 1\}$. Vrijednosti elemenata polja odlučuje hoće li pojedino svojstvo biti uključeno u konačni izračun dobrote. Tablica 5.2 prikazuje koji element polja predstavlja pojedino svojstvo.

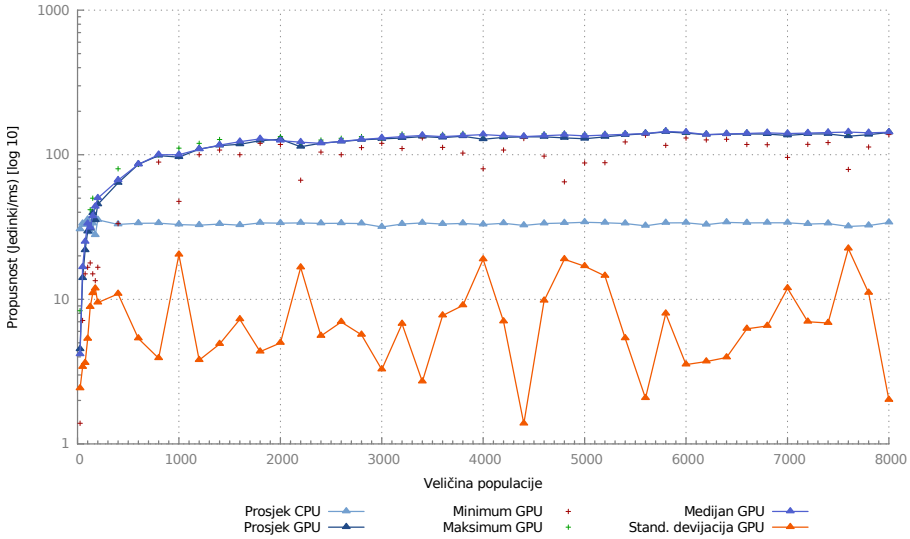
Indeks polja	0	1	2	3	4	5	6	7
Svojstvo	BALANS	NL	KI	WALSH	DEG	KP	SKI	AI

Tablica 5.2: Indeksi svojstava

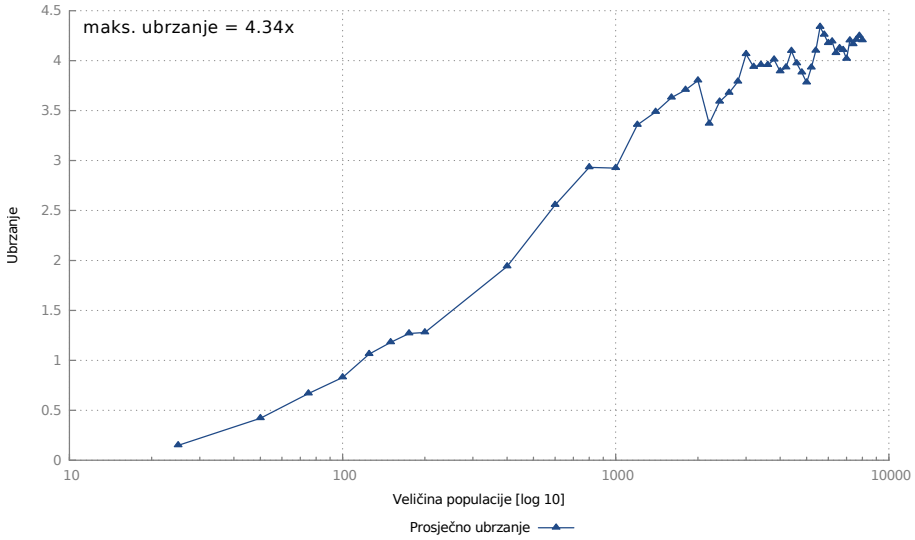
5.2.2. Rezultati

Rezultati su dobiveni na istom testnom računalu i grafičkoj kartici navedenim u prethodnim poglavljima (tablice 4.1 i 4.2). Prilikom testiranja odabrano je nekoliko različitih kombinacija svojstava koji su uključeni u konačni izračun dobrote. Mjerenja su provedene nad slučajno generiranim populacijama veličine od 25 do 8000 jedinki. Prilikom mjerenje je korištena maksimalna dostupna veličina radne grupe od 256 radnih jedinica.

Grafovi 5.1 i 5.2 prikazuju rezultate propusnosti i prosječnog ubrzanja korištenjem funkcije dobrote **FIT1 = BALANS + NL**.

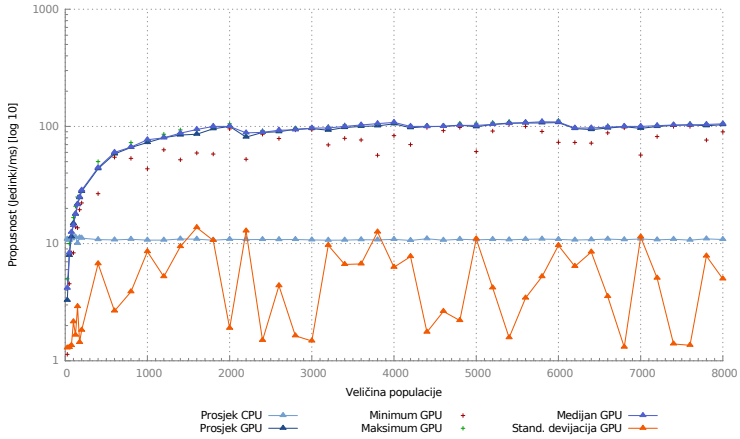


Slika 5.1: Propusnost, evaluacija kombinatoričkih funkcija, FIT1

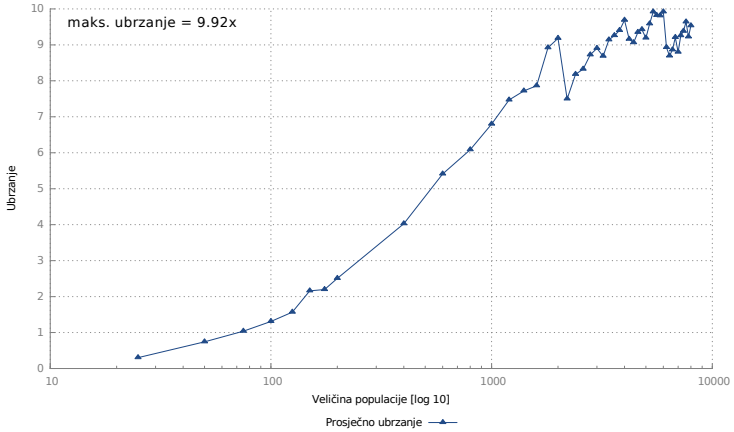


Slika 5.2: Prosječno ubrzanje, evaluacija kombinatoričkih funkcija, FIT1

Grafovi 5.3 i 5.4 prikazuju rezultate korištenjem funkcije dobrote **FIT2 = FIT1 + KI + WALSH + DEG**.

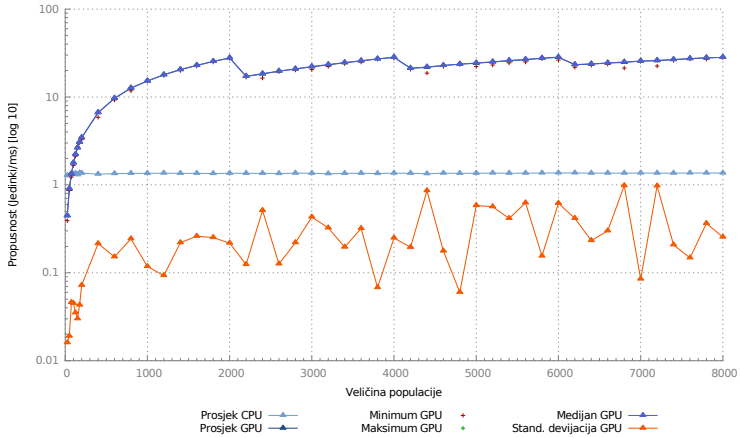


Slika 5.3: Propusnost, evaluacija kombinatoričkih funkcija, FIT2

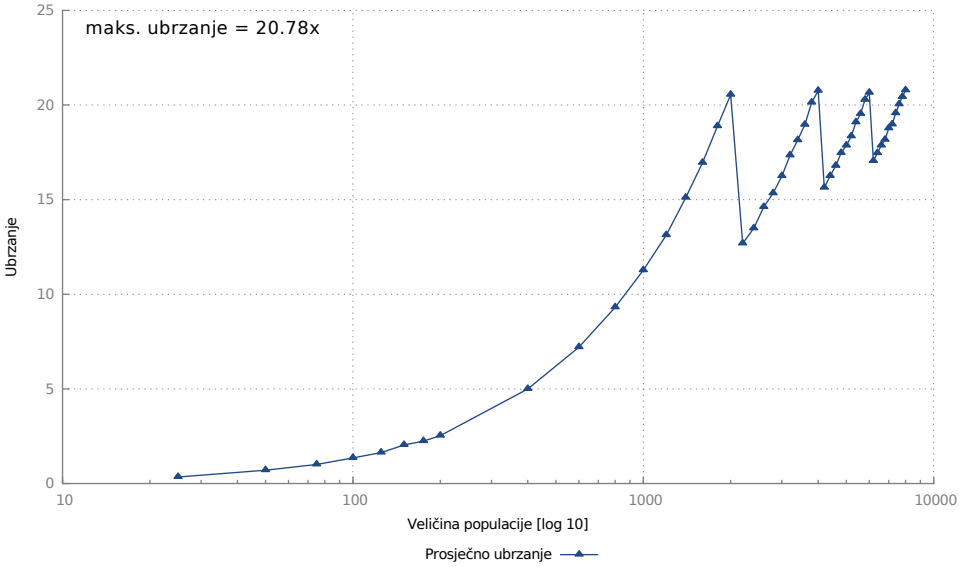


Slika 5.4: Prosječno ubrzanje, evaluacija kombinatoričkih funkcija, FIT2

Grafovi 5.5 i 5.6 prikazuju rezultate korištenjem funkcije dobrote **FIT3 = FIT2 + KP + SKI + AI**.



Slika 5.5: Propusnost, evaluacija kombinatoričkih funkcija, FIT3



Slika 5.6: Prosječno ubrzanje, evaluacija kombinatoričkih funkcija, FIT3

Rezultati pokazuju da ubrzanje raste kako funkcija dobrote postaje složenija (dodavanjem dodatnih svojstava). Kako se povećava udio vremena koji se troši na samu jezgrenu funkciju, tako i slabi udio vremena potrošenog na prijenos podataka između domaćina i OpenCL uređaja. Tablice 5.3 i 5.4 prikazuju trajanje pojedinih komponenti prilikom korištenja jednostavnije **FIT1** funkcije dobrote te složenije **FIT3** funkcije dobrote.

Tablica 5.3: Komponente - FIT1

Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	12.84
Jezgrena funkcija	12.03
Uređaj => Domaćin	1.23
Ukupno	26.1

Tablica 5.4: Komponente - FIT3

Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	12.77
Jezgrena funkcija	235.27
Uređaj => Domaćin	1.35
Ukupno	249.39

Funkcija cilja FIT3 ima najveće prosječno ubrzanje od 20.8 puta, ali i najduže trajanje izvođenja jezgrene funkcije.

5.3. Hibridni Hooke-Jeeves genetski algoritam

Hibridni Hooke-Jeeves genetski algoritam je algoritam koji pokušava unaprijediti brzinu konvergencije genetskog algoritma kombinirajući ga s deterministčkim optimizacijskim Hooke-Jeeves algoritmom (algoritam 5.3) [11].

