UNIVERZITET U NIŠU
ELEKTRONSKI
FAKULTET

IZRAČUNAVANJE VILENKIN-CHRESTENSON SPEKTRA KVATERNARNIH FUNKCIJA KORIŠĆENJEM MPI FRAMEWORK-A

SEMINARSKI RAD

Predmet: Primena višeznačne logike u obradi diskretnih signala

Student:

Tina Radenković, br. ind. 1128

Mentor:

Prof. dr Miloš Radmanović

Niš, februar 2021. god.

Sadržaj

1. Uvod

U današnje vreme je sve više podataka i informacija koje je potrebno na različite načine obrađivati. Veliki broj ovih podataka je teško predstaviti u njihovom osnovnom domenu, pa ih je neophodno transformisati u druge domene koji su jednostavniji za obradu. Jedan od najčešće primenjivanih domena je spektralni domen. Spektralne transformacije vrše redistribuciju sadržaja signala čime olakšavaju uočavanje njihovih karakterističnih svojstava kao i izvršavanje operacija koje su samo u ovom domenu dostupne. Međutim, zbog ovih svojstava one su i računski veoma zahtevne pa je često vreme obrade signala na ovaj način ograničavajući faktor. Kao rešenje ovog problema implementirani su brzi algoritmi poput algoritama za izračunavanje Vilenkin-Chrestenson-a (VC) spektra funkcija višeznačne logike (eng. *multiple-valued logic*) MVL. Transformacije korišćenjem VC spektra predstavljaju koristan matematički alat za analizu, sintezu i optimizaciju višeznačnih funkcija.

Razvojem grafičkih procesnih jedinica (GPU) za opštu namenu omogućena je jednostavnija implementacija brzih algoritama. Računarska grafika se velikim delom zasniva na matričnim proračunima što GPU čini veoma pogodnim za ova izračunavanja, a samim tim i za izračunavanja spektralnih transformacija. Algoritmi zasnovani na Furijeovim transformacijama (FFT algoritmi) su osmišljeni za izračunavanje spektra uz pomoć minimalnog broj operacija. Međutim, osnovne karakteristike GPU arhitekture utiču na promenu fokusa sa ukupnog izvršavanja na količinu memorijskih transfera. Zbog toga direktno mapiranje brzih algoritama na GPU ne obezbeđuje adekvatnu iskorišćenost računarskih resursa pa je neophodno prilagoditi ove algoritme GPU hardveru. Tako na primer, bolje je ponovo izvršiti neophodna izračunavanja, nego čuvati i ponovo učitavati podatke iz memorije.

Izračunavanje Vilenkin-Chrestenson spektra na GPU-u moguće je implementirati uz pomoć različitih algoritama, međutim dva najkorišćenija FFT algoritma su Cooley-Tukey i algoritmi konstantne geometrije. Dodatno, moguće je koristiti posebna mapiranja ovih algoritama na GPU arhitekturu i memoriju računarskog sistema. U hardveru GPU-a određene operacije se implementiraju direktno, kao pojedinačne instrukcije, dok se druge operacije mogu prevesti u više mašinskih instrukcija. Na primer, aritmetika adrese u algoritmu Cooley-Tukey utiče na korišćenje operacija koje nisu direktno implementirane u hardveru za razliku od algoritma konstantne geometrije. Cooley-Tukey algoritam čita i upisuje u iste memorijske lokacije i, prema tome, može biti implementiran korišćenjem tzv. in-place implementacije. Algoritam konstantne geometrije čita iz jednog skupa lokacija i upisuje u drugi pa, samim tim, zahteva tzv. out-of-place implementaciju. Kompromis između broja aritmetičkih operacija i memorijskih zahteva je od suštinskog značaja za efektivno izračunavanje i implementaciju uz pomoć GPU-a.

Programiranje GPU-a ima određena ograničenja koja je moguće prevazići pravilnim korišćenjem metoda poput upotrebe tzv. page-locked memorije, razmene argumenata za optimizaciju memorijskih transfera kao i korišćenja kompajlerskih opcija za dinamičku alokaciju memorije unutar GPU kernela. Ovim metodama omogućeno je jednostavnije i efektivnije programiranje GPU-a.

U ovom radu biće predstavljene teorijske osnove Vilenkin-Chrestenson spektra i transformacija funckija višeznačne logike, kao i detaljna analiza implemntiranih algoritama za izračunavanje Vilenkin-Chrestenson spektra kvaternarnih funkcija korišćenjem MPI framework-a.

2. Vilenkin-Chrestenson transformacija

Vilenkin-Chrestenson-ova transformacija je uopštavanje Walsh-ove transformacije za p vrednost gde je p prost broj. Ovu transformaciju je definisano prvobitno Chrestenson, a zatim i Vilenkin na osnovu čijih imena je ona i dobila naziv.

Vektor logičke funkcije p-vrednosti i n-promenljive je definisan na sledeći način: $\{ 0,1,....p\text{-}1\}^n \rightarrow \{ 0,1,....p\text{-}1\}$ sa $F = [f(0),f(1), ..., f(p^n - 1)]^T$. Funkcija f se može posmatrati funkcijom definisanom nad konačnom Abelian grupom $G = C_p{}^n = (\{ 0,1,....p\text{-}1\}^n , \oplus p)$, gde je Cp ciklična grupa reda, a p je operacija komponentnog sabiranja modula p. Ova karakteristika omogućava upotrebu Vilenkin-Chrestenson transformacije u obradi p-vrednosti funkcija kao Furijeovu transformaciju. Vilenkin-Chrestenson funkcija je definisana kao

$$\chi_{\omega}^{(p)}(z) = \exp\left(\frac{2\pi}{p} i \sum_{s=0}^{n-1} \omega_{n-1-s} z_s\right)$$

za w,z = $0,1,....p^n\text{-}1$ gde je i koren od -1, ws, zs pripadaju skupu $\{ 0,1,....p\text{-}1\}$ i w i z su dobijeni na sledeći način:

$$\omega = \sum_{s=0}^{n-1} \omega_s p^{n-1-s}, \quad z = \sum_{s=0}^{n-1} z_s p^{n-1-s} .$$

U matričnoj notaciji, Vilenkin-Chrestenson funkcije su predstavljene kao kolone Vilenkin-Chrestenson matrice:

$$\mathbf{VC}_p(n) = \bigotimes_{i=1}^{n} \mathbf{VC}_p(1)$$

gde je $VC_p(1)$ osnovna Vilenkin-Chrestenson matrica za cikličnu grupu Cp, a Ⓧ označava Kronekerov proizvod.

Vilenkin-Chrestenson spektar, predstavljen kao $S_f = [S_f(0), S_f(1),\ldots S_f(p^n\text{-}1)]^T$, je dobijen korišćenjem inverzne matrice $\mathbf{S}_f = \mathbf{VC}_p^{-1}(n) \cdot \mathbf{F}$. Za p=3 i n=2 Vilenkin-Chrestenson matrica transformacije se može dobiti na sledeći način:

Ciklična grupa	C_2	C_3	C_4
$\mathbf{VC}_p^{-1}(1)$	$\frac{1}{2}\begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}$	$\frac{1}{3}\begin{bmatrix} 1 & 1 & 1 \\ 1 & e_2 & e_1 \\ 1 & e_1 & e_2 \end{bmatrix}$	$\frac{1}{4}\begin{bmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{bmatrix}$

Slika 1. Vilenkin-Chrestenson matrice transformacije

gde je $e_k = \exp(2\pi ik/3)$, za k = 0, 1, 2. U opštem slučaju (i,j) element transformacione matrice se dobija kao $e_k = \exp(2\pi ik/p)$, za k = (p-i*j) mod p, gde i, j uzimaju vrednosti od 0 do p-1.

Kako bi se izvršila detaljnija analiza Vilenkin-Chrestenson transformacije posmatra se slučaj za p=3 i n-promenljivu funkciju sa tri vrednosti f (k1, ..., kn), gde svaka promenljiva ki, i ∈ {1, ..., n} uzima vrednosti iz skupa {0, 1, 2}. Neka je F(n) vektor funkcije f. U matričnoj notaciji Vilenkin-Chrestenson-ova transformacija može se zapisati kao:

$$\mathbf{VC}(n) = \bigotimes_{i=1}^{n} C(1), \quad \mathbf{C}(1) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \xi & \xi^2 \\ 1 & \xi^2 & \xi \end{bmatrix}$$

gde su ε i ε^2 kompleksni brojevi predstavljeni kao $\varepsilon = e^{2\pi i/3}$ i $\varepsilon^2 = e^{4\pi i/3}$ kada je $|\varepsilon| = |\varepsilon^2| = 1$. Vilenkin-Chrestenson spektar je dobijen množenjem vektora funkcije sa inverznom matricom transformacije:

$$\mathbf{S}(n) = \bar{\mathbf{C}}(n) * \mathbf{F}(n)$$

gde je $\bar{C}$(n) konjuktivna vrednost od C(n):

$$\bar{\mathbf{C}}(n) = \bigotimes_{i=1}^{n} \bar{\mathbf{C}}(1), \quad \bar{\mathbf{C}}(1) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & \xi^2 & \xi \\ 1 & \xi & \xi^2 \end{bmatrix}$$

U ovakvim sitacijama korisno je korišćenje kodiranja vrednosti ternarnih promenljivih k_i funkcije f (k_1, ..., k_n) na sledeći način: $k_1 \rightarrow \varepsilon^{k1}$, za [0,1,2] → [1,ε,$\varepsilon^2$].

Primer1: Razmotrimo dve promenljive funkcije f1 = x1x2. Vektor istine funkcije je F1 = [1, 1, 1, 1, x, x2, 1, x2, x], Vilenkin-Chrestenson spektar je S1 = [1, 1, 1, 1, v2, v1, 1, v1, v2], pri čemu v1 = 3x i v2 = 3x2.
Kako bi ovo izračunavanje moglo da se izvrši neophodan je i dovoljan uslov da višeznačna n-promenljiva funkcije f u GF(p) bude bent funkcija ukoliko je apsolutna vrednost svih njegovih Vilenkin-Chrestenson koeficijenata jednaka $p^{n/2}$. Zbog svojstava Vilenkin-Chrestenson transformacije, da bi ovaj kriterijum bio zadovoljen n mora biti parno, odnosno $n = 2 * k$, $k \in N$. Ovo je analogno definiciji bent funkcija korišćenjem svojstva njihovog Walsh spektra u GF(2). Vilenkin-Chrestenson-ov spektar bent funkcija se može u opštem obliku napisati na sledeći način: $S = p^{n/2} S^i$. Kako bi jednostavnije bila objašnjena ova transformacija koristiće se samo deo S^i. Za funkciju f1 iz primera 1 S^1 se može predstaviti kao [1, 1, 1, 1, x2, x, 1, x, x2].

Vilenkin-Chrestenson transformacije invarijantnih operacija se mogu definisati preko diskretnih funkcija u GF(p). Kako bi analiza bila jednostavnija, p se postavlja na vrednost tri, međutim sva izračunavanja važe za vrednosti p veće od 2. Neka $k \in \{1, 2\}$ i $\oplus_3$ označavaju sabiranje po modulu tri, što se može napisati u tabelarnom obliku kao:

$\oplus_3$	**0**	**1**	**2**
0	0	1	2
1	1	2	0
2	2	0	1

Za [0,1,2] → [1,ε,ε^2] matrica ima sledeći oblik:

$\oplus_3$	**0**	**E**	$\mathbf{E^2}$
0	1	ε	ε^2
E	ε	ε^2	1
$\mathbf{E^2}$	ε^2	1	ε

Komplement binarne promenljive se može izraziti kao ki = ki $\oplus$ 1. Ekvivalent u GF (p) je promena polariteta promenljive, tj. za p = 3 ovo je ki $\oplus_3$ k. Postoji pet vrsta Vilenkin-Chrestenson spektralno invarijantnih operacija:

1. komplement izlaza funkcije g(x) = f (x) $\oplus_3$x, ova operacije je ekvivalentna sa g(x) = f (x);
2. komplement ulaza funkcije g(x) = f (x) $\oplus_3$ x;
3. modifikacija izlaza funkcije g(x) = f (x) $\oplus_3$ x;
4. razmena ulaznih promenljivih xi $\leftrightarrow$ xj;
5. zamena ulazne promenljive xi $\rightarrow$ xi $\oplus_3$ kx$_j$.

Bilo koja linearna operacija nad funkcijom f(x) je Vilenkin-Chrestenson spektralno invarijantna operacija. Kao i u slučaju Walsh-ovih spektralno invarijantnih operacija, u spektralnom domenu Vilenkin-Chrestenson transformacija se ne menjaju apsolutne vrednosti koeficijenata transformacije. Rezultat izvršenja spektralno invarijantna operacija je ili permutacija spektralnih koeficijenata ili promena polariteta nekih od koeficijenata. Spektralno invarijantna operacija g (x) = f (x) $\oplus_3$ kao rezultat izvršenja ima promenu polariteta spektralnih koeficijenata S'$_i$(g)= S'$_i$(f) $\oplus_3$ ε^k. Na primer, dve promenljive ternarne funkcije f1 = x1x2 daju spektar S'$_i$(1) = [1,1,1,1, ε^2, ε,1, ε, ε^2], funkcija f2=f1$\oplus_3$1. Vektor istinitosti funkcije f2 je F2=[ε, ε, ε, ε, ε^2, 1, ε,1, ε^2] i odgovarajući spektar S'$_2$ = [ε, ε, ε, ε,1,ε^2, ε,ε^2,1]. Spektralno invarijantne operacije preostale četiri vrste rezultuju permutacijom spektralnih koeficijenata u Vilenkin-Chrestenson spektru date funkcije. Promena polariteta ulaznih promenljivih je predstavljena u sledećem primeru.

Primer2: Funkcija f1=x1x2, promena polariteta ulazne promenljive x1=x1$\oplus_3$1 dovodi do permutacije vektora istine u F3=[1, ε, ε^2, 1, ε^2, ε, 1,1, 1]. Nakon primene Vilenkin-Chrestenson transformacije spektar postaje S'$_3$ = [1,1,1, ε, 1, ε^2, ε^2,1, ε] i predstavlja permutaciju spektra

Primer3: Funkcija f4=f1$\oplus_3$x1. Transformacija funkcije f1 dovodi do permutacije vektora istinitosti i spektra u F4=[1,1,1, ε, ε^2,1, ε^2, ε,1], S'$_4$=[1, ε, ε^2, 1, 1, 1, 1, ε^2, ε].

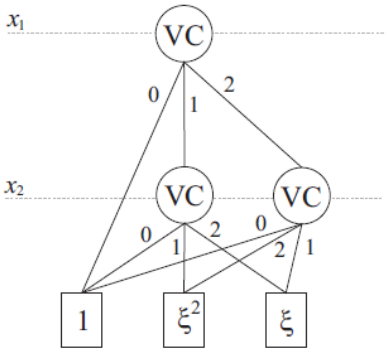
Ovo svojstvo spektralno invarijantnih operacija ima važne posledice na višeznačne bent funkcije. Iz prethodnih primera se može zaključiti da će bilo koja funkcija dobijena primenom jedne ili više spektralno invarijantnih operacija nad bent funkcijama takođe biti bent funkcija.

2.1. Vilenkin-Chrestenson dijagrami odluka

Dijagrami odlučivanja predstavljaju grafički metod predstavljanja diskretnih funkcija. Označeni su acikličknim grafom, koji se sastoji od skupa čvorova (terminalnih i ne terminalnih) i skupa međusobno povezanih grana. Dobija se smanjenjem stabla odluke. Stabla odlučivanja su strukture dobijene rekurzivnom primenom određenog pravila dekompozicije na sve promenljive date diskretne funkcije. U zavisnosti od od vrste diskretnih funkcija i izbora pravila dekompozicije postoje različite vrste dijagrama odlučivanja.

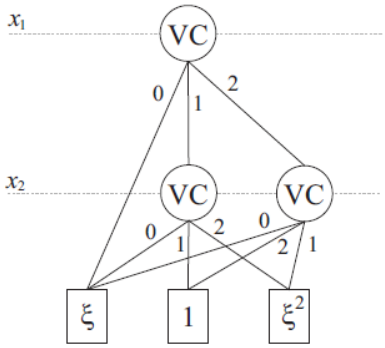
Dijagrami odluke Vilenkin-Chrestenson transformacija su predstavljeni na sličan način kao i Walsh-ovi dijagrami odluke za GF(p) gde je p veće od dva. Neka je f (x1, ..., xn) n-promenljiva diskretna funkcija u GF(p), gde je p prost. Vilenkin-Chrestenson dijagrami odluka se dobijaju rekurzivnom primenom Vilenkin-Chrestenson-ovog pravila dekompozicije za sve promenljive xi, i $\in$ {1, ..., n}.

U GF(3) inverzna Vilenkin-Chrestenson-ova transformacija za n = 1 se može izraziti kroz ternarne promenljive kao: $f=1/3(1 \cdot Sf(0) + (1 + a1x + a2x^2)Sf(1) + (1+a1x + \bar{}a2x^2)Sf(2))$ za $Sf(0) = f(0) + f(1) + f(2)$, $Sf(1) = f(0) + \xi^2 f(1) + \xi f(2)$ i $Sf(2) = f(0) + \xi f(1) + \xi^2 f(2)$.

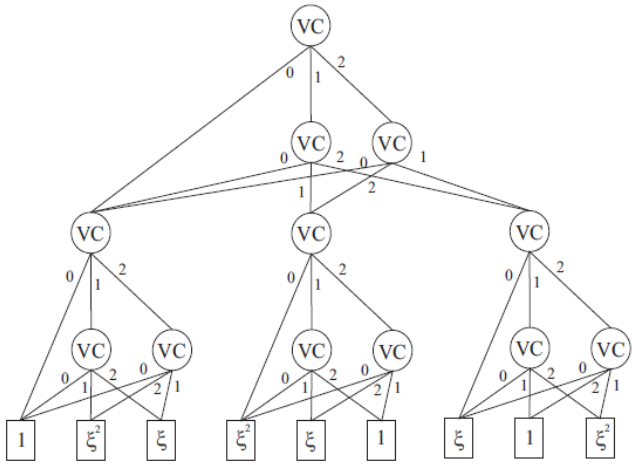


Slika 2. Vilenkin-Chrestenson dijagram odluke za $f=x_1x_2$

Kako bi se pročitale vrednosti funkcije, prelaze se moguće putanje na dijagramu, počev od terminalnih čvorova i završava se u korenu. Na svakom neterminalnom čvoru primenjena je obrnuta Vilenkin-Chrestenson dekompozicija. Izlazne grane svakog neterminalnog čvora odgovaraju Sf(0), Sf(1) i Sf(2) za nivoe 0, 1 i 2. Terminalni čvorovi Vilenkin-Chrestenson dijagrama odluke odgovaraju spektralnim koeficijentima date funkcije. Primeri Vilenkin-Chrestenson dijagram odluke dati su na slikama 3 i 4.



Slika 3. Vilenkin-Chrestenson dijagram odluke za $f_2 = f_1 \oplus_3 1$, gde je $f_1=x_1x_2$.



Slika 4. Vilenkin-Chrestenson dijagram odluke za $f=x_1x_2 \oplus_3 x_3x_4$.

2.2. Primene Vilenkin-Chrestenson transformacije

Vilenkin-Chrestenson transformacija se može posmatrati kao uopštena formulacija Walsh-ovih transformacija iz binarne u višeznačnu logiku. Stoga, ona se može koristiti u rešavanju različitih zadataka korišćenjem višeznačne logike umesto binarnih podataka.

Jedna od bitnijih primena Vilenkin-Chrestenson transformacija je u adaptivnoj obradi signala kroz konstrukciju generalizovanih talasnih paketa podataka. Ovaj pristup omogućava stvaranje detaljnih slika signala čija rezolucija varira u vremenu i frekvenciji.

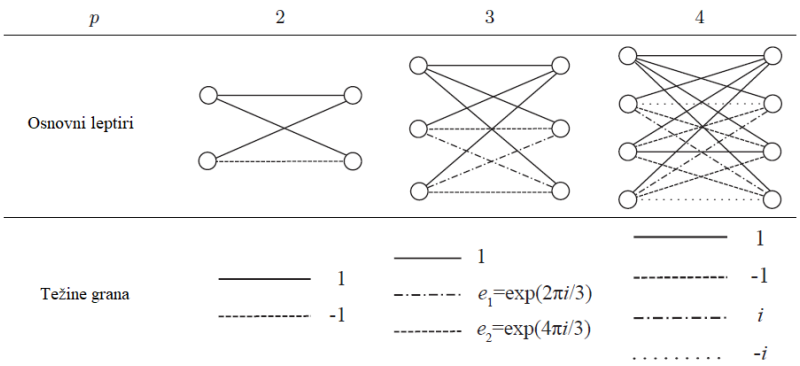
Algoritmi za otkrivanje grešaka (eng. error detection) i kompresiju podataka u komunkacionim kanalima se baziraju na funkcijama sa autokorelacijama. Autokorelacije se izračunavaju korišćenjem Wiener-Khinchin teoreme koja uključuje opsežno izračunavanje primenom Walsh spektra za binarne funkcije i Vilenkin-Chrestenson spektra za višeznačne logičke funkcije.

Vilenkin-Chrestenson transformacije su pronašle primenu i u metodama za spektralnu analizu mozaika. Mozaik je definisan kao obrazac stvoren periodičnim ponavljanjem uzorka tzv. seed-pattern. Mozaik struktura uzoraka i neki efekti buke mogu se analizirati koristeći Vilenkin-Chrestenson-ovu transformaciju. Na primer, ova metoda se može primeniti za otkrivanje grešaka u memorijama i FPGA-ima, budući da njihova infrastruktura ima mozaičnu strukturu ukoliko se ne posmatraju kontrolne i komunikacione linije.

U navedenim ali i sličnim primenama izračunanje Vilenkin-Chrestenson spektra je esencijalno za praktičnu implementaciju. Stoga, efikasna adaptacija Vilenkin-Chrestenson transformacija na GPU je jedna od značajnijih u pogledu proširenje njegove upotrebe u inženjerskoj praksi.

3. Brzi algoritmi za Vilenkin-Chrestenson transformaciju

Matrica Vilenkin-Chrestenson-ove transformacije može se faktorizovati na različite načine što predstvalja osnovu različitih algoritama brze transformacije (eng. fast transform algorithms). Dva najznačajnija algoritma su Cooley-Tukey i algoritmi konstantne geometrije. Na Slici 1. su predstavljeni osnovni leptiri za p vrednosti 2, 3 i 4. Elementima transformacione matrice odgovaraju težine grana predstavljene na slici 5.



Slika 5. Osnovni leptiri za Vilenkin-Chrestenson transformaciju za p=2,3,4

3.1. Cooley-Tukey algoritam

Cooley-Tukey algoritam je najčešće korišćen algoritam za izračunavanje brze Furijeove transformacije. Algoritam je prvi put objavljen 1969. od strane Džejmsa Kulija i Džona Tukija u članku “Brza Furijeova transformacija I njene primene"” (engl. The Fast Fourier Transformationa and its Applications). Algoritam je zasnovan na principu “zavadi pa vladaj” (engl. divide–and–conquer) i funkcioniše tako što izračunava diskretnu Furijeovu transformaciju cele ulazne sekvence koristeći diskretnu Furijeovu transformaciju podsekvenci ulazne sekvence.

Ovaj algoritam, uključujući njegovu rekurzivnu primenu, izmislio je Karl Fridrih Gaus oko 1805. godine, koji ga je koristio za interpolaciju trajektorija asteroida Palas i Juno, ali njegov rad nije bio široko prepoznatljiv. Međutim, Gaus nije analizirao asimptotsku vremensku složenost algoritma. Razne ograničene forme su takođe otkrivene nekoliko puta tokom 19. i početka 20. veka. FFT je postao popularan nakon što su Džejms Kuli iz IBM-a i Džon Tjuki sa Prinstona objavili rad 1965. godine u kome je algoritam detaljno opisan, kao i postupak kako ga praktično izvoditi na računaru.