Algoritam 5.3 Hibridni Hooke-Jeeves genetski algoritam, HHJGA

function $HHJGA(n, \Delta x_0, \varepsilon, p_m)$
  $P \leftarrow inicijaliziraj\_populaciju(n)$
  $P^B \leftarrow P$ ▷ Populacija baznih rješenja
  $\Delta x \leftarrow [\Delta x_0, \ldots, \Delta x_0]$
  while !uvjet_zaustavljanja() **do**
    for $i = 1$ **to** $velicina(P)$ **do**
      $x \leftarrow P_i,\ x^B \leftarrow {P^B}_i$
      $x \leftarrow istrazi(x, \Delta x_i)$
      if $dobrota(x) > dobrota(x^B)$ **then**
        $y \leftarrow x$
        for $j = 1$ **to** $dimenzije(x)$ **do**
          $x_j \leftarrow 2x_j - {x^B}_j$
        end for
        $x^B \leftarrow y$
      else
        $\Delta x_i \leftarrow \Delta x_i/2,\ x \leftarrow x^B$
        if $\Delta x_i < \varepsilon$ **and** $dobrota(najbolji(P)) \geq dobrota(x)$ **then**
          $a \leftarrow selekcija(P),\ b \leftarrow selekcija(P)$
          $x \leftarrow krizanje(a, b),\ x \leftarrow mutacija(x, p_m)$
          $\Delta x_i \leftarrow \Delta x_0,\ x^B \leftarrow x$
        end if
      end if
      $P_i \leftarrow x,\ {P^B}_i \leftarrow x^B$
    end for
  end while
  return $najbolji(P)$
end function

Prilikom izvođenja algoritma od posebne je važnosti funkcija `istrazi` koja obavlja lokalno pretraživanje prostora rješenja (algoritam 5.4).

Algoritam 5.4 Funkcija `istrazi` Hooke-Jeeves algoritma

```
function istrazi(x^P, Δx)
    x ← x^P
    for i = 1 to dimenzije(x) do
        P ← funkcija_cilja(x)
        x_i = x_i + Δx                          ▷ Pozitivan pomak za Δx
        N ← funkcija_cilja(x)
        if N > P then                           ▷ Pozitivan pomak nije donio poboljšanje
            x_i = x_i − 2 × Δx                  ▷ Negativan pomak za Δx
            N ← funkcija_cilja(x)
            if N > P then                       ▷ Negativni pomak nije donio poboljšanje
                x_i = x_i + Δx                  ▷ Povratak na početnu vrijednost
            end if
        end if
    end for
    return x
end function
```

Algoritam radi s populacijom jedinki (rješenja) na način da provodi jednu iteraciju Hooke-Jeeves algoritma nad svakom jedinkom, a u trenutku kada jedinka više ne može napredovati primjenjuje operatore genetskog algoritma. U početku se stvara inicijalna populacija jedinki, jedinke su prikazane kao polje realnih brojeva gdje svaki element polja predstavlja jednu dimenziju funkcije cilja. Iteracija algoritma započinje lokalnim pretraživanjem prostora rješenja po svim dimenzijama uz pomak Δx (funkcija `istrazi`).

Ukoliko je nađeno rješenje koje je bolje od baznog rješenja x^B (bazna rješenja za svaku jedinku se pamte u populaciji P^B), bazno rješenje se prebacuje preko pronađenog i smanjuje se pomak Δx. Kada se pomak dovoljno smanji ($\Delta x < \varepsilon$), rješenje se zamjenjuje potomkom nastalim križanjem i mutacijom dviju roditeljskih jedinki odabranih operatorom selekcije. Operatori križanja i mutacije se ponašaju na isti način kao i u običnom genetskom algoritmu.

5.3.1. Opis implementacije

Iz pseudokoda HHJGA je vidljivo da je računski najzahtjevnija funkcija `istrazi`, koja u najgorem slučaju može imati i do $3 \times DIM$ poziva funkcije cilja (gdje je DIM broj dimenzija funkcije cilja). Stoga će se u ovoj implementaciji paralelizirati funkcija `istrazi`

tako da se paralelno izvodi nad svim jedinkama unutar populacije. Ostatak algoritma ostaje slijedan, iz razloga što OpenCL nema podršku za generiranje slučajnih brojeva potrebnih za operatore mutacije i križanja. Na ovaj način će se prvo paralelno provesti lokalno pretraživanje nad svim jedinkama te se zatim nad njima pojedinačno provodi jedna slijedna iteracija algoritma. Prototip jezgrene funkcije prikazan je programskim odsječkom 5.5.

```
__kernel void HJExplore(
    __global const float *population,
    __global const float *delta,
    __global float *resultPopulation,
    __global float *resultFitness,
    const int populationSize
)
```

Tekst programa 5.5: Prototip jezgrene funkcije Hooke-Jeeves istraži

Argument `population` je polje realnih brojeva duljine $DIM \times brojJedinki$, gdje svako potpolje duljine DIM predstavlja jednu jedinku. Za svaku jedinku definiran je i trenutni pomak (Δx), te se nalazi u posebnoj populaciji pomaka duljine $brojJedinki$ (argument `delta`). Argument `resultPopulation` je polje duljine $DIM \times brojJedinki$ te će služiti za pohranu jedinki koje su nastale kao rezultat lokalnog pretraživanja. Polje `resultFitness` služi za pohranu vrijednosti dobrote novonastalih jedinki kako se ne bi morale računati na domaćinu.

5.3.2. Rezultati

Testiranja su provedena nad jedinkama koje imaju 30 dimenzija (korištene funkcije cilja imaju 30 varijabli). Prilikom testiranja korištene su funkcije cilja preuzete iz COCO (engl. *Comparing Continuous Optimisers*) biblioteke kontinuiranih funkcija. Sve funkcije unutar ove biblioteke su višedimenzionalne ($\Re^D$), te je domena pretraživanja u intervalu $[-5, 5]^D$.

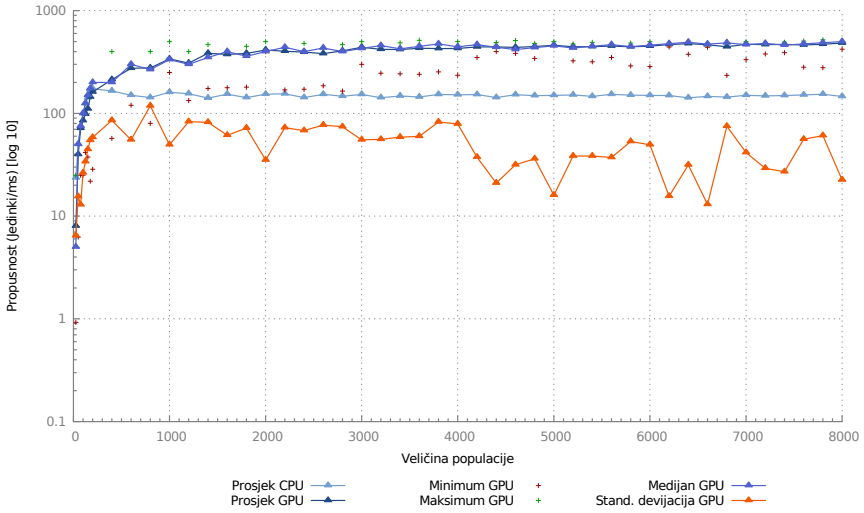
Funkcija	**Definicija**
sferna funkcija	$f(\mathbf{x}) = \|\|\mathbf{x}\|\|^2$
Rastrigin funkcija	$f(\mathbf{x}) = 10\Big(D - \sum_{i=1}^{D} cos(2\pi x_i)\Big) + \|\|\mathbf{x}\|\|^2$
Weierstrass funkcija	$f(\mathbf{x}) = 10\Big(\frac{1}{D}\sum_{i=1}^{D}\sum_{k=0}^{11}\frac{1}{2^k}cos(2\pi 3^k(x_i+\frac{1}{2})) - f_0\Big)^3$ $f_0 = \sum_{k=0}^{11}\frac{1}{2^k}cos(2\pi 3^k\frac{1}{2})$

Tablica 5.5: Ispitne funkcije (COCO)

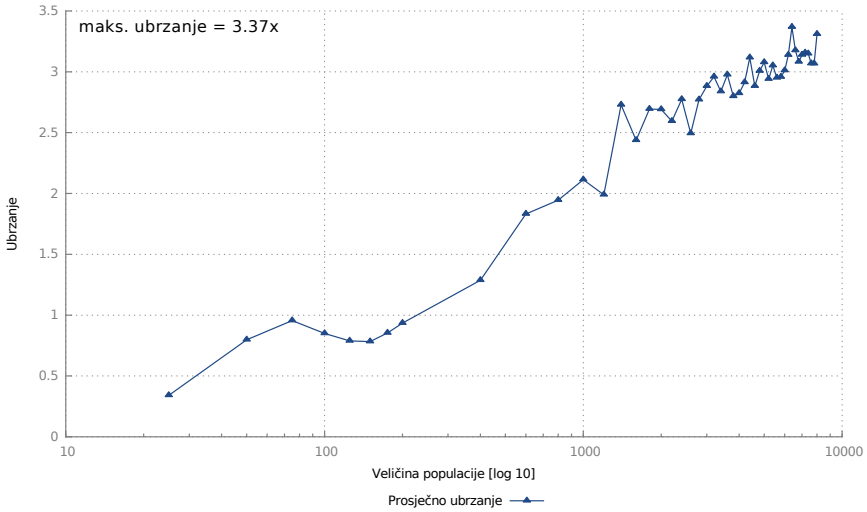
COCO biblioteka je danas često korištena prilikom ocjene kvalitete optimizacijskih algoritama. Kako je cilj ovog rada pokazati da je OpenCL izvedba pogodna za ubrzavanje takvih algoritama, koristit će se funkcije cilja relevantne za to područje. Za testiranje su odabrane 3 funkcije različite računske složenost (tablica 5.5).

Na grafovima os apscisa označava veličinu populacije nad kojom je provedeno lokalno pretraživanje. Mjerena je propusnost (broj lokalnih pretraživanja obavljenih u jednoj milisekundi) i prosječno ubrzanje lokalnog pretraživanja. Veličina radne grupe je postavljena na 128 radnih jedinica.

Grafovi 5.7 i 5.8 prikazuju propusnost i prosječno ubrzanje lokalnog pretraživanja korištenjem sferne funkcije.

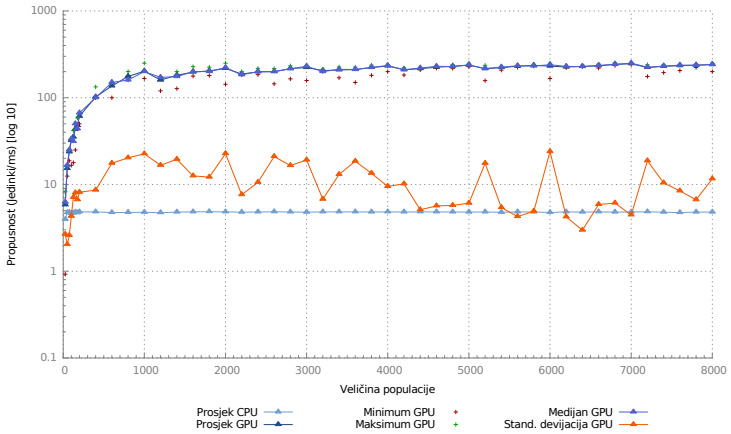


Slika 5.7: Propusnost, Hooke-Jeeves istraži, sferna funkcija, 30 varijabli

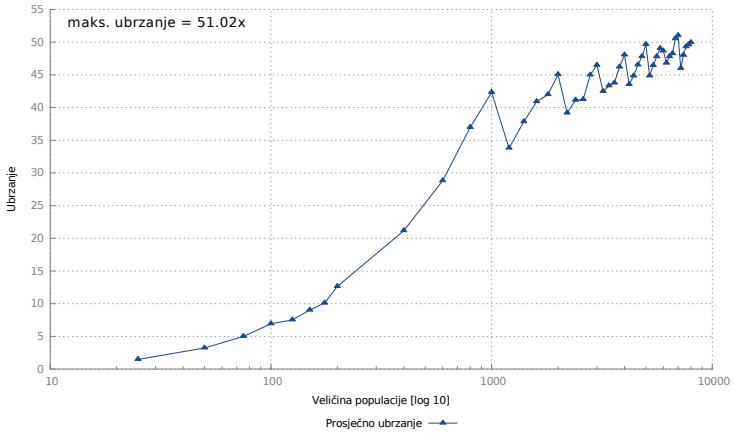


Slika 5.8: Prosječno ubrzanje, Hooke-Jeeves istraži, sferna funkcija, 30 varijabli

Grafovi 5.9 i 5.10 prikazuju propusnost i prosječno ubrzanje lokalnog pretraživanja korištenjem Rastrigin funkcije.

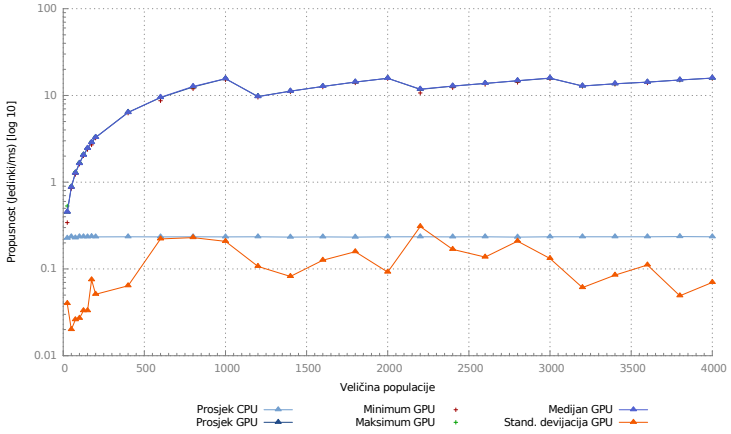


Slika 5.9: Propusnost, Hooke-Jeeves istraži, Rastrigin funkcija, 30 varijabli

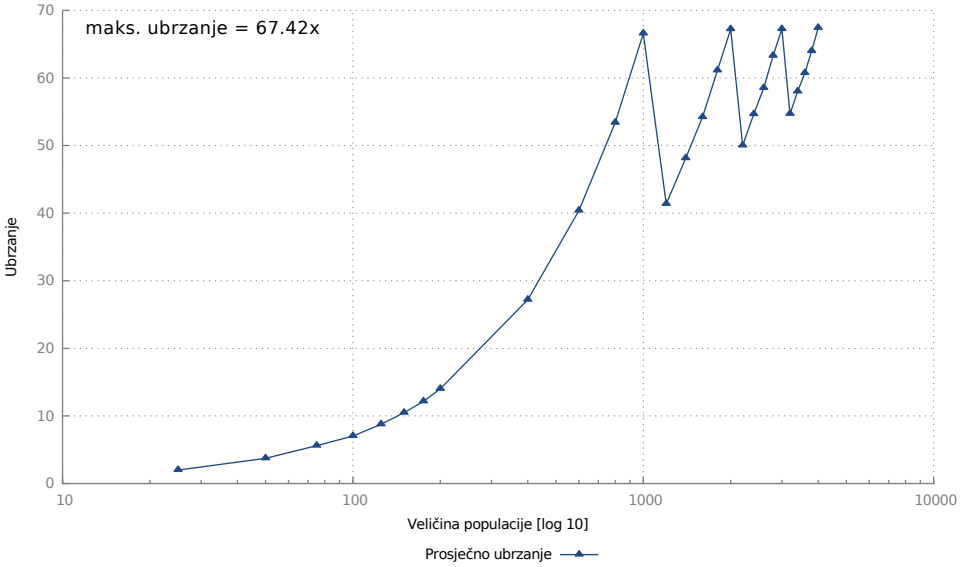


Slika 5.10: Prosječno ubrzanje, Hooke-Jeeves istraži, Rastrigin funkcija, 30 varijabli

Grafovi 5.11 i 5.12 prikazuju propusnost i prosječno ubrzanje lokalnog pretraživanja korištenjem Weierstrass funkcije.



Slika 5.11: Propusnost, Hooke-Jeeves istraži, Weierstrass funkcija, 30 varijabli



Slika 5.12: Prosječno ubrzanje, Hooke-Jeeves istraži, Weierstrass funkcija, 30 varijabli

Tablice 5.6 i 5.7 prikazuju raspored vremena izvođenja po komponentama za sfernu i Rastrigin funkciju, za veličinu populacije od 8000 jedinki.

Tablica 5.6: Hooke-Jeeves istraži komponente sferna funkcija

Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	0.495
Jezgrena funkcija	1.81
Uređaj => Domaćin	0.541
Ukupno	2.84

Tablica 5.7: Hooke-Jeeves istraži komponente Rastrigin funkcija

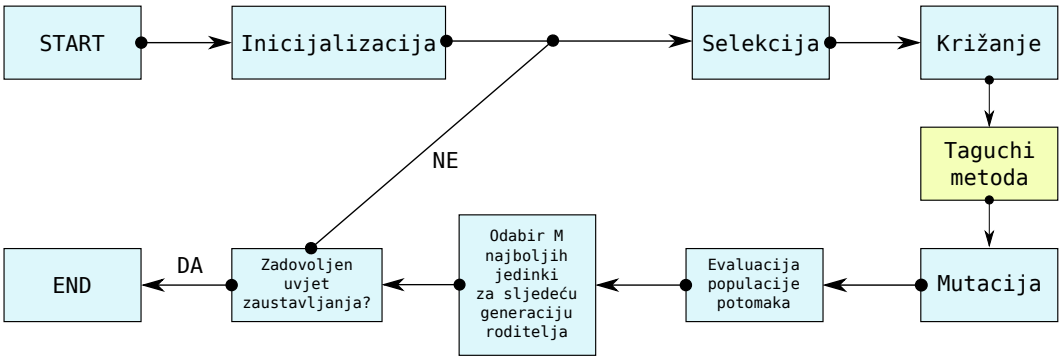
Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	0.505
Jezgrena funkcija	18.16
Uređaj => Domaćin	0.511
Ukupno	19.176

5.4. Hibridni Taguchi genetski algoritam

Kako su algoritmi koji rješavaju probleme globalne optimizacije od rastuće važnosti, mnoga istraživanja su fokusirana na razna unaprjeđenja postojećih algoritama poput genetskog algoritma. Hibridni Taguchi genetski algoritam ili HTGA to pokušava izvesti kombinirajući genetski algoritam sa Taguchi metodom [20].

Taguchi metoda je statistička metoda koji je razvio dr. Genichi Taguchi kako bi poboljšao kvalitetu proizvodnih procesa. Ova metoda je predstavnik DoE (engl. *Design of Experiments*) postupaka , te se već dugi niz godina uspješno primjenjuje u problemima osiguranja kvalitete proizvodnih procesa.

HTGA opisan u [20] postavlja Taguchi metodu između operatora križanja i mutacije kako je prikazano slikom 5.13. Proizvodni proces koji u genetskom algoritmu želimo optimirati je proces generiranja potomaka križanjem dviju jedinki. Metoda određuje od kojeg roditelja uzeti koji genetski materijal u cilju stvaranja optimalnog potomka.



Slika 5.13: Hibridni Taguchi genetski algoritam (HTGA)

Dva ključna alata korištena u Taguchi metodi su:

Ortogonalna polja (engl. *orthogonal arrays*) kojima se istovremeno može promatrati više procesnih parametara. U Taguchi metodi su korištena dvodimenzionalna ortogonalna polja (ortogonalne matrice) čija je općenita oznaka (5.2):

$$L_n(2^{n-1}) \tag{5.2}$$

gdje je

- $\mathbf{n = 2^k}$ broj eksperimenata, k je pozitivan cijeli broj veći od 1
- **2** broj razina za svaki faktor, broj razina odgovara broju roditelja
- **n-1** broj stupaca dvodimenzionalnog ortogonalnog polja

Uobičajeno korištena dvodimenzionalna ortogonalna polja su $L_4(2^3)$, $L_8(2^7)$, $L_{16}(2^{15})$ i $L_{32}(2^{31})$. U nastavku (tablica 5.8) je prikazano ortogonalno polje $L_8(2^7)$:

Tablica 5.8: Dvodimenzionalno ortogonalno polje $L_8(2^7)$

A	B	C	D	E	F	G
1	1	1	1	1	1	1
1	1	1	2	2	2	2
1	2	2	1	1	2	2
1	2	2	2	2	1	1
2	1	2	1	2	1	2
2	1	2	2	1	2	1
2	2	1	1	2	2	1
2	2	1	2	1	1	2

Dvodimenzionalno ortogonalno polje $L_8(2^7)$ može se koristiti prilikom optimizacije kontinuiranih funkcija sa do 7 dimenzija (varijabli). U kontekstu kontinuirane optimizacije svaki red matrice predstavlja jedan eksperiment čiji su pojedini faktori (A, B, ... G) uzeti od roditelja $R1$ i $R2$. Pri tome svaki stupac označava od kojeg roditelja uzimamo genetski materijal za pojedini faktor. Primjerice, treći eksperiment generirao bi se na način da je prvi faktor uzet od $R1$, drugi faktor od $R2$, treći faktor od $R2$, ... itd. U prvom koraku Taguchi metode potrebno je generirati svih n eksperimenata te se zatim ocjenjuje njihova kvaliteta.

Odnos signala i šuma (engl. *SNR, Signal to noise ratio*) je metoda kojim se mjeri kvaliteta provedenih eksperimenta. SNR se računa kao (izraz 5.3)

$$\eta_i = (f(x_1, x_2, \dots x_n))^2 \tag{5.3}$$

ukoliko se radi o maksimizaciji, ili ukoliko se radi o minimizaciji kao (izraz 5.4)

$$\eta_i = \frac{1}{(f(x_1, x_2, \dots x_n))^2} \tag{5.4}$$

gdje $f(x_1, x_2, \dots x_n)$ predstavlja višedimenzionalnu funkciju cilja koju optimiramo, a varijable $x_1, x_2, \dots x_n$ predstavljaju vrijednosti faktora i-tog eksperimenta.

Cjelokupni postupak će bit prikazan na primjeru minimizacije funkcije 5.5:

$$f(x) = \sum_{i=1}^{7} {x_i}^2 \tag{5.5}$$

Operatorom selekcije odabrano je dvoje roditelja ($R1$ i $R2$) čiji optimalni potomak treba proizvesti Taguchi metodom (5.9)

Tablica 5.9: Primjer: Roditelji $R1$ i $R2$ odabrani selekcijom

Faktor	A	B	C	D	E	F	G
Razina 1 ($R1$)	1.0	1.0	1.0	1.0	0.0	0.0	0.0
Razina 2 ($R2$)	0.0	0.0	0.0	0.0	1.0	1.0	1.0

Koristeći ortogonalnu matricu $L_8(2^7)$ prikazanu tablicom 5.8 generira se 8 eksperimenata i računa njihov SNR, rezultati su prikazani tablicom 5.10.