Algoritmi za izračunavanje brze Furijeove transformacije se koriste za izračunavanje Furijeove transformacije za datu ulaznu sekvencu. Diskretna furijeova transformacija transformiše diskretni impulsni signal iz vremenskog domena u domen frekvencije. Brza Furijeova transformacija se koristi u mnogo različitih sfera, ukjučujući digitalnu obradu signala, telekomunikacije, i analizu zvučnog signala. Vremenska složenost algoritma brze Furijeove transformacije na standardnim mikroprocesorskim sistemima sa fon Nojmanovom arhitekturom je O(NlogN). Ubrzana varijanta algoritma radi u vremenskoj složenosti O(logN).

Ovaj algoritam za Vilenkin-Chrestenson-ovu transformaciju je zasnovan na Good-Thomas faktorizaciji koja potiče od Kronekerovog proizvoda matrice transformacije.

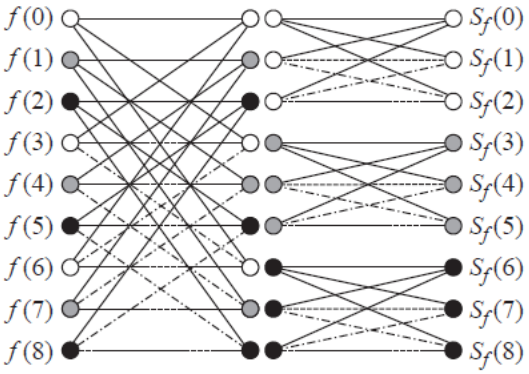
Primer izračunavanja za p=3 i n=2.
Vilenkin-Chrestenson matrica transformacija:

$$\mathbf{VC}_3^{-1}(2) = \frac{1}{9}(\mathbf{V}_1(2) \cdot \mathbf{V}_2(2))$$

$$\mathbf{V_1}(2) = \begin{bmatrix} 1 & 1 & 1 \\ 1 & e_2 & e_1 \\ 1 & e_1 & e_2 \end{bmatrix} \otimes \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 1 \\ 1 & 0 & 0 & e_2 & 0 & 0 & e_1 & 0 & 0 \\ 0 & 1 & 0 & 0 & e_2 & 0 & 0 & e_1 & 0 \\ 0 & 0 & 1 & 0 & 0 & e_2 & 0 & 0 & e_1 \\ 1 & 0 & 0 & e_1 & 0 & 0 & e_2 & 0 & 0 \\ 0 & 1 & 0 & 0 & e_1 & 0 & 0 & e_2 & 0 \\ 0 & 0 & 1 & 0 & 0 & e_1 & 0 & 0 & e_2 \end{bmatrix}$$

$$\mathbf{V_2}(2) = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \otimes \begin{bmatrix} 1 & 1 & 1 \\ 1 & e_2 & e_1 \\ 1 & e_1 & e_2 \end{bmatrix} = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & e_2 & e_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & e_1 & e_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & e_2 & e_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & e_1 & e_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & e_2 & e_1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & e_1 & e_2 \end{bmatrix}$$

Na Slici 6 je prikazan dijagram toka algoritma. Elementi koji učestvuju u formiranju leptira su označeni istom bojom kao i leptir. Budući da se izlazni podaci leptira čuvaju na istim mestima kao i njegovi ulazni podaci, i funkcija i spektar mogu se čuvati pomoću jednog niza što omogućava in-place implementaciju algoritma.



Slika 6. Cooley-Tukey FFT algoritam za Vilenkin-Chrestenson transformaciju za p=3, n=2

3.2. Constant Geometry algoritam

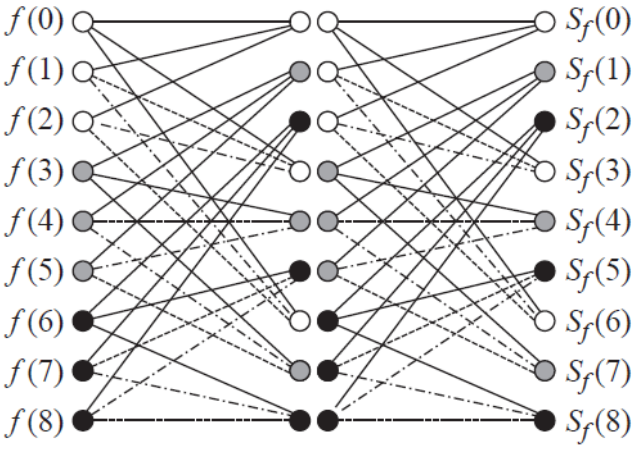
Implementacija Constant Geometry algoritma se oslanja na faktorizaciju transformacione matrice. Korišćenjem ovog algoritma omogućena je brža implementacija Vilenkin-Chrestenson transformacije na grafičkoj procesnoj jedinici u odnosu na korišćenje Cooley-Tukey algoritma. Brže izvršenje ove transformacije je omogućeno primenom pojednostavlje aritmetike adresa kod Constant Geometry algoritma što odgovara programiranju i internoj strukturi grafičke jedinice. Međutim, mana korišćenja ovog algortima u implementaciji Vilenkin-Chrestenson transformacije je u veličini i učestalosti memorijskih zahteva.

Primer izračunavanja za p=3 i n=2.
Vilenkin-Chrestenson matrica transformacija:

$$\mathbf{VC}_3^{-1}(2) = \frac{1}{9}\mathbf{A}(2)^2$$

$$\mathbf{A}(\mathbf{2}) = \begin{bmatrix} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 \\ 1 & e_2 & e_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & e_2 & e_1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & e_2 & e_1 \\ 1 & e_1 & e_2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & e_1 & e_2 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & e_1 & e_2 \end{bmatrix}$$

Na Slici 4. je prikazan graf toka koji odgovara faktorizaciji u navedenom primeru. Kako se u svakom koraku izvode identična izračunavanja, Constant Geometry algoritam je pogodan za VLSI implementaciju. Međutim, u ovoj implementaciji nije moguće upisivati podatke na istim memorijskim lokacijama na kojima se čuvaju funkcije unosa, pa je neophodna out-of-place implementacija što rezultira povećanjem meorijskog prostora.



Slika 7. Constant geometry FFT algoritam za Vilenkin-Chrestenson transformaciju za p=3, n=2

3.3. Razlike između Cooley-Tukey i Constant geometry algoritma

U Cooley-Tukey algoritmima i algoritmima zasnovanim na Good-Thomas faktorizaciji, matrice faktorizacije Ci su međusobno različite, međutim u svakom koraku algoritma se ponavlja pn-1 identičnih izračunavanja koji se nazivaju leptiri (eng. butterflies). Leptiri su proračuni definisani osnovnom GF-transformacionom matricom GGF(p) i u svakom koraku se menja skup podataka nad kojima se oni primenjuju. To podrazumeva da je u svakom koraku potrebno izračunati adrese lokacija podataka koji se koriste kao ulazni podaci za izračunavanje leptira. Udaljenost između memorijskih lokacija sa kojih se podaci čitaju je različita u svakom koraku i smanjuje se sa pn – 1 u prvo koraku na p u posednjem koraku. Dobra karakteristika je to da se privremeni rezultati izračunavanja nakon svakog koraka mogu sačuvati u istim memorijskim lokacijama iz kojih se podaci čitaju. Ova karakteristika omogućava in-place implementaciju Cooley-Tukey algoritama, što znači da se da se izlaz koraka u algoritmu čuva u istim lokacijama iz kojih se čitaju ulazni podaci. Stoga, memorija potrebna za izračunavanje spektra je identična onoj koja se koristi za čuvanje funkcije koja se obrađuje.

Glavna ideja algoritama konstantne geometrije je izvođenje operacija izračunavanja nad podacima preuzetih iz istih memorijskih lokacija u svakom koraku. Dakle, adrese memorijskih lokacija koje su dobijene u prvom koraku se koriste i u svim ostalim koracima za pribavljanje podataka. Posledica ove osobine algortima je ta da se privremeni rezultati ne mogu sačuvati u istim memorijskim lokacijama iz kojih se preuzimaju podaci. Ovakva implementacija onemogućava in-place izračunavanja i zahteva dvostruko više memorijskog prostora.

Razlika u adresnoj aritmetici je odlučujući faktor u izboru algoritma. U sledećem primeru je prikaz značaj adresne aritmetike u implemtaciji Cooley-Tukey i Constant geometry algoritama.

Primer 1:
Za p=3 i n=2 primenom Cooley-Tukey algoritma, u prvom i drugom koraku operacije se izvode nad ulaznim skupovima podataka {0, 3, 6}, {1, 4, 7}, {2, 5, 8} i {0, 1, 2}, {3, 4, 5}, {6, 7, 8}. Rezultati operacija se skaldište u istim memorijskim lokacijama.
U slučaju Constant geometry algoritma, u oba koraka operacije se primenjuju nad skupovima podataka na memorijskim lokacijama {0, 1, 2}, {3, 4, 5}, {6, 7, 8}, a rezultat je smešten na lokacijama {0, 3, 6}, {1, 4, 7}, {2, 5, 8}.

4. Implementacija FFT algoritama na GPU

Programiranje uz pomoć grafičke procesne jedinice (GPU), poznato i kao GPGPU (eng. General-purpose computing on graphics processing units) omogućava paralelizaciju sekvencijanog programa korišćenjem konkurentnih niti. Ovaj vid izračunavanja je omogućava korišćenje deljive memorije, čitanje sa proizvoljne adrese u memoriji, brži tok podataka iz i u GPU kao i podršku za operacije nad celobrojin podacima i bitovima.

Programi koji se izvršavaju paralelno se sastoje iz dva dela: host program koji se izvršava na procesoru (CPU) i device program koji se izvršava na grafičkom procesoru (GPU). Host progam, najčešće pisan C ili C++ programskim jezikom, kreira strukture podataka koje upravljaju komunikacijom host-device. Device program je u većini slučajeva kreiran korišćenjem OpenCL C (ili CUDA C) i implementira kernel funkcije.

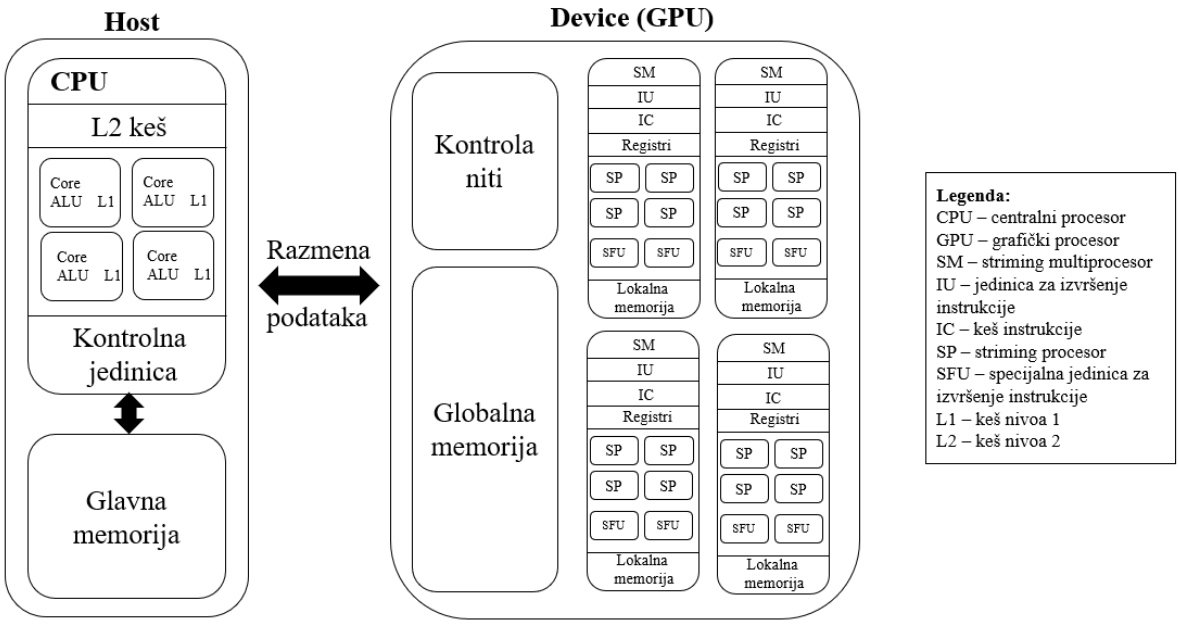
Uvođenje GPU programiranja stvorilo je novo hardversko okruženje sa specifičnim karakteristikama kojima bi trebalo prilagoditi postojeće FFT algoritme kako bi se pravilno i u potpunosti iskoristili raspoloživi resursi. Većina implementacija FFT algoritama na GPU-u koristi grafički API kao što je OpenGL ili DirectX. Međutim, ovi programi ne podržavaju grafiku, pristup deljivoj memoriji kao ni sinhronizaciju pristupa koji su dostupni na modernim grafičkim jedinicama. Performanse FFT algoritama mogu u mnogome zavisiti od organizacije memorijske jedinice i njenog korišćenja. Iako grafičke jedinice pružaju visok stepen paralelizma memorije, faza mešanja indeksa (eng. index-shuffling stage) FFT algoritama kao što je Cooley-Tukey može biti prilično skupa operacija zbog neskladnih pristupa memoriji.