Tablica 5.10: Primjer: Eksperimenti i pripadajuće vrijednosti SNR-a

Eksperiment	A	B	C	D	E	F	G	Funkcija cilja	η_i
1	1.0	1.0	1.0	1.0	0.0	0.0	0.0	4.0	$\frac{1}{16.0}$
2	1.0	1.0	1.0	0.0	1.0	1.0	1.0	6.0	$\frac{1}{36.0}$
3	1.0	0.0	0.0	1.0	0.0	1.0	1.0	4.0	$\frac{1}{16.0}$
4	1.0	0.0	0.0	0.0	1.0	0.0	0.0	2.0	$\frac{1}{4.0}$
5	0.0	1.0	0.0	1.0	1.0	0.0	1.0	4.0	$\frac{1}{16.0}$
6	0.0	1.0	0.0	0.0	0.0	1.0	0.0	2.0	$\frac{1}{4.0}$
7	0.0	0.0	1.0	1.0	1.0	1.0	0.0	4.0	$\frac{1}{16.0}$
8	0.0	0.0	1.0	0.0	0.0	0.0	1.0	2.0	$\frac{1}{4.0}$

Zatim se računaju efekti pojedinih faktora, čiji će odnos odlučiti o tome od kojeg roditelja se uzima genetski materijal za pojedini faktor optimalnog potomka. Efekti pojedinih faktora po razinama računaju se izrazom 5.6

$$E_{fl} = \sum \eta_i \; za \; faktor \; f \; na \; razini \; l \tag{5.6}$$

Primjerice , efekti za obje razine faktora B računali bi se kao (5.7):

$$\begin{aligned} E_{B1} &= \textstyle\sum_{i=1,2,5,6} \eta_i = \frac{1}{16} + \frac{1}{36} + \frac{1}{16} + \frac{1}{4} = 0.403 \\ E_{B2} &= \textstyle\sum_{i=3,4,7,8} \eta_i = \frac{1}{16} + \frac{1}{4} + \frac{1}{16} + \frac{1}{4} = 0.625 \end{aligned} \tag{5.7}$$

Konačni rezultati izračuna efekata i optimalni potomak prikazani su tablicom 5.11

Tablica 5.11: Primjer: Vrijednosti efekata po razinama i optimalni potomak

	A	B	C	D	E	F	G
E_{f1} $(R1)$	0.403	0.403	0.403	0.25	0.625	0.625	0.625
E_{f2} $(R2)$	0.625	0.625	0.625	0.777	0.403	0.403	0.403
Optimalna razina (roditelj)	2	2	2	2	1	1	1
Optimalni potomak	0.0	0.0	0.0	0.0	0.0	0.0	0.0

Faktori optimalnog potomka se biraju na način da se uspoređuju izračuni efekata za svaku razinu (roditelja). Ukoliko je $E_{f1} > E_{f2}$ genetski materijal za faktor f će biti uzet od roditelja $R1$, a ako je $E_{f2} > E_{f1}$ bit će uzet od roditelja $R2$.

Iz primjera je vidljivo da je zbog neovisnosti obavljanja pojedinih eksperimenata ovaj postupak prirodno pogodan za paralelizaciju, što je implementirano u jednodimenzionalnoj i dvodimenzionalnoj inačici koristeći standard OpenCL.

5.4.1. Opis 1D implementacije

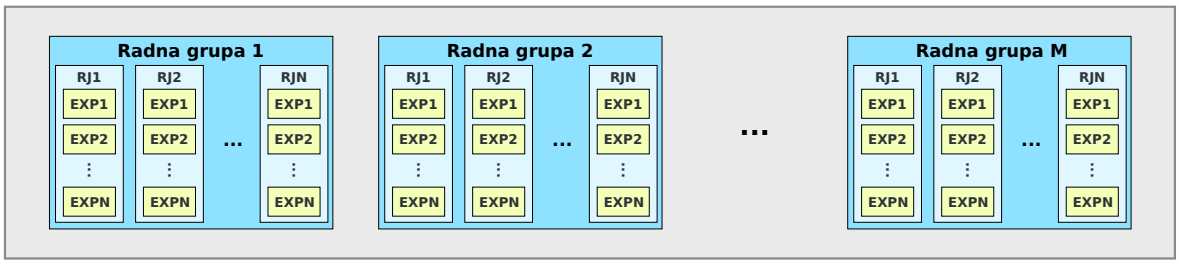
Jednodimenzionalna implementacija slična je implementacijama iz prethodnih poglavlja, te se temelji na paralelnom provođenju metode nad više roditelja. Prvo je potrebno operatorom selekcije odabrati populaciju roditelja veličine $2 \times brojPotomaka$. Roditelji i potomci su prikazani kao jedinke čiji je genotip vektor realnih brojeva veličine DIM, gdje DIM predstavlja broj dimenzija (varijabli) funkcije cilja. Provođenje metode nad dvoje roditelja će rezultirati jednim potomkom, pa treba osigurati i odgovarajući spremnik veličine $brojPotomaka$ za njihovu pohranu. Prototip OpenCL jezgrene funkcije je dan je odsječkom 5.6.

```
__kernel void TaguchiCrossover1D(
    __global const float *inputParents,
    __global float *outputChildren,
    const int crossoverNum
)
```

Tekst programa 5.6: Prototip jezgrene funkcije Taguchi križanja

Jezgrena funkcija koristi samo globalni identifikator kako bi adresirala svoj par roditelja te ih prepisala u svoju privatnu memoriju za daljnju obradu. Svaka radna jedinica koristeći genetski materijal roditelja i ortogonalnu matricu L provodi sve njome zadane eksperimente i proizvodi jednog optimalnog potomka. Ortogonalna matrica L smještena je u **konstantnu memoriju**, jer će sve radne jedinice unutar osnovne skupine istovremeno htjeti pristupiti

istom elementu (eksperimenti se provode slijedno, jedan za drugim), što će rezultirati operacijom prijenosa (engl. *broadcast operation*).



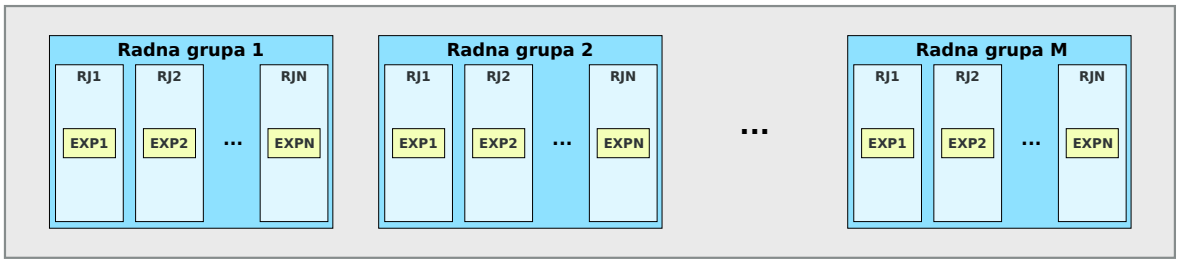
Slika 5.14: Jednodimenzionalna OpenCL inačica HTGA

5.4.2. Opis 2D implementacije

U odnosu na jednodimenzionalnu implementaciju koja samo paralelno izvodi Taguchi metodu nad različitim parovima roditelja, dvodimenzionalna inačica paralelizira i samu metodu. U dvodimenzionalnoj inačici **1 radna grupa** koja se sastoji od EXP (broj eksperimenata) radnih jedinica izvodi Taguchi metodu nad parom roditelja, pri čemu svaka radna jedinica provodi jedan eksperiment zadan ortogonalnom matricom L. Ovisno o broju **računskih jedinica** kojima raspolaže OpenCL uređaj više radnih grupa će se izvršavati paralelno.

Iz ovoga se da zaključiti da je veličina radne grupe (engl. *workgroup size*) određena brojem EXP, koji predstavlja broj eksperimenata koje treba provesti. Primjerice, ukoliko bi za provedbu Taguchi metode koristili ortogonalnu matricu $L_{32}(2^{31})$ onda bi veličina radne grupe bila 32 radne jedinice, jer treba provesti 32 eksperimenta.

Ovime se ograničavamo u odnosu na jednodimenzionalnu inačicu gdje se veličina radne grupe može postaviti na maksimalnu dostupnu vrijednost na OpenCL uređaju, jer se sve radne jedinice izvršavaju neovisno. Kod dvodimenzionalne implementacije to nije moguće, jer radne jedinice unutar radne grupe moraju moći međusobno komunicirati radi izmjene rezultata izračuna SNR-a pojedinih eksperimenata.



Slika 5.15: Dvodimenzionalna OpenCL inačica HTGA

Kako je postupak preopsežan za jednostavan prikaz pseudokodom, referencirat ćemo se na izvorni programski kod jezgrene funkcije prikazan u **dodatku C**.

Prvo je potrebno definirati broj dimenzija i broj eksperimenata (linije 1-3). Kako je u ovoj jezgrenoj funkciji korištena ortogonalna matrica $L_{32}(2^{31})$ te funkcija cilja ima 30 varijabli, vrijednosti su $DIM = 30$ i $EXP = 32$.

Unutar jezgrene funkcije prvo se radi dohvaćanje **grupnog identifikatora** i **lokalnog identifikatora** (linije 120-133). Uporabom grupnog identifikatora može se izračunati gdje u polju `inputParents` počinju podaci (par roditelja) za ovu radnu grupu.

Zatim je potrebno stvoriti dvodimenzionalno polje u dijeljenoj memoriji (OpenCL lokalna memorija) dimenzija $2 \times DIM$ (linije 128-133). Polje će služiti za pohranu roditeljskih jedinki čijem će genetskom materijalu pristupati sve radne jedinice unutar radne grupe. Svaka radna jedinica unutar radne grupe je odgovorna za prepisivanje dimenzije pojedinog roditelja, koja odgovara njezinom **lokalnom identifikatoru** iz globalne u dijeljenu memoriju.

U sljedećem koraku se provodi sinkronizacija dijeljene (OpenCL lokalne) memorije (linija 135), sve radne jedinice unutar radne grupe moraju doći do ove točke prije nego mogu nastaviti s daljnjim izvršavanjem. Ovime osiguravamo da je genetski materijal roditelja ispravno prepisani u dijeljenu memoriju.

Zatim je potrebno alocirati polja u dijeljenoj memoriji za pohranu izračunatih SNR-ova (veličine EXP) i rezultirajućeg potomka (veličine DIM) (linije 137-139). Funkcija `taguchiMethod2D` prvo računa SNR eksperimenta koji odgovara njezinom **lokalnom identifikatoru**. Rezultat upisuje u za to rezervirano polje u dijeljenoj memoriji, na mjesto također određeno lokalnim identifikatorom.

U nastavku je opet potrebno provesti sinkronizaciju dijeljene memorije kako bi osigurali da su sve radne jedinice unutar radne grupe izvršile njima zadani eksperiment. Sljedeći korak provode samo radne jedinice za koje vrijedi $lokalniID < DIM$, što će rezultirati time da će nekoliko radnih jedinica preskočiti izvođenje ovog odsječka. Unutar odsječka se dobiveni SNR-ovi koriste za računanje efekata faktora, svaka radna jedinica računa efekte za faktor koji odgovara njezinom **lokalnom identifikatoru**. Ovisno o odnosu rezultirajućih efekata se u polje u dijeljenoj memoriji rezervirano za pohranu potomka, na mjesto određeno **lokalnim identifikatorom**, upisuje genetski materijal odgovarajućeg roditelja.

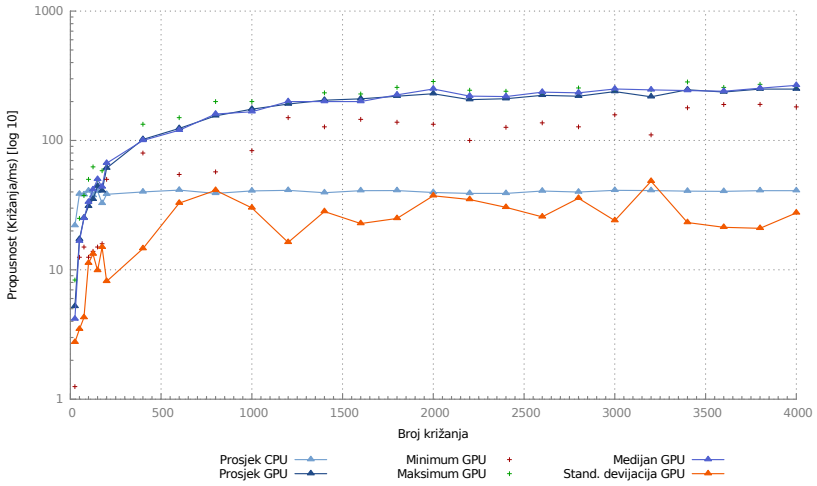
Po izlasku iz funkcije svaka radna jedinica (za koju vrijedi $lokalniID < DIM$) prepisuje dimenziju potomka koji odgovara njezinom lokalnom identifikatoru na zato predviđeno mjesto u globalnoj memoriji (linije 141-144).