4.1. Mapiranje Vilenkin-Chrestenson transformacije na GPU

Mapiranje na GPU predstavlja jedan od ključnih koraka u implementaciji algoritama. Ispravnim mapiranjem mogu se iskoristiti sve prednosti hardvera računarskog sistema i na taj način unaprediti izvršenje programa.

Mapiranje FFT algoritama podrazumeva da se sva izračunavanja uključena u leptire izvršavaju na device uređajima odnosno na GPU-u, dok se ostatak procedura izvršava na CPU-u, odnosno na host uređaju. Svaka nit izvršava radix-r leptir nad različitim ulaznim podacima, pa je stoga broj niti u svakom koraku jednak N=r gde je N=pn. Ovakva organizacija proračuna omogućava postojanje velikog broja niti koje izvode iste operacije nad različitim podacima istovremeno, što se dobro pokazalo na hardveru SIMD GPU.

Performanse brzih algoritama u velikoj meri zavise od organizacije memorije pa se efikasnim mapiranjem podataka na GPU memorijski podsistem postižu bolji rezultati u izvršenju FFT algoritama. Organizacija GPU memorijskom podsistema prikazana je na Slici 8. Apstraktni model GPU memorije sadrži četiri memorijske regije. Globalna memorija je najveće veličine (obično 1-2 GB), ali ima najsporiji pristup. Ova memorija ima pristup za čitanje/pisanje i čuva podatke za ceo uređaj. Stalna memorija deli isti memorijski region sa globalnom memorijom, ali sa dve glavne razlike: dostupne su samo operacije čitanja podataka i njen sadržaj može biti keširan samo u L1 keš memoriju, obično veličine 64 KB. Lokalna memorija je prisutna na svakom GPU-u i obično je veličine od 32-48 KB. To je memorija za čitanje i pisanje kojoj može pristupiti samo lokalni uređaj. Najčešće se koristi za čuvanje podataka koje koriste niti u procesima programa. Privatna memorija koristi region memorije datoteka sa brzim registrom. Svaka nit ima određeni broj registara za čuvanje svojih privatnih podataka. Na ovaj način ograničen je broj niti koje mogu istovremeno da budu aktivne što dovodi do poboljšanja performansi na grafičkim procesorima.



Slika 8. Organizacija sistema sa GPU

U svim preslikavanjima, vektor ulazne funkcije i Vilenkin-Chrestenson spektar se čuvaju u globalnoj memoriji. Registri se koriste samo za skladištenje operanada nad kojima niti izvršavaju operacije koje implementiraju leptir i pomoćnih promenljivih koje mogu da čuvaju informacije o vrednosti rastojanjima za trenutni korak ili transformaciju radiksa. Mapiranje Vilenkin-Chrestenson transformacije na GPU se može implementirati na dva načina:

- kada je poznata cela osnovna Vilenkin-Chrestenson matrica transformacija

$VC_p^{-1}(1)$;

- kada je prva Vilenkin-Chrestenson funkcija $X_1^{(p)}(z)$ poznata, a ostali elementi transformacione matrice se računaju.

Kako bi se ukazalo na prednost korišćenja GPU-a u implementaciji FFT algoritama izvršena je analiza mapiranja ovih algoritama na GPU koiršćenjem lokalne i glavne memorije. Ograničenja koja su uzeta u obzir su sledeća: svaki pristup GPU globalnoj memoriji može trajati najduže 800 klok ciklusa, izračunavanja su ograničena na 32 instrukcije po klok ciklusu pa je iz tog razloga brže ponovno računanje operanda u odnosu na njihovo stalno skladištenje, svako $X_w^{(p)}(z)$ gde je w različito od nule uzima sve moguće vrednosti elemenata VC_p^{-1}. Stoga je dovoljno čuvati na primer $X_w^{(p)}(1)$ i odrediti ostale elemente transformacione matrice.

Stalna memorija se kešira u brzu keš memoriju i ona može emitovati sadržaj određene lokacije kada sve niti pristupaju istoj memorijskoj lokaciji. Kako nije potrebno modifikovati $X_1^{(p)}(z)$ ili $VC_p^{-1}(1)$ posle inicijalizacije, upotreba konstantne memorije potencijalno može dati bolje performanse nego korišćenje lokalne memorije.

Takođe, izbor algoritma definiše neophodna mapiranja, pa je zato predstavljenja analiza mapiranja Cooley-Tukey i Constant geometry algoritama na GPU. U GPU implementaciji Cooley-Tukey algoritma se koristi indeks *osnova* mapiranja memorijske adrese koja sadrži prvi operand za svaku nit na sledeći način:

$$osnova \leftarrow id\_niti \bmod rastojanje + r*rastojanje * (id\_niti / rastojanje)$$

gde je *id_niti* pozitivan ceo broj koji identifikauje nit, *r* je radix transformacije, a *rastojanje* odgovara rastojanju između dva elementa koja se koriste u istom koraku izračunavanja leptira. U implementaciji Cooley-Tukey algoritma u k-tom koraku *rastojanje* = p^{n-k}. Memorijske adrese ostalih operanada se računaju kao *osnova + i * rastojanje* gde *i* ima vrednosti od 0 do r-1. Rezultati izračunavanja se skladište u istim memorijskim lokacijama kao i operandi (in-place implementacija).

U GPU implementaciji algoritma konstantne geometrije, indeks *osnova* mapiranja memorijske adrese se može izračunati na sledeći način:

$$osnova \leftarrow id\_niti * r.$$

Lokacije ostalih operanda se izračunavaju kao *osnova + i*, a lokacije za čuvanje rezultata kao *threadID + i (N/r)*, gde je i = 0,1,...r-1. Kako se lokacije ulaznih i izlaznih podataka razlikuju, izvršena je out-of-place implementacija.

U algoritmu konstantne geometrije memorijske lokacije operanada su nezavisne od trenutnog koraka u transformaciji. Aritmetičke operacija za izračunavanje adresa operanada su jednostavnije, iako ovaj algoritam zahteva dodatne proračune za čuvanje podataka. Na hardverskom nivou određene operacije, kao što su sabiranje i množenje, su implementirane direktno pa se u tom obliku mogu korisititi dok ostale operacije poput moduo operacija ili deljenja zahtevaju izvršenje više instrukcija. Kao rezultat, kada se *kernel* programa konstantne geometrije kompajlira za GPU, prevodi se u manji broj mašinskih instrukcija u odnosu na Cooley-Tukey algoritam. Na primer, radix-3 program za Cooley-Tukey algoritam se prevodi u 34, a za Constant geometry algoritam u 19 mašinskih instrukcija. Slično i za radix-8 postoji 159, odnosno 108 mašinskih uputstava za oba ova algoritma. Ovo svojstvo algoritma konstantne geometrije može biti korisno kod manje moćnih grafičkih jedinica.

5. Implementacija Vilenkin-Chrestenson transformacije na GPU

Program pisan za sistem sa grafičkom procesnom jedinicom se sastoji od dela koje se izvršva na host uređaju i dela namenjenom device uređaju. U ovom poglavlju biće predstavljeni i analizirani host i device programi za izračunavanje Vilenkin-Chrestenson transformacije na GPU.

5.1. Host program

Algoritam 1 predstavlja pseudokod host programa za izračunavanje Vilenkin-Chrestenson transformacije.

Algoritam 1: Host program

1: **if** p prost **then**
2: Izračunaj $X_1^{(p)}(z)$ ili $VC_p^{-1}(1)$ koji odgovaraju cikličnoj grupi Cp
3: **else** {p složen}
4: Izračunaj ili pribavi faktorizaciju za p unesenu od strane korisnika
5: Izračunaj $X_1^{(p)}(z)$ ili $VC_p^{-1}(1)$
6: **end if**
7: Kreiraj structure podataka u deljivoj memoriji
8: Upiši vector funkcije $X_1^{(p)}(z)$ (ili $VC_p^{-1}(1)$) na device uređaj
9: Izvrši kernel program sa N/r niti koje se izvršavaju paralelno
10: Pročitaj Vilenkin-Chrestenson spektar sa device uređaja i prebaci na host uređaj

Host program prihvata funkciju sa vrednošću p, predstavljenu njenim vektorom funkcije, za bilo koju vrednost p koja je pozitivan ceo broj. Ukoliko je p prosto onda se računa prva Vilenkin-Chrestenson funkcija $X_1^{(p)}(z)$ ili cela osnovna matrica transformacija $VC_p^{-1}(1)$, u zavisnosti od mapiranja. Ukoliko je p složeno, može se izvršiti faktorizacija funkcije p ili prihvatiti faktorizacija koja je prosleđena od strane korisnika, a zatim izvršiti transformacija. Ovi zadaci se izvode kao koraci 1 do 6 Algoritma 1. Korak 7 algoritma je sastavni deo svih GPU računarskih programa i uključuje stvaranje struktura podataka neophodnih za obradu rezultata dobijenih od device uređaja.

Komunikacija između host i device uređaja korišćenjem PCIe magistrale najviše umanjuje GPU performanse, pa je iz tog razloga prenos podataka obavljen pomoću bafera u tzv. page-locked (pinned) memoriji umesto korišćenja standardnih operacija čitanje i upisa podataka. Korišćenje pinned memorije sprečava straničenje memorije od strane operativnog sistema što garantuje da je memorijski prostor kontinualan i smešten na disku na uzastopnim memorijskim lokacijama. Dodatno, procesor kontroliše i ograničava upotrebu sistemske memorije kao pinned memorije kako bi se ostavilo dovoljno memorijskih resursa za ostale procese koji se izvodi u sistemu.

Kod Constant geometry algoritma, koji je implementiran out-of-place, zamena argumenata se koristi kako bi se minimizovalo zauzeće propusnog opsega. Dodata je i promenljiva koja sadrži broj koraka algoritma i u svakom koraku se vrši zamena pokazivača na ulazni i izlazni bafer. Bafer koji čuva rezultat izračunavanja u neparnim koracima postaje bafer za čitanje podataka u parnim brojevima i obrnuto. Nakon svih izračunavanja, podaci se

čitaju samo iz bafera koji sadrži rezultujući spektar. Na ovaj način, postojanje dva bafera u GPU globalnoj memoriji u algoritmu konstantne geometrije povećava memorijski prostor ali ne povećava zauzetost propusnog opsega GPU-a u poređenju sa implementacijom Cooley-Tukey algoritma.