5.4.3. Rezultati

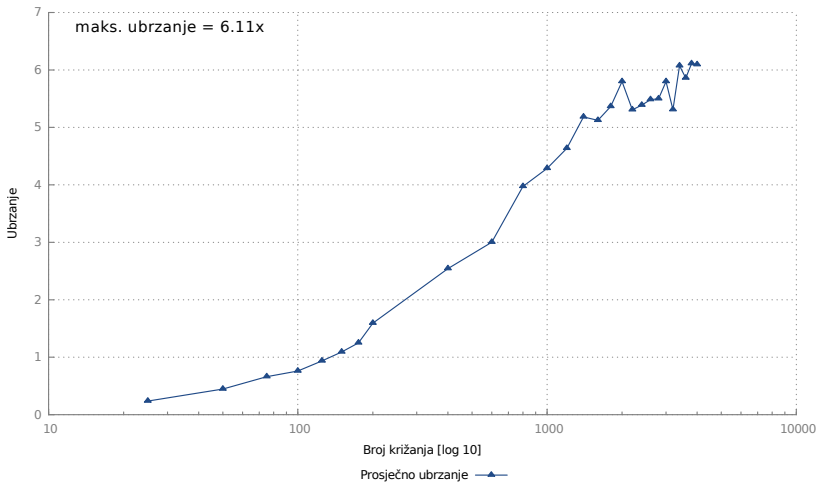
Testiranja su provedena nad jedinkama koje imaju 30 dimenzija (prikaz poljem realnih brojeva). Prilikom testiranja korištene su funkcije cilja preuzete iz COCO (engl. Comparing Continuous Optimisers) biblioteke kontinuiranih funkcija (tablica 5.5). Na grafovima os apscisa označava broj generiranih potomaka Taguchi metodom. Mjerena je propusnost (broj Taguchi metoda obavljenih u jednoj milisekundi) te prosječno ubrzanje. Radna grupa u 1D inačici ima 256 radnih jedinica.

1D implementacija

Grafovi 5.16 i 5.17 prikazuju propusnost i prosječno ubrzanje 1D inačice Taguchi križanja korištenjem sferne funkcije.

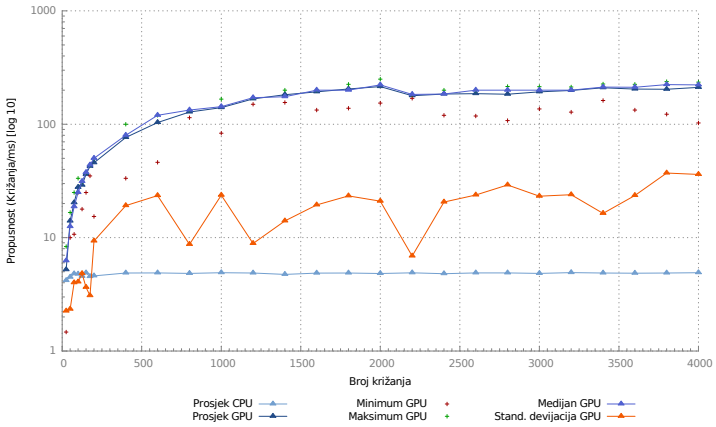


Slika 5.16: Propusnost, Taguchi križanje 1D, sferna funkcija, 30 varijabli

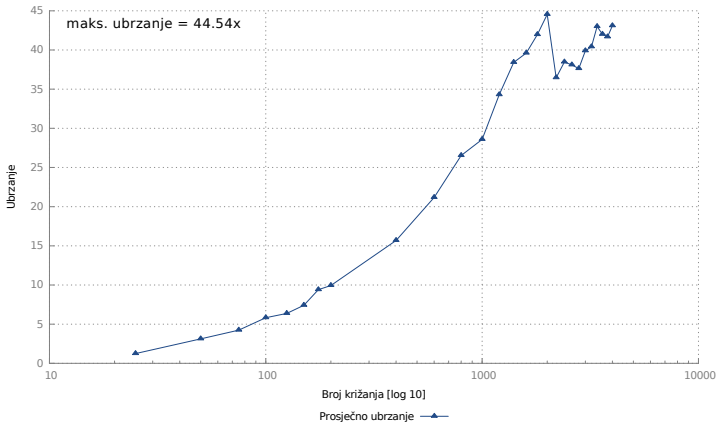


Slika 5.17: Prosječno ubrzanje, Taguchi križanje 1D, sferna funkcija, 30 varijabli

Grafovi 5.18 i 5.19 prikazuju propusnost i prosječno ubrzanje 1D inačice Taguchi križanja korištenjem Rastrigin funkcije.

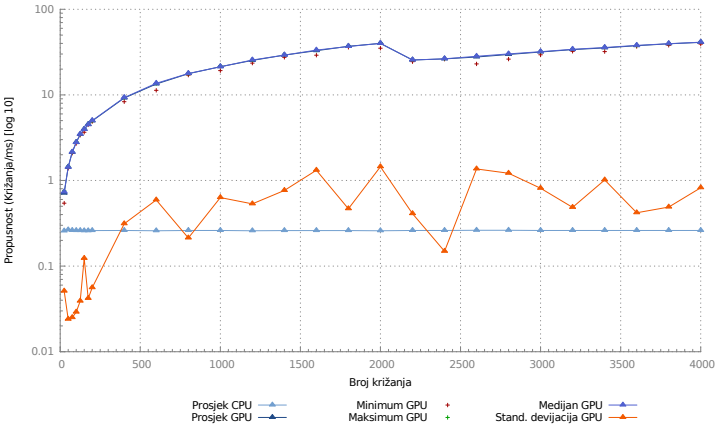


Slika 5.18: Propusnost, Taguchi križanje 1D, Rastrigin funkcija, 30 varijabli

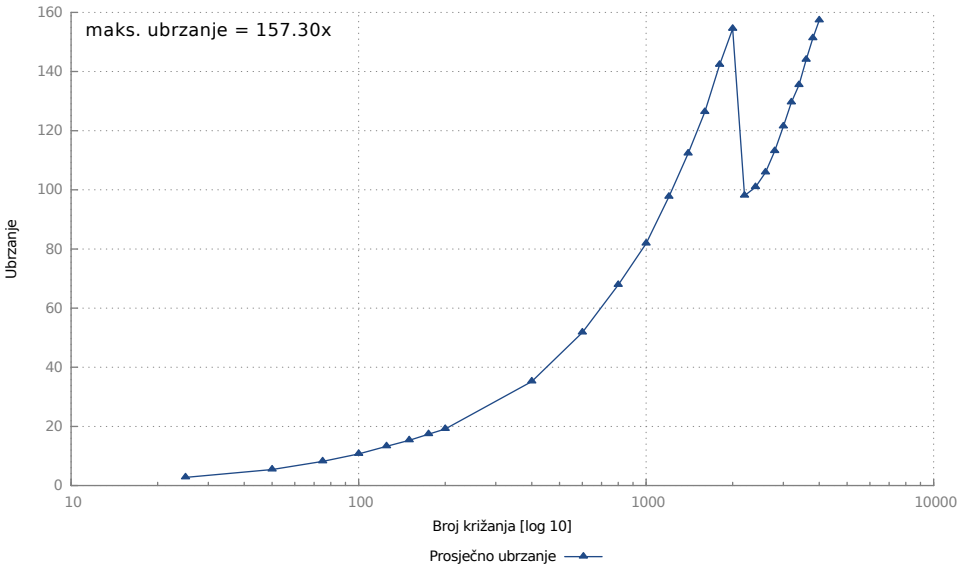


Slika 5.19: Prosječno ubrzanje, Taguchi križanje 1D, Rastrigin funkcija, 30 varijabli

Grafovi 5.20 i 5.21 prikazuju propusnost i prosječno ubrzanje 1D inačice Taguchi križanja korištenjem Weierstrass funkcije.



Slika 5.20: Propusnost, Taguchi križanje 1D, Weierstrass funkcija, 30 varijabli



Slika 5.21: Prosječno ubrzanje, Taguchi križanje 1D, Weierstrass funkcija, 30 varijabli

Vidljivo je da kao i kod Hooke-Jeeves istraži funkcije, ubrzanje raste kako raste složenost funkcije cilja. Dodatna prednost Taguchi metode je mali broj uvjetnih grananja i pravilno pristupanje ortogonalnoj matrici L u konstantnoj memoriji (svaka radna jedinica unutar radne grupe pristupa istom elementu matrice istovremeno). Tablice 5.12 i 5.13 prikazuju raspored vremena izvođenja po komponentama za sfernu i Rastrigin funkciju, prilikom generiranja 8000 potomaka.

Tablica 5.12: Taguchi 1D komponente sferna funkcija

Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	2.028
Jezgrena funkcija	7.717
Uređaj => Domaćin	0.484
Ukupno	10.229

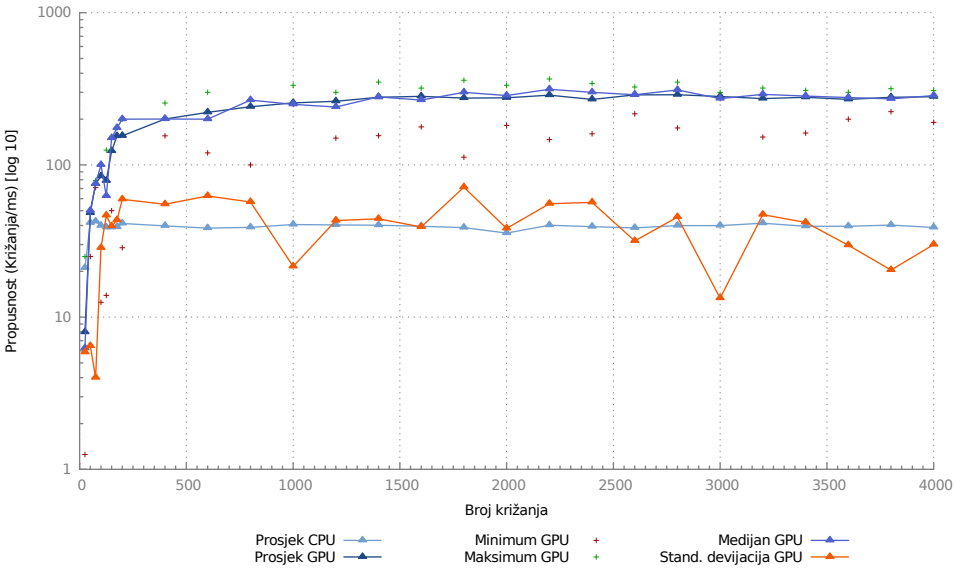
Tablica 5.13: Taguchi 1D komponente Rastrigin funkcija

Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	2.063
Jezgrena funkcija	11.568
Uređaj => Domaćin	0.489
Ukupno	14.12

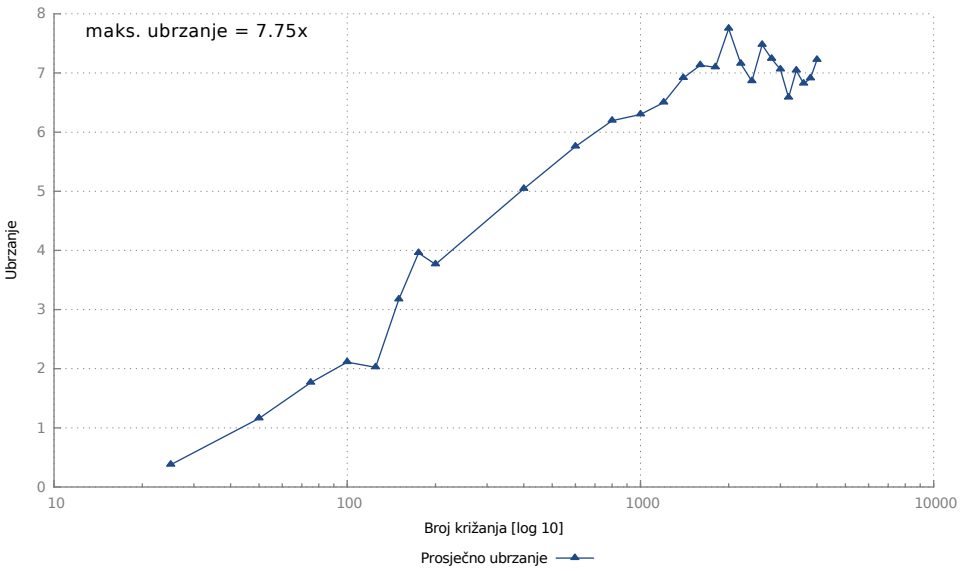
2D implementacija

Mjerenja propusnosti i prosječnog ubrzanja dvodimenzionalne inačice provedena su nad jedinkama koje imaju 30 dimenzija (prikaz poljem realnih brojeva). Radna grupa u 2D inačici ima 32 radne jedinice.

Grafovi 5.22 i 5.23 prikazuju propusnost i prosječno ubrzanje 2D inačice Taguchi križanja korištenjem sferne funkcije.

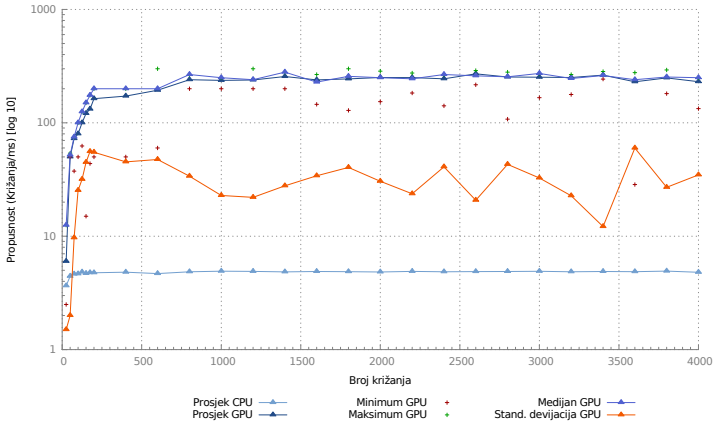


Slika 5.22: Propusnost, Taguchi križanje 2D, sferna funkcija, 30 varijabli

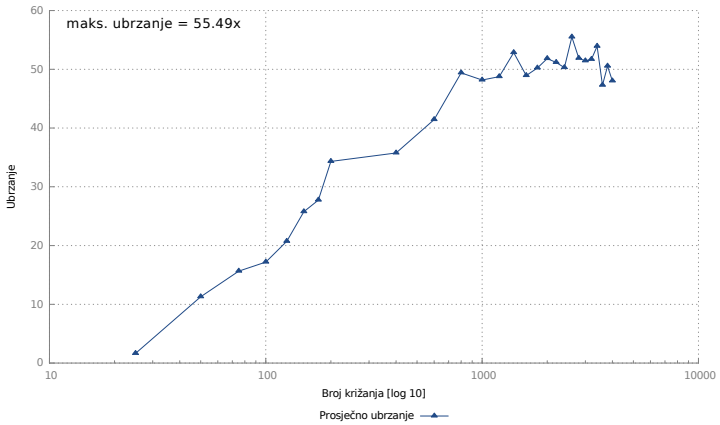


Slika 5.23: Prosječno ubrzanje, Taguchi križanje 2D, sferna funkcija, 30 varijabli

Grafovi 5.24 i 5.25 prikazuju propusnost i prosječno ubrzanje 2D inačice Taguchi križanja korištenjem Rastrigin funkcije.

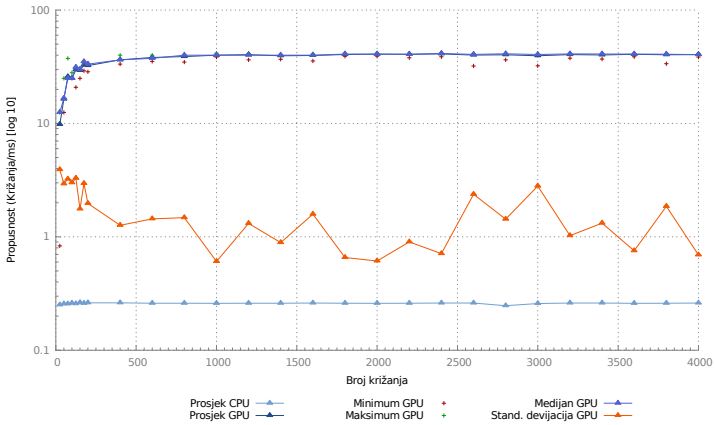


Slika 5.24: Propusnost, Taguchi križanje 2D, Rastrigin funkcija, 30 varijabli

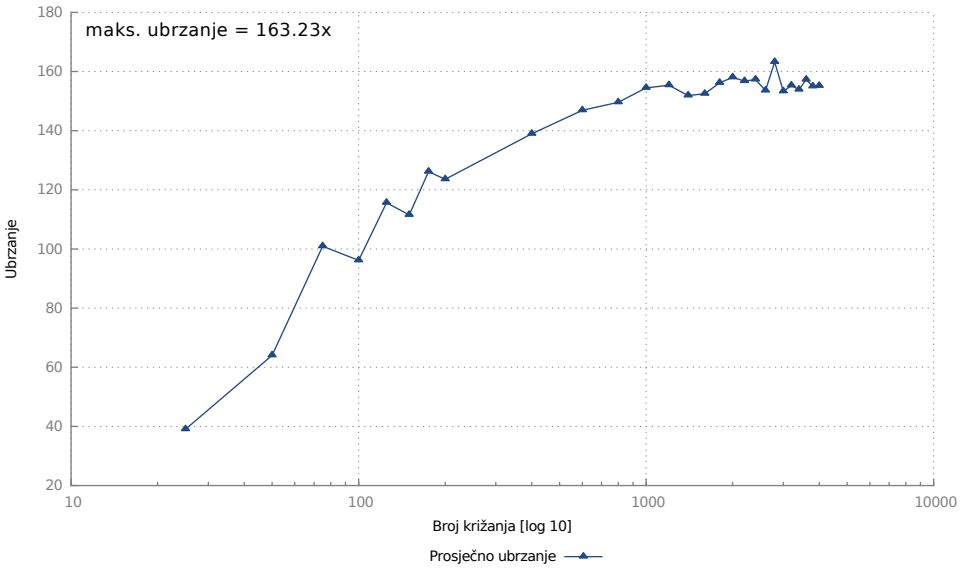


Slika 5.25: Prosječno ubrzanje, Taguchi križanje 2D, Rastrigin funkcija, 30 varijabli

Grafovi 5.26 i 5.27 prikazuju propusnost i prosječno ubrzanje 2D inačice Taguchi križanja korištenjem Weierstrass funkcije.



Slika 5.26: Propusnost, Taguchi križanje 2D, Weierstrass funkcija, 30 varijabli



Slika 5.27: Prosječno ubrzanje, Taguchi križanje 2D, Weierstrass funkcija, 30 varijabli

Rezultati dvodimenzionalne inačice su bolji od jednodimenzionalne inačice, kod Rastrigin funkcije čak i do 25%. Ovakvi rezultati pokazuju da unatoč ograničenju u generiranju samo jednog potomka po radnoj grupi, ova implementacija bolje iskorištava podatkovni paralelizam grafičke kartice provodeći samo jedan eksperiment po radnoj jedinici. Tablice 5.14 i 5.15 prikazuju raspored vremena izvođenja po komponentama za sfernu i Rastrigin funkciju, prilikom generiranja 8000 potomaka.

Tablica 5.14: Taguchi 2D komponente sferna funkcija

Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	2.252
Jezgrena funkcija	4.061
Uređaj => Domaćin	0.493
Ukupno	6.806

Tablica 5.15: Taguchi 2D komponente Rastrigin funkcija

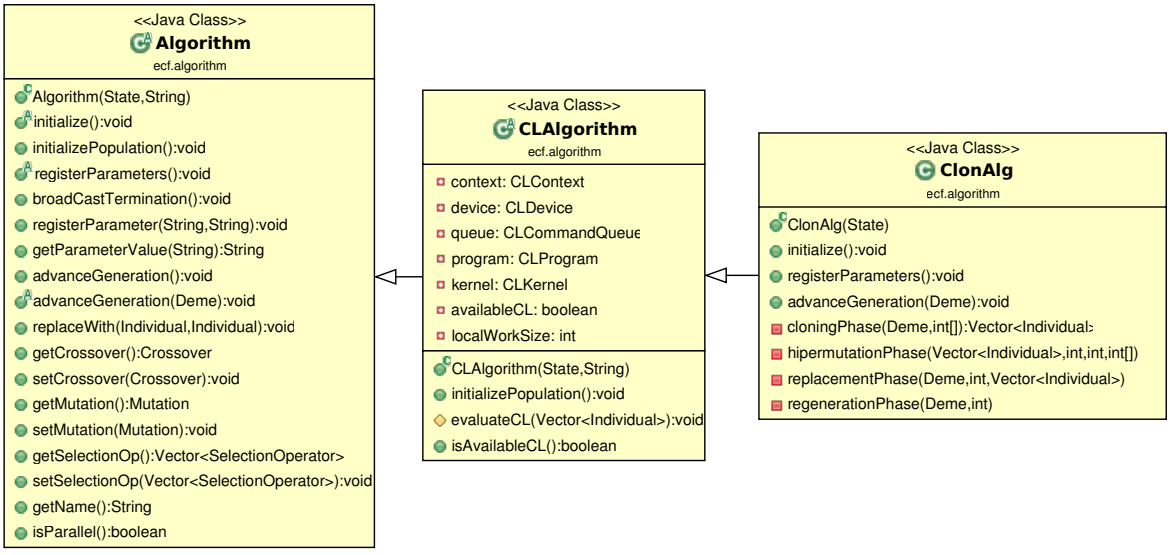
Komponenta	**Trajanje (ms)**
Domaćin => Uređaj	2.270
Jezgrena funkcija	8.272
Uređaj => Domaćin	0.491
Ukupno	11.033

5.5. Evolutionary Computation Framework

ECF (engl. *Evolutionary Computation Framework*) je razvojno okruženje za evolucijske i srodne algoritme, koje se razvija na Fakultetu elektrotehnike i računarstva. Cilj okruženja je pružiti brzo ostvarenje evolucijskog programa za korisnički problem (engl. *rapid prototyping*). Okruženje je izvorno pisano za programski jezik C++, ali je Java inačica u razvoju (ovdje opisana nadogradnja odnosi se na Java inačicu).

Unutar razvojnog okruženja implementirano je nekoliko evolucijskih algoritama (genetski algoritam, algoritam roja čestica, ... itd.), te načina prikaza rješenja (engl. *genotype*) (niz bitova, realni vektor, permutacijski vektor, ... itd.) zajedno s odgovarajućim evolucijskim operatorima. Iako korisnik može samostalno definirati dodatne algoritme, prikaze rješenja i/ili operatore, za korištenje postojećih algoritama potrebno je samo definirati funkciju cilja.