5.2. Device program

Pseudokodovi device uređaja koji odgovaraju Cooley-Tukey i Constant geometry algoritmu za Vilenkin-Chrestenson-ovu transformaciju su predstavljeni algoritmima 2 i 3. U ovim implementacijama samo prva Vilenkin-Chrestenson funkcija $X_1^{(p)}(z)$ se prenosi u GPU konstantnu memoriju dok se ostali elementi transforamcione matrice računaju na način objašnjen ranije. Jedan ulazno-izlazni bafer u globalnoj memoriji se koristi za implementaciju Cooley-Tukey algoritma, dok se u algoritmu konstantne geometrije koriste zasebni baferi za ulazne i izlazne tokove podataka. Niz *vc* u GPU konstantnoj memoriji čuva vrednosti za $X_1^{(p)}(z)$. Linije 2, 5 i 12 u algoritmima 2 i 3 implementiraju adresnu aritmetiku. Promenljiva *acc* se koristi za skladištenje međurezultata.

Program device uređaja opisuje posao koji izvršava jedna nit što u implementaciji Vilenkin-Chrestenson transformacije predstavlja operacije izračunavanja jednog radix-r leptira. Broj niti za svaki korak algoritma je N=r na nivou programa, međutim stvaran broj niti koje su istovremeno aktivne zavisi od korišćenja grafičke procesne jedinice. Niti sa uzastopnim identifikatorima pristupaju uzastopnim lokacijama u globalnoj memoriji GPU-a čime je omogućen istovremeni višestruki pristup memoriji.

Algoritam 2: Device program Cooley-Tukey algoritma

1: *id_niti* ← uzima globalni identifikator niti
2: *osnova* ← *id_niti* mod *rastojanje* + *r***rastojanje**(*id_niti*/*rastojanje*)
3: *acc* ← 0
4: **for** i ← 0 **to** r - 1 **do**
5: *op*[*i*] ← *inout*[*osnova* + *i***rastojanje*]
6: **end for**
7: **for** i ← 0 **to** r - 1 **do**
8: **for** j ← 0 **to** r - 1 **do**
9: *k* ← (*r* – *i***j*) mod *r*
10: *acc.re* ← *acc.re* + *vc*[*k*].*re* * *op*[*j*].*re* – *vc*[*k*].*im* * *op*[*j*].*im*)
acc.im ← *acc.im* + *vc*[*k*].*re* * *op*[*j*].*im* – *vc*[*k*].*im* * *op*[*j*].*re*)
11: **end for**
12: *inout*[*osnova* + *i***rastojanje*] = *acc*
13: *acc* ← 0
14: **end for**

Dinamička alokacija memorije unutar GPU kernela nije moguća, pa je iz tog razloga neophodno je korišćenje biblioteka kernela za izračunavanje transformacija ili korišćenje kompajlerskih opcija kao što je u OpenCL-u C justintime koplajler koje omogućavaju kreiranje proizvoljnog radix-r leptira. Za predstavljanje Vilenkin-Chrestenson funkcije i kreiranje rezultujućeg spektra koristi se float2 vector kao tip podatka čime je omogućena efektivna upotreba propusnog opsega GPU memorije.

Algoritam 3 predstavlja pseudokod device programa za implementaciju Vilenkin-Chrestenson transformacije. Ova implementacija podrazumeva odvojene strukture podataka za čuvanje ulazne i izlazne operanada. Niz *vc* u konstantnoj memoriji zadržava vrednosti prve Vilenkin-Chrestenson funkcije, dok se ostatak matrice izračunava u koraku 9 Algoritma 3. Stalna GPU memorija deli isti prostor sa globalnom memorijom ali su dozvoljene samo operacije čitanja podataka i mogućnost keširanaj u brzu memoriju.

Algoritam 3: Device program algoritma konstantne geometrije

1: *id_niti* ← uzima globalni identifikator niti
2: *osnova* ← *id_niti* * *r*
3: *acc* ← 0
4: **for** i ← 0 **to** r - 1 **do**
5: *op*[*i*] ← *inout*[*osnova* + *i*]
6: **end for**
7: **for** i ← 0 **to** r - 1 **do**
8: **for** j ← 0 **to** r - 1 **do**
9: *k* ← (*r* – *i***j*) mod *r*
10: *acc.re* ← *acc.re* + *vc*[*k*].*re* * *op*[*j*].*re* – *vc*[*k*].*im* * *op*[*j*].*im*)
acc.im ← *acc.im* + *vc*[*k*].*re* * *op*[*j*].*im* – *vc*[*k*].*im* * *op*[*j*].*re*)
11: **end for**
12: *output*[*id_niti* + *i*(N/r)*] = *acc*
13: *acc* ← 0
14: **end for**

Device programi pisani za Cooley-Tukey i algoritam konstantne geometrije se razlikuju jedino u linijama 2, 5 i 12 Algoritama 2 i 3 u kojima su implementirane operacije preuzimanja i čuvanja podataka na osnovu odgovarajuće aritmetike adrese. Takođe, ova dva programa se prevode u različit broj GPU instrukcija i koriste različit broj registara. Ova razlika se može jasno uočiti na primeru OpenCL implementacije Cooley-Tukey i algoritama konstantne geometrije za p = 4 i r = 16. AMD APP Profiler 2.5 pokazuje da se, kada se odgovarajuća GPU jezgra kompajliraju za AMD Evergreen GPU arhitekturu, jezgro OpenCL C bazirano na algoritmu Cooley-Tukey prevodi u 369 ALU operacija i 16 operacija za preuzimanje podataka. Ovo zahteva 22 registra za svaku GPU nit. Za jezgro OpenCL C koji nadopunjuje algoritam konstantne geometrije, broj ALU operacija je 344, sa 8 operacija za preuzimanje podataka. Za ovo jezgro, 14 registara po niti je dovoljno.

Uticaj artimetike adresa u implementaciji Vilenkin-Chrestenson transformacije prikazan je Tabeli 1 na primeru Cooley-Tukey i algoritma konstantne geometrije. Razlika između ovih algoritama je predstavljena brojem ALU i fetch operacijama kao i brojem registara koji su korišćeni za svaku nit.

Na osnovu Tabele 1 može se zaključiti da aritmetika adrese značajno utiče na broj operacija i registara koji se koriste u implementaciji na GPU-u. Algoritam konstantne geometrije zahteva manje ALU operacija zahvaljujući jednostavnijim izračunavanjima neophodnim za pribavljanje podataka. Takođe, u implementaciji algoritma konstantne geometrije se koristi manji broj registara što je posledica pribavljanja dva operanda u jednoj 16b instrukciji.

r	3	4	9	16
Broj ALU instrukcija				
Cooley-Tukey	34	41	143	369
Constant geometry	20	24	118	344
Broj fetch instrukcija				
Cooley-Tukey	3	4	9	16
Constant geometry	3	2	9	8
Broj korišćenih registara				
Cooley-Tukey	6	9	14	22
Constant geometry	6	7	12	14

Tabela 1. Uporedni prikaz broja ALU, fetch operacija i broja korišćenih registara po niti za Cooley-Tukey i Constant geometry algoritam

U tabelama 2 i 3 data su vremena izvršavanja Vilenkin-Chrestenson transformacije za različite vrednosti p (p=3 i p=4). Kao platforme korišćenu su Nvidia i AMD CPU i GPU procesori. Na osnovi ovih tabela može se zaključiti da algoritam konstantne geometrije koji se izvršava na AMD procesoru daje najbolje performanse.

p=3	Broj ulaznih promenljivih n				
Platforma/Procesor/Algoritam	12	13	14	15	16
Nvidia / GPU / Cooley-Tukey	1	4	11	35	103
Nvidia / GPU / Constant geometry	1	4	11	36	112
Nvidia / CPU	12	49	172	562	1762
AMD / GPU/ Cooley-Tukey	5	17	58	175	558
AMD / GPU/ Constant geometry	5	16	55	168	525
AMD / CPU	46	180	577	1825	5866

Tabela 2. Vreme izvršenja Vilenkin-Chrestenson transformacije u ms za p=3

p=4	Broj ulaznih promenljivih n				
Platforma/Procesor/Algoritam	9	10	11	12	13
Nvidia / GPU / Cooley-Tukey	1	2	7	30	125
Nvidia / GPU / Constant geometry	1	2	8	35	137
Nvidia / CPU	8	30	125	546	2403
AMD / GPU/ Cooley-Tukey	2	8	39	162	663
AMD / GPU/ Constant geometry	2	7	37	156	622
AMD / CPU	31	171	764	3432	15428

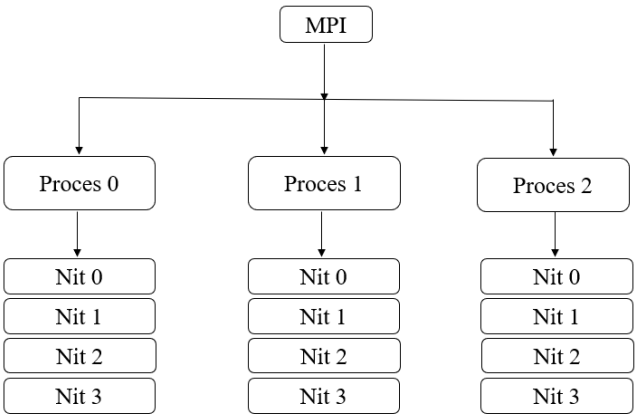
Tabela 3. Vreme izvršenja Vilenkin-Chrestenson transformacije u ms za p=4

6. Izračunavanje Vilenkin-Chrestenson spektra kvaternarnih funkcija korišćenjem MPI framework-a

U ovom radu izvršena je implementacija Vilenkin-Chrestenson spektra kvaternarnih funkcija. Aplikacija je kreirana korišćenjem MPI tehnologije, a kao razvojno okruženje korišćen je Visual Studio 2019. Izvorni kod programa je dostupan na Bitbucket repozitorijumu *https://bitbucket.org/tina98/VCspektar/src/master/*.

6.1. Pregled korišćene tehnologije

MPI (eng. Message Passing Interface) framework je interfejs za prosleđivanje poruka. Definisan je standard koji opisuje sintaksu i semantiku jezgra bibliotečkih rutina korisnih širokom krugu korisnika koji pišu programe za prosleđivanje poruka C, C ++ ili Fortran jezikom. Upotreba MPI-a omogućava implementaciju point-to-point i grupne komunikacije između procesa. Point-to-point operacije su posebno od koristi u uzorkovanoj ili neregularnoj komunikaciji, na primer, u paralelnoj arhitekturi podataka u kojoj svaki procesor rutinski zamenjuje regione podataka sa određenim procesorima ili u master-slave arhitekturama kada master proces šalje instrukcije ostalim procesima u sistemu. Suprotno od point-to-point operacija su grupne operacije gde jedan proces određenom podskupu procesa šalje neophodne podatke.



Slika 9. Prikaz MPI organizacije izvršenja programa

U projektu je korišćena Microsoft implementacija MPI standarda Microsoft MPI (MS-MPI) za kreiranje i upravljanje paralelnim procesima u implementaciji Vilenkin-Chrestenson spektra kvaternarnih funkcija.

6.2. Korišćene MPI funkcije u implementacija Vilenkin-Chrestenson spektra kvaternarnih funkcija

U implementaciji Vilenkin-Chrestenson spektra korišćene su grupne operacije i point-to-point operacije. Korišćene point-to-point operacije su:

- MPI_Send(void* *podaci*, int *broj_podataka*, MPI_Datatype *tip_podataka*, int *odrediste*, int *oznaka*, MPI_Comm *komunikator*) – podrazumeva slanje podataka sa adrese *podaci*, ukupan broj podataka koji se šalje je *broj_podataka* tipa *tip_podataka*. Podaci se salju kroz *komunikator* sa oznakom poruke *oznaka* i proces kome se podaci šalju je definisan parametrom *odrediste*.

- MPI_Recv(void* *podaci*, int *broj_podataka*, MPI_Datatype *tip_podataka*, int *izvor*, int *oznaka*, MPI_Comm *komunikator*, MPI_Status* *status*) – podrazumeva prijem podataka na adrese *podaci*, ukupan broj podataka koji se šalje je *broj_podataka* tipa *tip_podataka*. Podaci se salju kroz *komunikator* sa oznakom poruke *oznaka*, proces koji šalje podatke je definisan parametrom *izvor*, dok je status poruke određen parametrom *status*.