U ovom potpoglavlju će biti opisana nadogradnja ECF razvojnog okruženja, kojom je dodana mogućnost evaluacije funkcije cilja više jedinki paralelno koristeći standard OpenCL. Jezgru okruženja čini razred `Algorithm` unutar `ecf.algorithm` paketa. Svi algoritmi koji su korišteni unutar razvojnog okruženja moraju naslijediti ovaj razred. Kako bi se omogućila evaluacija jedinki koristeći standard OpenCL, razred `Algorithm` je nadograđen razredom `CLAlgorithm`. Razred `CLAlgorithm` moraju naslijediti svi algoritmi koji žele evaluirati jedinke koristeći standard OpenCL. Trenutno je implementirana podrška samo za prikaz podataka vektorom realnih brojeva. UML dijagram nadogradnje prikazan je slikom 5.28.



Slika 5.28: UML dijagram nadogradnje ECF razvojnog okruženja

Unutar razreda `CLAlgorithm` dodana je inicijalizacija OpenCL uređaja i jezgrene funkcije. Postavke poput lokacija datoteke s izvornim tekstom programa, ime jezgrene funkcije koja se želi koristiti te veličina radne grupe se mogu definirati XML atributima. Dodani XML atributi su:

- `cl.kernelFilePath` putanja do datoteke s izvornim tekstom programa jezgrene funkcije
- `cl.kernelName` ime jezgrene funkcije
- `cl.workGroupSize` veličina radne grupe, ako nije postavljena uzima se vrijednost 128

Implementirana je pomoćna metoda `evaluateCL` koja obavlja evaluaciju populacije jedinki predstavljene razredom `Deme` ili bilo kojim razredom koji nasljeđuje klasu `Vector`. Metoda `evaluateCL` će u slučaju da OpenCL uređaj nije dostupan preusmjeriti jedinke unutar populacije u pomoćnu metodu `evaluate`, koja je dio razreda `Algorithm`.

Kako bi preusmjeravanje bilo moguće, potrebno je implementirati funkciju cilja ne samo kao jezgrenu funkciju, nego i implementirajući sučelje `IEvaluate`. Prototip očekivane funkcije cilja prikazan je odsječkom 5.7.

```
__kernel void fitnessKernel (
    __global const float *individuals,
    __global float *fitness,
    const int populationSize
)
```

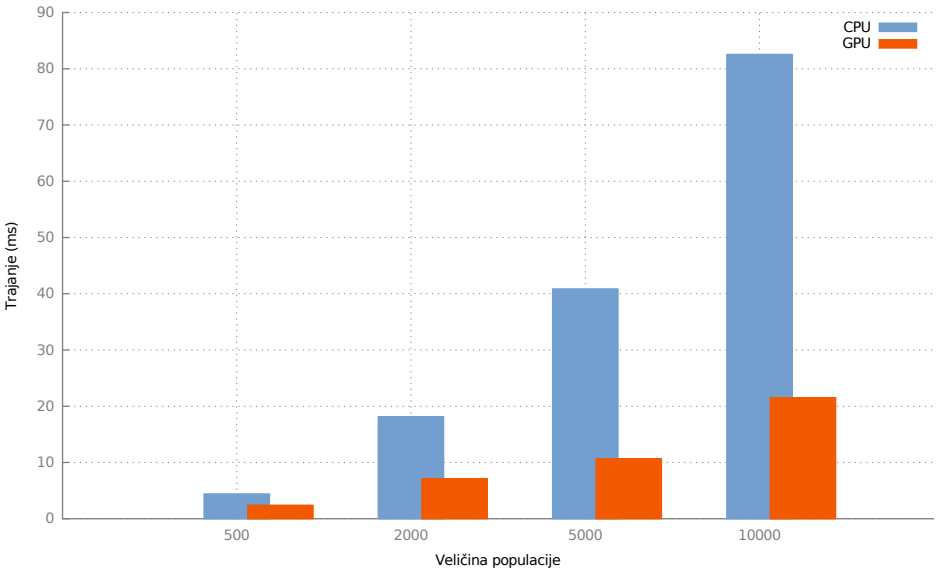
Tekst programa 5.7: Prototip ECF jezgrene funkcije evaluacije

Argument `individuals` predstavlja polje u globalnoj memoriji u koje je su smještene jedinke koje treba evaluirati (jedinke koriste zapis poljem realnih brojeva, gdje svaki broj predstavlja jednu dimenziju rješenja). Argument `fitness` je polje u koje se zapisuju rezultirajuće vrijednosti funkcije cilja za pojedinu jedinku, te argument `populationSize` predstavlja ukupni broj jedinki unutar populacije.

Osnovna izvedba jezgrene funkcije evaluacije dostupna je unutar nadogradnje, te je samo potrebno implementirati funkciju cilja za specifičan korisnički problem. Kao primjer algoritma koji koristi ovu nadogradnju je implementiran **algoritam umjetnog imunološkog sustava** (razred `ClonAlg`).

5.5.1. Rezultati

U nastavku su prikazani rezultati usporedbe brzine evaluacije populacija jedinki različitih veličina. Kao funkcija cilja je korištena Rastrigin funkcija s 30 varijabli, a veličina radne grupe je postavljena na 256 radnih jedinica.



Slika 5.29: Rezultati trajanja evaluacije (ECF)

6. Zaključak

Prilagodba postojećih algoritama za rad u hibridnim paralelnim okolinama danas je aktivno područje istraživanja. Uz danas dostupna programska radna okruženja, poput okruženja OpenCL, moguće je postići znatna ubrzanja paralelnih inačica algoritama u odnosu na slijedne. Prilikom implementacije takvih algoritama potrebno je prvo razmotriti može li se sam algoritam paralelizirati u potpunosti ili samo određeni, računalno najzahtjevniji dijelovi.

Prilikom implementacije simetričnih kriptografskih algoritama (DES i AES) se više instanci algoritma izvodilo paralelno. Rezultati su pokazali da je moguće postići značajna ubrzanja, ovisno o načinu rada algoritma. Kod onih algoritama koji su imali česte pristupe memoriji rezultati su bili lošiji, iz čega se može zaključiti da kod algoritama namijenjenih radu na grafičkoj kartici treba pažljivo optimirati pristup memoriji.

Paralelne implementacije nekoliko različitih varijanti evolucijskih algoritama su ukazale na još jednu bitnu karakteristiku dobrih algoritama namijenjenih paralelnom izvođenju. Računskom uređaju je potrebno dati dovoljno posla, kako u smislu količine paralelnih radnih zadataka, tako u smislu same složenosti tih radnih zadataka. Složenost zadataka mora biti odgovarajuća, kako cijena inicijalizacije i pokretanja zadataka na računskom uređaju ne bi bila veća od jednostavnog slijednog izvođenja zadataka. Osim toga potrebno je paziti da radni zadaci tijekom izvođenja međusobno ne odudaraju puno jedni od drugih, zbog raznih uvjetnih grananja.

Poštivanjem ovih smjernica mogu se razviti efikasni paralelni algoritmi, čija brzina izvođenja značajno nadmašuje brzinu izvođenja njihovih slijednih inačica. Pri tome se OpenCL pokazao kao odlično višeplatformsko radno okruženje za implementaciju ovih algoritama.

LITERATURA

[1] G. Barlas, A. Hassan, i Y. Al Jundi. An analytical approach to the design of parallel block cipher encryption/decryption: A CPU/GPU case study. U *Parallel, Distributed and Network-Based Processing (PDP), 2011 19th Euromicro International Conference on*, stranice 247–251. IEEE, 2011.

[2] Leo Budin, Marin Golub, Domagoj Jakobović, Leonardo Jelenković, i Nataša Jocić. *Operacijski sustavi*. Element, 2010.

[3] Linda Burnett, W Millan, Edward Dawson, i A Clark. Simpler methods for generating better Boolean functions with good cryptographic properties. *Australasian Journal of Combinatorics*, 29:231–248, 2004.

[4] Marko Čupić. Prirodom inspirirani optimizacijski algoritmi. *Fakultet elektrotehnike i računarstva*, 2010, 2009.

[5] Rob Farber. *CUDA application design and development*. Morgan Kaufmann, 2011.

[6] B. Gaster, L. Howes, D.R. Kaeli, P. Mistry, i D. Schaa. *Heterogeneous computing with OpenCL*. Morgan Kaufmann, 2011.

[7] K. Iwai, N. Nishikawa, i T. Kurokawa. Acceleration of AES encryption on CUDA GPU. *International Journal of Networking and Computing*, 2(1):pp–131, 2012.

[8] Domagoj Jakobović. Evolving Cryptographically Sound Boolean Functions, 2013.

[9] Domagoj Jakobović. Paralelno programiranje - predavanja. Fakultet elektrotehnike i računarstva, 2011.

[10] D.B. Kirk i W.H. Wen-mei. *Programming massively parallel processors: a hands-on approach*. Morgan Kaufmann, 2010.

[11] Danko Komlen. Modeliranje genske regulacijske mreže pomocu hibridnog koevolucijskog algoritma. Diplomski rad, Fakultet elektrotehnike i računarstva, 2012.

[12] G. Liu, H. An, W. Han, G. Xu, P. Yao, M. Xu, X. Hao, i Y. Wang. A Program Behavior Study of Block Cryptography Algorithms on GPGPU. U *Frontier of Computer Science and Technology, 2009. FCST'09. Fourth International Conference on*, stranice 33–39. IEEE, 2009.

[13] S.A. Manavski. CUDA compatible GPU as an efficient hardware accelerator for AES cryptography. U *Signal Processing and Communications, 2007. ICSPC 2007. IEEE International Conference on*, stranice 65–68. IEEE, 2007.

[14] James McLaughlin i John A Clark. Evolving balanced Boolean functions with optimal resistance to algebraic and fast algebraic attacks, maximal algebraic degree, and very high nonlinearity.

[15] Aaftab Munshi, Benedict Gaster, Timothy G Mattson, i Dan Ginsburg. *OpenCL programming guide*. Addison-Wesley Professional, 2011.

[16] N. Nishikawa, K. Iwai, i T. Kurokawa. High-Performance Symmetric Block Ciphers on CUDA. U *Networking and Computing (ICNC), 2011 Second International Conference on*, stranice 221–227. IEEE, 2011.

[17] Ashu Rege NVIDIA. An Introduction to modern GPU architecture, 2009. URL `ftp://download.nvidia.com/developer/cuda/seminar/TDCI_Arch.pdf`.

[18] Timo Stich NVIDIA. OpenCL on NVIDIA GPUs, 2009. URL `http://sa09.idav.ucdavis.edu/docs/SA09_NVIDIA_IHV_talk.pdf`.

[19] nVidia corp. *OpenCL Best Practices Guide, Version 1.0*. nVidia corp., 2010.

[20] Jinn-Tsong Tsai, Tung-Kuan Liu, i Jyh-Horng Chou. Hybrid Taguchi-genetic algorithm for global numerical optimization. *Evolutionary Computation, IEEE Transactions on*, 8(4):365–377, 2004. ISSN 1089-778X. doi: 10.1109/TEVC.2004.826895.

[21] Ryoji Tsuchiyama, Takashi Nakamura, Takuro Iizuka, Akihiro Asahara, Satoshi Miki, i Satoru Tagawa. *The OpenCL Programming Book*. Fixstars Corporation, 2010.

[22] X. Wang, X. Li, M. Zou, i J. Zhou. AES finalists implementation for GPU and multicore CPU based on OpenCL. U *Anti-Counterfeiting, Security and Identification (ASID), 2011 IEEE International Conference on*, stranice 38–42. IEEE, 2011.