Grupne operacije korišćene u projektu su:

- MPI_Bcast(void **podaci*, int *broj_podataka*, MPI_Datatype *tip_podataka*, int *koren*, MPI_Comm *komunikator*) – podrazumeva slanje podataka na adrese *podaci*, ukupan broj podataka koji se šalje je *broj_podataka* tipa *tip_podataka*. Podatke šalje proces sa rankom *koren*, dok svi procesi u komunikatoru vrše prijem.
- MPI_Scatter(const void ***podaci_S*, int *broj_podataka_S*, MPI_Datatype *tip_podataka_S*, void * *podaci_P*, int *broj_podataka_P*, MPI_Datatype *tip_podataka_P*, int *koren*, MPI_Comm *komunikator*) - i-ti segment bafera se salje i-tom rocesu u grupi gde su segmenti iste velicine
- MPI_Gather(const void ***podaci_S*, int *broj_podataka_S*, MPI_Datatype *tip_podataka_S*, void * *podaci_P*, int *broj_podataka_P*, MPI_Datatype *tip_podataka_P*, int *koren*, MPI_Comm *komunikator)* - kreira svoj bafer skupljajuci od ostalih tako što i-ti podatak odgovaram i-tom procesu. Segment i se odredjuje na osnovu ranka procesa.

Za inicijalizaciju MPI procesa iskorišćena je funkcija MPI_Init, a za oznaku kraja izvršenja programa funkcija MPI_Finalize. Definisanje ranka procesa određeno je funkcijom MPI_Comm_rank, dok je ukupan broj procesa određen primenom funkcije MPI_Comm_size.

6.3. Prosleđivanje parametara i definisanje pomoćnih promenljivih u programu

Programu se kao paramtar prosleđuje kvaternarna funkcija predstavljena vektorom istinitosti. Na osnovu ovog vektora i korišćenjem matrica transformacije određuje se Vilenkin-Chrestenson spektar funkcije.

U programu su dodatno definisane promenljive koje su neophodne u izračunavanjima unutar svakog procesa. Sve kreiranje promenljive se nalaze u lokalnoj memoriji svake niti, dok postoje i globalno definisane promenljive poput promenljive Pi koja definiše vrednost broja Pi na 3.14 i promenljiva n koja označava broj promenljivih u funkciji. U programu je ona definisana na 4.

U lokalnoj memoriji svake niti nalaze se sledeće promenljive:

- int size – celobrojna promenljiva koja određuje ukupan broj aktivnih procesa;
- int rank – celobrojna promenljiva koja određuje rank procesa;
- int** rez – celobrojna matrica koja predstavlja osnovnu Vilenkin-Chrestenson matricu transformacije;
- int konacni_rez[16][16] – celobrojna matrica dobijena kao rezultat Kronekerovog proizvoda Vilenkin-Chrestenson matrica transformacije;
- int transponovana_mat[16][16] – matrica dobijena primenom operacije transponovanja nad matricom konacni_rez;
- int r[4][4], part_rez[4][4], rez – pomoćne matrice;

- int elMat – element matrice koji se prosleđuje svakom procesu u procesu određivanja Kronekerovog proizvoda;
- int vektor[16], int lok_vrsta[16], int lok_rez – pomoćni nizovi upotrebljeni u množenju transponovane matrice transponovana_mat sa unesenim vektorom funkcije i
- int spektar[16] – celobrojni niz koji sadrži Vilenkin-Chrestenson spektar zadatake kvaternarne funkcije.

6.4. Implementacija korišćenih funkcija u programu

U programu je implementirana funkcija za određivanje osnovne Vilenkin-Chrestenson matrice transformacije. Funkciji se kao parametar prosleđuje celobrojna p vrednost zadatake kvaternarne funkcije, a kao rezultat funkcija vraća matricu koja predstavlja Vilenkin-Chrestenson matrice transformacije. Elementi rezultujuće matrice su određeni primenom formule:

$$e^{2*\pi*i*k/p}$$

gde je i indeks vrste matrice, a k se izračunava primenom formule:

$$k = (p - i * j) \bmod p,$$

gde je j indeks kolone transformacione matrice.

```
int** compute_VC_transformMatrix(int p)
{
    int** transformMatrix;
    transformMatrix = new int* [p];
    for (int i = 0; i < p; i++)
        transformMatrix[i] = new int[p];

    int k;
    for(int i=0; i<p; i++)
        for (int j = 0; j < p; j++)
        {
            k = (p - i * j) % p;
            transformMatrix[i][j] = exp(2 * Pi * i * k / p);
        }

    return transformMatrix;
}
```

Slika 10. Implementacija funkcije za određivanje matrice Vilenkin-Chrestenson transformacije

U glavnoj (main()) funkciji programa vrši se deklaracija promenljivih za svaku nit, inicijalizacija MPI procesa, određivanje ukupnog broja procesa i ranka svakog od aktivnih procesa (Slika 11).

```
MPI_Init(&argc, &argv);
MPI_Comm_rank(MPI_COMM_WORLD, &rank);
MPI_Comm_size(MPI_COMM_WORLD, &c);
```

Slika 11. Inicijalizacija MPI procesa, određivanje ranka procesa i ukupnog broja procesa

Master proces (proces sa rankom nula) poziva funkciju compute_VC_transformMatrix (int p) za p vrednost 4 čime je dobijena osnovna Vilenkin-Chrestenson matrica transformacije. Ovaj deo programa se izvršava sekvencijalno. (Slika 12).

```
rez = compute_VC_transformMatrix(n);
```

Slika 12. Poziv funkcije compute_VC_transformMatrix u master procesu

Sledeći korak u implementaciji Vilenkin-Chrestenson spektra se izvršava paralelno i odnosi se na određivanje Kronekerovog proizvoda matrica transformacije. Operacija množenja je implementirana korišćenjem funkcije MPI_Bcast kojom se šalje matrica transformacije svim aktivnim procesima iz procesa sa rankom nula, a zatim se koristi funkcija MPI_Scatter kojom se šalje po jedan element matrice transformacije neophodan u određivanju parcijalnih proizvoda (Slika 13).

```
MPI_Bcast(&r, 9, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Scatter(&r, 1, MPI_INT, &elMat, 1, MPI_INT, 0, MPI_COMM_WORLD);
```

Slika 13. Slanje elemenata matrice korišćenjem grupnih operacija

Kada je svaki od procesa primio neophodne podatke za parcijalno izračunavanje konačne matrice, vrši se množenje dobijenog elementa iz matrice transformacije i svih elemenata u matrici (Slika 14).

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < n; j++)
    {
        part_rez[i][j] = elMat * r[i][j];
        printf("%d ", part_rez[i][j]);
    }
}
```

Slika 14. Množenje matrice transformacije

Svaki od procesa učestvuje u formiranju konačnog rezultata. Parcijalni rezultati množenja matrice se korišćenjem funckije MPI_Gather sažimaju u master procesu (Slika 15).

```
MPI_Gather(&part_rez, 4, MPI_INT, konacni_rez, 4, MPI_INT, 0, MPI_COMM_WORLD);
```

Slika 15. Kreiranje konačne matrice u master procesu

Ovako dobijena matrica se transponuje kako bi se izvršilo množenje sa prosleđenim vektorom funkcije. Ovaj deo programa se izvršava sekvencijalno u master procesu (Slika 16).

```
for (int i = 0; i < n*n; ++i)
    for (int j = 0; j < n*n; ++j) {
        transponovana_mat[j][i] = konacni_rez[i][j];
    }
```

Slika 16. Transponovanje matrice

Sledeći deo implementacije se izvršava paralelno i odnosi se na množenje prethodno dobijene matrice sa prosleđenim vektorom funkcije. Svaka vrsta matrice se prosleđuje jednom procesu korišćenjem funkcije MPI_Scatter, dok se korišćenjem funkcije MPI_Bcast prosleđuje ceo vektor svakom procesu (Slika 17).

```
MPI_Scatter(&transponovana_mat[0][0], n * n, MPI_INT, lok_vrsta, n * n, MPI_INT, 0, MPI_COMM_WORLD);
MPI_Bcast(vektor, n * n, MPI_INT, 0, MPI_COMM_WORLD);
```

Slika 17. Prosleđivanje vektora funkcije i vrste matrice transformacije svakom procesu

Svaki proces izvršava operacije množenja nad dobijenim elementima vektora funkcije i transponovane matrice transformacije (Slika 18).

```
lok_rez = 0;
for (int i = 0; i < n * n; i++)
    lok_rez += lok_vrsta[i] * vektor[i];
```

Slika 18. Izvršavanje operacija množenja u okviru svakog procesa

Parcijalni rezultati dobijeni u svakom procesu se prosleđuju master procesu funkcijom MPI_Gather gde se vrši štampanje dobijenog spektra (Slika 19).

```
MPI_Gather(&lok_rez, 1, MPI_INT, &spektar[0], 1, MPI_INT, 0, MPI_COMM_WORLD);
if (rank == 0)
{
    printf("Spekatr:\n");
    for (int i = 0; i < n * n; i++)
    {
        printf("%d ", spektar[i]);
    }

}
```

Slika 19. Određivanje Vilenkin-Chrestenson spektra i njegovo štampanje u master procesu

Na kraju programa pozvana je funkcija MPI_Finalize koja poništava sve prethodno pokrenute procese (Slika 20).

```
MPI_Finalize();
```

Slika 20. Poziv funkcije MPI_Finalize na kraju programa

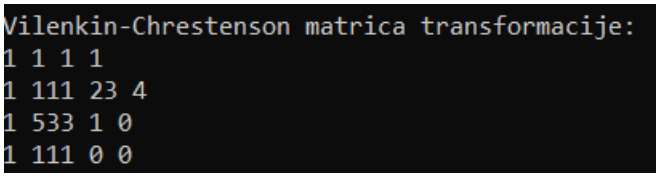
Pored opisanog postupka implementacije Vilenkin-Chrestenson spektra izvršene su i funkcije štampanja međurezultata.

6.5. Rezultati projekta

Izvršenje programa je testirano sa različitim brojem aktivnih procesa. U prvom slučaju broj aktivnih procesa je 16, dok je u drugom slučaju broj aktivnih procesa 4. Na osnovu performasni izvršenja utvrđeno je da se brže izvršava program sa 16 aktivnih procesa. Programu su prosleđeni i različiti vektori funkcija i analiza dobijenih rezultata je data u nastavku.

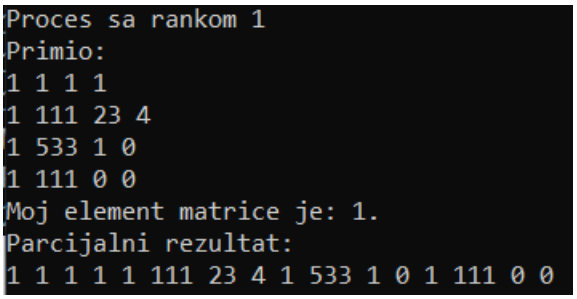
- Rezultati izvršenja programa za vektor funkcije [0,1,2,3,1,2,3,2,1,1,1,0,0,1,0,1]

Primenom funkcije compute_VC_transformMatrix dobijena je osnovna Vilenkin-Chrestenson matrica transformacije (Slika 21).



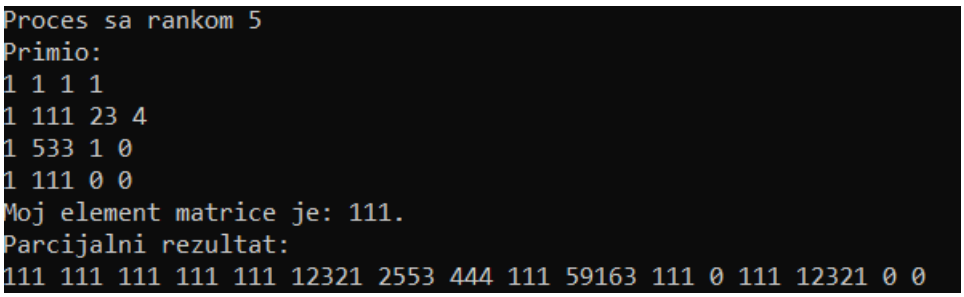
Slika 21. Vilenkin-Chrestenson matrica transformacije

Program izvršava 16 aktivnih procesa. Svakom procesu je prosleđenja matrica transformacije i jedan njen element kako bi svi procesi zajedno implementirali Kronekerov proizvod matrica transformacije. Na slici 22 prikazani su podaci koje je dobio proces sa rankom 1 kao i rezultat izvršenja operacija u ovom procesu.



Slika 22. Rezultat izvršenja programa procesa sa rankom 1

Na slici 23 prikazani su podaci koje je dobio proces sa rankom 5 kao i rezultat izvršenja operacija u ovom procesu.



Slika 23. Rezultat izvršenja programa procesa sa rankom 5

Na slici 24 prikazani su podaci koje je dobio proces sa rankom 6 kao i rezultat izvršenja operacija u ovom procesu.