Razvoj računalno zahtjevnih algoritama u hibridnoj paralelnoj okolini

Sažetak

Tijekom posljednjih nekoliko godina, mnoga istraživanja bave se tematikom učinkovite prilagodbe i implementacije postojećih algoritama za izvođenje na grafičkim karticama (engl. *GPU - Graphics Processing Unit*). U radu su ispitane mogućnosti ubrzanja danas često korištenih kriptografskih algoritma (AES i DES), te algoritma koji su bazirani na evolucijskom računanju (GA, Hibridni Taguchi GA, Hibridni Hooke-Jeeves GA). Paralelizacija je kod nekih algoritama izvedena samo za određene, računalno najzahtjevnije operacije. Rezultati su pokazali da se čak i sa standardnom grafičkom karticom mogu postići osjetna ubrzanja u radu ovih algoritama. U radu su opisani radno okruženje OpenCL i korišteni algoritmi, te je dan pregled arhitekture grafičkih kartica. Unutar ECF programskog okruženja implementirana je podrška za evaluaciju populacija jedinki na grafičkoj kartici korištenjem standarda OpenCL.

Ključne riječi: Paralelno izvođenje, OpenCL, GPGPU, AES, DES, Kombinatoričke funkcije, Taguchi metoda, HTGA, Hooke-Jeeves, ECF

Development of computationally demanding algorithms in a hybrid parallel environment

Abstract

In recent years there have been many studies dealing with the topic of effective adaptation and implementation of existing algorithms to run on graphics cards. This paper investigates the possibilities of accelerating some of today's frequently used cryptographic algorithms (AES and DES), and algorithms that are based on evolutionary computation (GA, Hybrid Taguchi GA, Hybrid Hooke-Jeeves GA). For some algorithms parallelization is performed only for specific, most computationally demanding operations. The results showed that even with a standard graphics card one can achieve a noticeable acceleration of these algorithms. The paper describes the OpenCL framework and the algorithms used, and an overview of the graphics cards architecture. Within the ECF programming environment the support for the evaluation of populations of individuals on the GPU using the OpenCL standard is implemented.

Keywords: Parallel execution, OpenCL, GPGPU, AES, DES, Boolean functions, Taguchi method, HTGA, Hooke-Jeeves, ECF

Dodatak A

OpenCL kernel - Hooke-Jeeves istraži

```
#define DIM 30
#define fitness(DATA) MyFitness(DATA);

/******************************************/
/* FITNESS FUNCTION                       */
/******************************************/


float MyFitness(float *data) {

    // Definition

}


/******************************************/
/* HOOKE-JEEVES EXPLORE                   */
/******************************************/


float explore(float *data, float delta) {

    float fit[3];
    fit[0] = fitness(data);

    float values[DIM];
    for (int i = 0; i < DIM; i++) {

        values[i] = data[i];

    }

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

        data[i] = values[i] + delta;
        fit[1] = fitness(data);

        if (fit[1]>fit[0]) {
            data[i] = values[i] - delta;
            fit[1] = fitness(data);
            if (fit[1]>fit[0])
                data[i] = values[i];
        }

    }

    return fitness(data);

}

```

```
51  __kernel void HJExplore( __global const float *population,
52          __global const float *deltas,
53          __global float *resultPopulation,
54          __global float *resultFitness,
55          const int populationSize ) {
56
57      const size_t id = get_global_id( 0 );
58      if ( id >= populationSize ) return;
59
60      const size_t startPos = DIM * id;
61
62      float delta = deltas[id];
63
64      float individual[DIM];
65      for (int i = 0; i < DIM; i++) {
66          individual[i] = population[startPos+i];
67      }
68
69      float fitness = explore(individual, delta);
70
71      resultFitness[id] = fitness;
72      for (int i = 0; i < DIM; i++) {
73          resultPopulation[startPos+i] = individual[i];
74      }
75
76  }
```

Dodatak B

OpenCL kernel - Taguchi križanje

1 radna jedinica / 1 potomak

```
#define DIM 30
#define EXPS 32
#define fintess(DATA) MyFitness(DATA);


__constant short L[32][31] = {
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
    {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
    {0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1},
    {0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0},
    {0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1},
    {0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0},
    {0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0},
    {0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1},
    {0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1},
    {0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,1,1,1,1,0,0,1,1,0,0,1,1,0,0,1,0},
    {0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1,1,1,0,0,1,1,0,0},
    {0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1},
    {0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0},
    {0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1},
    {0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,1},
    {0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,0},
    {1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1},
    {1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0},
    {1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0},
    {1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,0,1},
    {1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0},
    {1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1},
    {1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1},
    {1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0},
    {1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0},
    {1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1},
    {1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1},
    {1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0},
    {1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1},
    {1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0},
    {1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0},
    {1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1}
};


/*****************************************/
/* FITNESS FUNCTION                      */
/*****************************************/

```

```
47
48  float MyFitness(float *data) {
49
50      // Definition
51
52  }
53
54
55  float calcSNR(float *exp) {
56
57      float expFitness = fintess(exp);
58      return 1/(expFitness*expFitness);
59
60  }
61
62
63  void taguchiMethod1D(float parents[2][DIM], float *child) {
64
65      // EXPERIMENTS
66      float exps[EXPS][DIM];
67      // SOUND TO NOISE RATIOS
68      float snrs[EXPS];
69
70      // LEVEL FACTORS
71      float E0[DIM] = { 0.0 };
72      float E1[DIM] = { 0.0 };
73
74      short indexL;
75
76      // GENERATE EXPERIMENTS AND CALCULATE SNRs
77      for (int exp = 0; exp < EXPS; exp++) {
78
79          for (int dim = 0; dim < DIM; dim++) {
80
81              indexL = L[exp][dim];
82              exps[exp][dim] = parents[indexL][dim];
83
84          }
85
86          snrs[exp] = calcSNR(exps[exp]);
87
88      }
89
90      // CALCULATE LEVEL FACTORS FOR EVERY DIMENSION
91      // AND PRODUCE OPTIMAL CHILD
92      for (int dim = 0; dim < DIM; dim++) {
93
94          for (int exp = 0; exp < EXPS; exp++) {
95
96              indexL = L[exp][dim];
97              if (indexL == 0) {
98                  E0[dim] += snrs[exp];
99              } else {
100                 E1[dim] += snrs[exp];
101             }
102
103         }
104
105         if (E0[dim] >= E1[dim]) {
106             child[dim] = parents[0][dim];
107         } else {
108             child[dim] = parents[1][dim];
109         }
110
111     }
112
113 }
114
115
```

```
116 __kernel void TaguchiCrossover1D( __global const float *inputParents,
117         __global float *outputChildren,
118         const int crossoverNum ) {
119
120     const size_t id = get_global_id( 0 );
121     if ( id >= crossoverNum ) return;
122
123     const size_t startPosParents = 2 * DIM * id;
124     const size_t startPosChild = DIM * id;
125
126     // GLOBAL => LOCAL
127     float parents[2][DIM], child[DIM];
128     for (int i = 0; i < 2*DIM; i++) {
129         if (i < DIM) {
130             parents[0][i] = inputParents[startPosParents+i];
131         } else {
132             parents[1][i-DIM] = inputParents[startPosParents+i];
133         }
134     }
135
136     taguchiMethod1D(parents, child);
137
138     // LOCAL => GLOBAL
139     for (int i = 0; i < DIM; i++) {
140         outputChildren[startPosChild+i] = child[i];
141     }
142
143 }
```

Dodatak C

OpenCL kernel - Taguchi križanje

32 radne jedinice / 1 potomak

```
#define DIM 30
#define EXPS 32
#define fintess(DATA) MyFitness(DATA);


__constant short L[32][31] = {
     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0},
     {0,0,0,0,0,0,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1},
     {0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1},
     {0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0},
     {0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1},
     {0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0},
     {0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1,1,1,1,1,0,0,0,0},
     {0,0,0,1,1,1,1,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,0,0,0,0,1,1,1,1},
     {0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1},
     {0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,1,1,1,1,0,0,1,1,0,0,1,1,0,0,1,0},
     {0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1,1,1,0,0,1,1,0,0},
     {0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1},
     {0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0},
     {0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1},
     {0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,0,0,1,1,1,1,0,0,1,1,0,0,0,0,1,1},
     {0,1,1,1,1,0,0,1,1,0,0,0,0,1,1,1,1,0,0,0,0,1,1,1,1,0,0,1,1,0,0},
     {1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1},
     {1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,1,0},
     {1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0},
     {1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,0,1},
     {1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0},
     {1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1},
     {1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0,1,0,1,0,0,1,0,1},
     {1,0,1,1,0,1,0,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,0,1,0,1,1,0,1,0},
     {1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,1,0},
     {1,1,0,0,1,1,0,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1},
     {1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1},
     {1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0},
     {1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0,1,0,0,1},
     {1,1,0,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,0,1,1,0},
     {1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,0,1,1,0,1,0,0,1,1,0,0,1,0,1,1,0},
     {1,1,0,1,0,0,1,1,0,0,1,0,1,1,0,1,0,0,1,1,0,0,1,1,0,1,0,0,1,1,1}
};


/*****************************************/
/* FITNESS FUNCTION                      */
/*****************************************/


```

```
47
48
49  float MyFitness(float *data) {
50
51      // Definition
52
53  }
54
55  float calcSNR(float *exp) {
56
57      float expFitness = fintess(exp);
58      return 1/(expFitness*expFitness);
59
60  }
61
62
63  void taguchiMethod2D(float parents[2][DIM],
64          __local float *child,
65          __local float *snrs,
66          size_t lId) {
67
68      // EXPERIMENT
69      float exp[DIM];
70
71      // LEVEL FACTORS
72      float E0 = 0;
73      float E1 = 0;
74
75      short indexL;
76
77      // GENERATE EXPERIMENT AND CALCULATE SNR
78      if (lId < EXPS) {
79
80          for (int dim = 0; dim < DIM; dim++) {
81              indexL = L[lId][dim];
82              exp[dim] = parents[indexL][dim];
83          }
84
85          snrs[lId] = calcSNR(exp);
86
87      }
88
89      // LOCAL MEMORY SYNC
90      barrier(CLK_LOCAL_MEM_FENCE);
91
92      // CALCULATE LEVEL FACTORS FOR LOCAL_ID DIMENSION
93      // AND PRODUCE LOCAL_ID OPTIMAL CHILD PART
94      if (lId < DIM) {
95
96          for (int exp = 0; exp < EXPS; exp++) {
97              indexL = L[exp][lId];
98              if (indexL == 0) {
99                  E0 += snrs[exp];
100             } else {
101                 E1 += snrs[exp];
102             }
103         }
104
105         if (E0 >= E1) {
106             child[lId] = parents[0][lId];
107         } else {
108             child[lId] = parents[1][lId];
109         }
110
111     }
112
113 }
114
115
```

```
116 __kernel void TaguchiCrossover2D( __global const float *inputParents,
117         __global float *outputChildren,
118         const int crossoverNum ) {
119
120     const size_t gId = get_group_id( 0 );
121     if ( gId >= crossoverNum) return;
122
123     const size_t lId = get_local_id( 0 );
124
125     const size_t startPosParents = 2 * DIM * gId;
126     const size_t startPosChild = DIM * gId;
127
128     // GLOBAL => SHARED
129     __local float parents[2][DIM];
130     if (lId < DIM) {
131         parents[0][lId] = inputParents[startPosParents+lId];
132         parents[1][lId] = inputParents[startPosParents+DIM+lId];
133     }
134
135     barrier(CLK_LOCAL_MEM_FENCE);
136
137     __local float child[DIM];
138     __local float snrs[EXPS];
139     taguchiMethod2D(parents, child, snrs, lId);
140
141     // SHARED => GLOBAL
142     if (lId < DIM) {
143         outputChildren[startPosChild+lId] = child[lId];
144     }
145
146 }
```