```
Proces sa rankom 6
Primio:
1 1 1 1
1 111 23 4
1 533 1 0
1 111 0 0
Moj element matrice je: 23.
Parcijalni rezultat:
23 23 23 23 23 2553 529 92 23 12259 23 0 23 2553 0 0
```

Slika 24. Rezultat izvršenja programa procesa sa rankom 6

Nakon primene Kronekerovog proizvoda nad matricama transformacije, dobijena konačna matrica je smeštena u procesu sa rankom 0 (Slika 25).

```
Matrica:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 111 23 4 1 111 23 4 1 111 23 4 1 111 23 4
1 533 1 0 1 533 1 0 1 533 1 0 1 533 1 0
1 111 0 0 1 111 0 0 1 111 0 0 1 111 0 0
1 1 1 1 111 111 111 111 23 23 23 23 4 4 4 4
1 111 23 4 111 12321 2553 444 23 2553 529 92 4 444 92 16
1 533 1 0 111 59163 111 0 23 12259 23 0 4 2132 4 0
1 111 0 0 111 12321 0 0 23 2553 0 0 4 444 0 0
1 1 1 1 533 533 533 533 1 1 1 1 0 0 0 0
1 111 23 4 533 59163 12259 2132 1 111 23 4 0 0 0 0
1 533 1 0 533 284089 533 0 1 533 1 0 0 0 0 0
1 111 0 0 533 59163 0 0 1 111 0 0 0 0 0 0
1 1 1 1 111 111 111 111 0 0 0 0 0 0 0 0
1 111 23 4 111 12321 2553 444 0 0 0 0 0 0 0 0
1 533 1 0 111 59163 111 0 0 0 0 0 0 0 0 0
1 111 0 0 111 12321 0 0 0 0 0 0 0 0 0 0
```

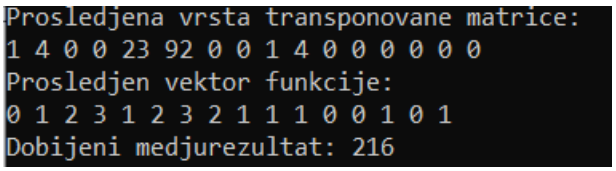
Slika 25. Matrica dobijena primenom Kronekerovog proizvoda nad matricama transformacije

Kako bi se izvračunao Vilenkin-Chrestenson spektar zadate funkcije neophodno je vektor funkcije pomnožiti sa transponovanom matricom koja je dobijena u prethodnom koraku. Iz ovog razloga u master procesu se vrši transponovanje dobijene matrice (Slika 26).

```
Transponovana matrica:
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1
1 111 533 111 1 111 533 111 1 111 533 111 1 111 533 111
1 23 1 0 1 23 1 0 1 23 1 0 1 23 1 0
1 4 0 0 1 4 0 0 1 4 0 0 1 4 0 0
1 1 1 1 111 111 111 111 533 533 533 533 111 111 111 111
1 111 533 111 111 12321 59163 12321 533 59163 284089 59163 111 12321 59163 12321
1 23 1 0 111 2553 111 0 533 12259 533 0 111 2553 111 0
1 4 0 0 111 444 0 0 533 2132 0 0 111 444 0 0
1 1 1 1 23 23 23 23 1 1 1 1 0 0 0 0
1 111 533 111 23 2553 12259 2553 1 111 533 111 0 0 0 0
1 23 1 0 23 529 23 0 1 23 1 0 0 0 0 0
1 4 0 0 23 92 0 0 1 4 0 0 0 0 0 0
1 1 1 1 4 4 4 4 0 0 0 0 0 0 0 0
1 111 533 111 4 444 2132 444 0 0 0 0 0 0 0 0
1 23 1 0 4 92 4 0 0 0 0 0 0 0 0 0
1 4 0 0 4 16 0 0 0 0 0 0 0 0 0 0
```

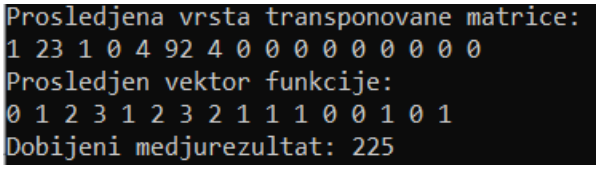
Slika 26. Dobijena transponovana matrica

Sledeći korak u implementaciji je množenje dobijene transponovane matrice i prosleđenog vektora funkcije. Ovaj deo programa je realizovan paralelno. Primenom funkcija MPI_Scatter i MPI_Bcast se šalju neophodni podaci procesima koji zatim vrše množenje vrste transponovane matrice i vektora funkcije. Na slici 27 su prikazi podaci nad kojima proces sa rankom 11 izvršava opisane operacije kao i rezultat izvršenja ovih operacija.



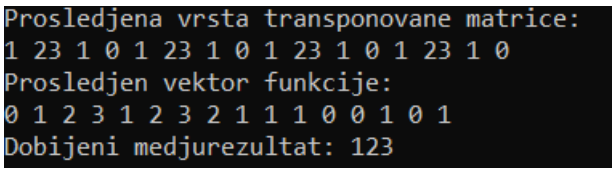
Slika 27. Prosleđeni parametri i rezultat izvršenja programa procesa sa rankom 11

Na slici 28 su prikazi podaci nad kojima proces sa rankom 14 izvršava opisane operacije kao i rezultat izvršenja ovih operacija.



Slika 28. Prosleđeni parametri i rezultat izvršenja programa procesa sa rankom 14

Na slici 29 su prikazi podaci nad kojima proces sa rankom 2 izvršava opisane operacije kao i rezultat izvršenja ovih operacija.



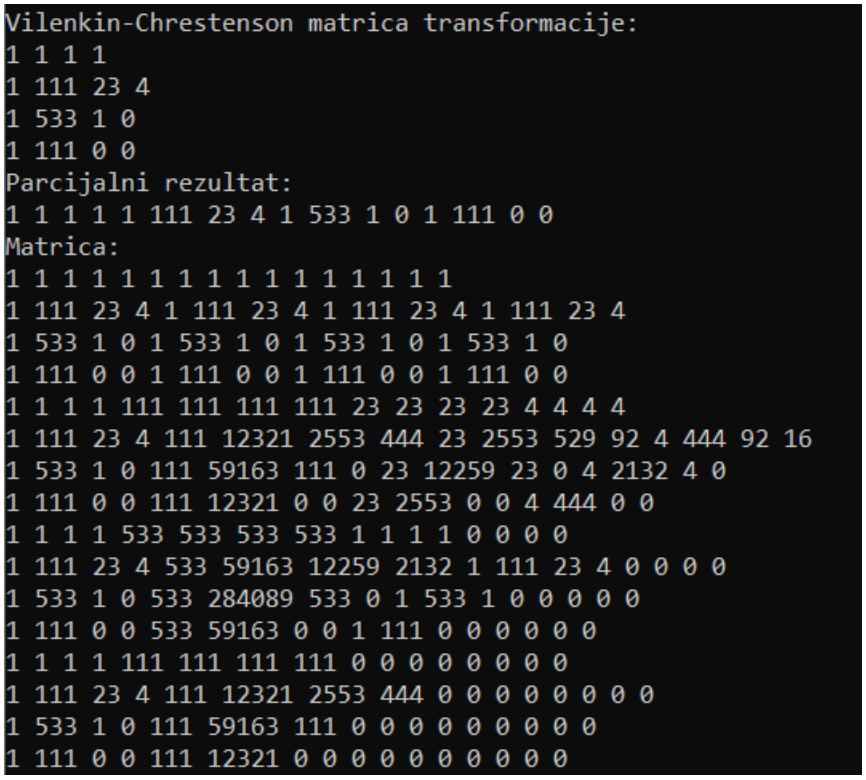
Slika 29. Prosleđeni parametri i rezultat izvršenja programa procesa sa rankom 2

Primenom funkcije MPI_Gather u master procesu se formira konačni Vilenkin-Chrestenson spektar zadate funkcije (Slika 30).

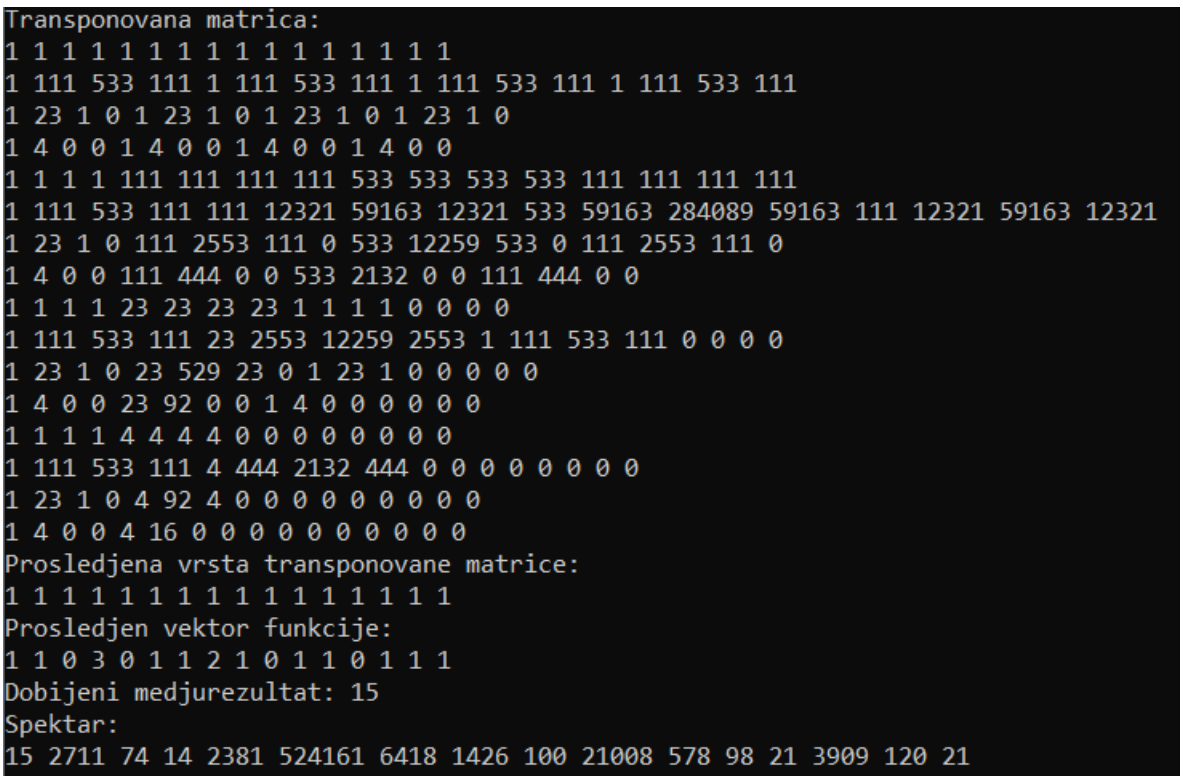
Slika 30. Dobijeni Vilenkin-Chrestenson spektar funkcije zadate vektorom [0,1,2,3,1,2,3,2,1,1,1,0,0,1,0,1]

- Rezultati izvršenja programa za vektor funkcije [1,1,0,3,0,1,1,2,1,0,1,1,0,1,1,1]

Master proces izvršava sve operacije kao i ostali aktivni procesi ali i operacije koje se obrađuju sekvencijalno. Iz ovog razloga u analizi rezultata izvršenja programa za vektor funkcije [1,1,0,3,0,1,1,2,1,0,1,1,0,1,1,1] će biti predstavljen kompletan kod koji ovaj proces izvršava, kao i kod koji izvršavaju ostali procesi. Na slikama 31 i 32 je prikazan rezultat programa koji se izvršava u procesu sa rankom 0.



Slika 31. Prvi deo rezultata izvršenja programa master procesa



Slika 32. Drugi deo rezultata izvršenja programa master procesa

U master procesu su izvršena izračunavanja međurezultata u paralelnim regionima, izvršavanje sekvencijalnog programa kao i prikaz konačnih rezultata. Za funkciju predstavljenu vektorom [1,1,0,3,0,1,1,2,1,0,1,1,0,1,1,1] dobijen je Vilenkin-Chrestenson spektar [15 2711 74 14 2381 524161 6418 1426 100 21008 578 98 21 3909 120 21].

Rezultati izvršenja programa u procesu sa rankom 10 su prikazani na slici 33.

```
Proces sa rankom 10
Primio:
1 1 1 1
1 111 23 4
1 533 1 0
1 111 0 0
Moj element matrice je: 1.
Parcijalni rezultat:
1 1 1 1 1 111 23 4 1 533 1 0 1 111 0 0
Prosledjena vrsta transponovane matrice:
1 23 1 0 23 529 23 0 1 23 1 0 0 0 0 0
Prosledjen vektor funkcije:
1 1 0 3 0 1 1 2 1 0 1 1 0 1 1 1
Dobijeni medjurezultat: 578
```

Slika 33. Rezultati izvršenja programa u procesu sa rankom 10

Rezultati izvršenja programa u procesu sa rankom 15 su prikazani na slici 34.

```
Proces sa rankom 15
Primio:
1 1 1 1
1 111 23 4
1 533 1 0
1 111 0 0
Moj element matrice je: 0.
Parcijalni rezultat:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Prosledjena vrsta transponovane matrice:
1 4 0 0 4 16 0 0 0 0 0 0 0 0 0 0
Prosledjen vektor funkcije:
1 1 0 3 0 1 1 2 1 0 1 1 0 1 1 1
Dobijeni medjurezultat: 21
```

Slika 34. Rezultati izvršenja programa u procesu sa rankom 15

Rezultati izvršenja programa u procesu sa rankom 14 su prikazani na slici 35.

```
Proces sa rankom 14
Primio:
1 1 1 1
1 111 23 4
1 533 1 0
1 111 0 0
Moj element matrice je: 0.
Parcijalni rezultat:
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Prosledjena vrsta transponovane matrice:
1 23 1 0 4 92 4 0 0 0 0 0 0 0 0 0
Prosledjen vektor funkcije:
1 1 0 3 0 1 1 2 1 0 1 1 0 1 1 1
Dobijeni medjurezultat: 120
```

Slika 35. Rezultati izvršenja programa u procesu sa rankom 14

6.6. Analiza performansi programa

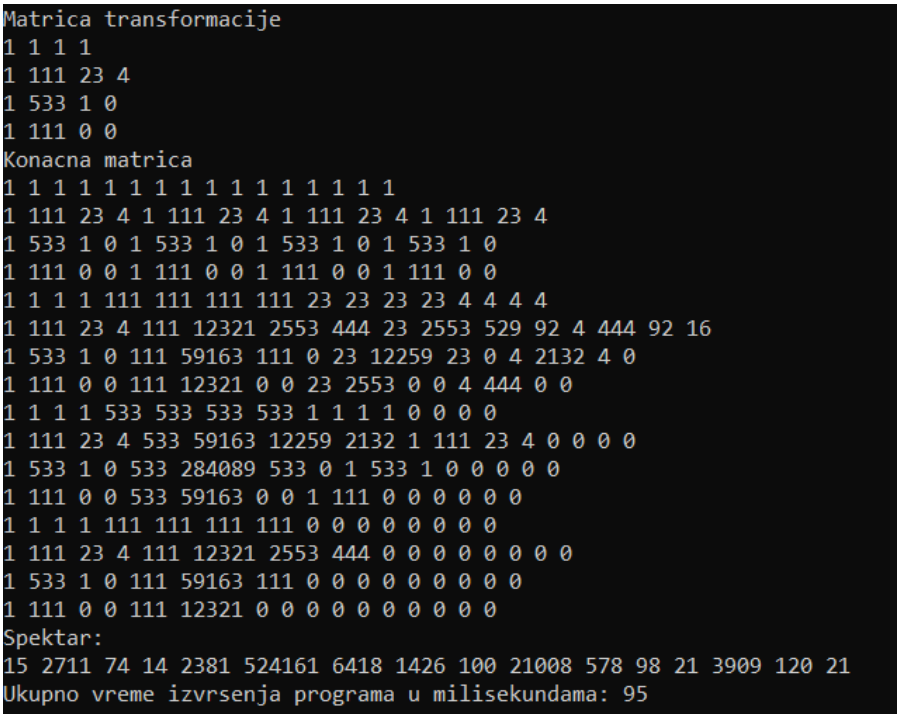
U projektu je implementirano i sekvencijalno izračunavanje Vilenkin-Chrestenson spektra kako bi se izvršila uporedna analiza rezultata dobijenih paralelnim i sekvencijalnim izvršenjem.

Na slici 36 prikazano je sekvencijalno izračunavanje Vilenkin-Chrestenson spektra, dok je na slici 37 prikazan rezultat izvršenja ovog programa. Najpre je određena matrica transformacije, a zatim je primenjen Kronekerov proizvod nad dobijenim matrica. Ovako dobijena matrica reda 16x16 je transponovana kako bi se kasnije odredio Vilenkin-Chrestenson spektar njenim množenjem sa prosleđenim vektorom funkcije.

```
auto start = high_resolution_clock::now();

transformMatrix = new int* [n];
for (int i = 0; i < n; i++)
    transformMatrix[i] = new int[n];
int k;
for (int i = 0; i < n; i++)
    for (int j = 0; j < n; j++)
    {
        k = (n - i * j) % n;
        transformMatrix[i][j] = exp(2 * Pi * i * k / n);
    }
for(int i=0; i<n; i++)
    for (int j = 0; j < n; j++)
    {
        for (int k = 0; k < n; k++)
            for (int l = 0; l < n; l++)
            {
                matrix[i * n + k][j * n + l] = transformMatrix[i][j] * transformMatrix[k][l];
            }
    }

for (int i = 0; i < n * n; ++i)
    for (int j = 0; j < n * n; ++j) {
        Tmatrix[j][i] = matrix[i][j];
    }

int spectar[16] = {};

for (int i = 0; i < n * n; ++i)
    for (int j = 0; j < n * n; ++j) {
        spectar[i] += Tmatrix[i][j] * vector[j];
    }
```

Slika 36. Sekvencijalno izračunavanje Vilenkin-Chrestenson spektra

Na slici 37 prikazan je rezultat sekvencijane implementacije Vilenkin-Chrestenson spektra. Najpre je prikazana Vilenkin-Chrestenson matrica transformacije, zatim matrica dobijena primenom Kronekerovog proizvoda nad matricama transformacije označena kao konačna matrica na slici 37. Množenjem ovako dobijene matrice i prosleđenog vektora funkcije određen je Vilenkin-Chrestenson spektar. Za funkciju predstavljenu vektorom [1,1,0,3,0,1,1,2,1,0,1,1,0,1,1,1] dobijen je Vilenkin-Chrestenson spektar [15 2711 74 14 2381 524161 6418 1426 100 21008 578 98 21 3909 120 21] koji odgovara vektoru dobijenom paralelnim izračunavanjem nad istom funkcijom. Ukupno vreme izvršenja programa je 95 milisekundi.



Slika 37. Rezultat sekvencijalnog izračunavanja Vilenkin-Chrestenson spektra

Paralelnom implementacijom Vilenkin-Chrestenson spektra prosečno vreme potrebno za izvršenje programa je 73.18 ms. U tabeli 4 je prikazano ukupno vreme izvršenja programa za svaki od procesa.

Proces	Vreme izvršenja (ms)
P0	43
P1	52
P2	78
P3	63
P4	56
P5	51
P6	42
P7	71
P8	35
P9	81
P10	103
P11	96
P12	124
P13	120
P14	113
P15	43

Tabela 4. Vreme izvršenja programa po procesima izraženo u milisekundama

Na osnovu dobijenih rezultata može se zaključiti da je vreme izvršenja paralelne implementacije Vilenkin-Chrestenson spektra manje za 21.82ms u odnosu na sekvencijano izvršenje. Ovakav zaključak je i bio očekivan obzirom na to da je ukupan posao u paralelnoj implementaciji podeljen većem broju niti koje se istovremeno izvršavaju.

7. Zaključak

Vilenkin-Chrestenson transformacija je diskretna multiplikativna transformacija koja je veliku primenu pronašla u kodiranju informacija različitih domena poput talasnih podataka signala, u metodama za spektralnu analizu mozaika kao i u algortmima za otkrivanje grešaka. Poboljšanje efikasnosti izračunavanja Vilenkin-Chrestenson spektra funkcija sa različitim brojem promenljivih može proširiti područje primene Vilenkin-Chrestenson transformacija. Matrica Vilenkin-Chrestenson-ove transformacije može se faktorizovati na različite načine što predstvalja osnovu različitih algoritama brze transformacije (eng. fast transform algorithms). Dva najznačajnija algoritma su Cooley-Tukey i algoritmi konstantne geometrije. Takođe, adekvatnim mapiranjem Vilenkin-Chrestenson transformacije na GPU omogućeno je brže i jednostavnije izračunavanje Vilenkin-Chrestenson spektra. U ovom radu implementirano je paralelno izračunavanje Vilenkin-Chrestenson spektra korišćenjem MPI framework-a. Izvršena je analiza dobijenih rezultata kao i uporedni prikaz paralelne i sekvencijalne implementacije. Na osnovu analize performansi oba pristupa može se zaključiti da implementacija Vilenkin-Chrestenson spektra korišćenjem grafičke procesne jedinice daje bolje rezultate u odnosu na sekvencijano izvračunavanje.

8. Reference

[1] Gajić D, Stanković R. Computation of the Vilenkin-Chrestenson Transform on a GPU. Department of Computer Science, Faculty of Electronic Engineering, University of Niš.

[2] Farkov Y. Discrete wavelets and the Vilenkin-Chrestenson transform. Mathematical Notes. 2011

[3] Stanković R., Astola J.T., Moraga C. Representation of Multiple-Valued Logic Functions. Morgan & Claypool. 2012

[4] Stanković S., Stanković M., Astola J. Representation of Multiple-valued Bent Functions Using Vilenkin-Chrestenson Decision Diagrams. 2011

[5] R. S. Stankovi´c, T. Sasao, C. Moraga, "Spectral transform decision diagrams" iRepresentations of Discrete Functions, T. Sasao, M. Fujita, (eds.), 55-92., Kluwer Academic Publishers, 1996.

[6] R. S. Stankovic, J. T. Astola, Spectral Interpretation of Decision Diagrams, Springer 2003.

[7] O. S. Rothaus, "On 'bent' functions", Journal of Combinatorial Theory, Ser. A, Vol. 20, 300-305, 1976.

[8] D. B. Gajić, R. S. Stanković. (2011). GPU accelerated computation of fast spectral transforms. Facta Universitatis - Series: Electronics and Energetics, 24(3):483–499.

[9] J. Hennessy and D. Patterson. (2008). Computer Organization and Design: The Hardware/Software Interface. Morgan Kaufmann.

[10] J. Astola, R. S. Stankovi´c, Fundamentals of Switching Theory and Logic Design, Springer, 2006.

[11] J. W. Cooley and J. W. Tukey. (1965). An algorithm for the machine calculation of complex Fourier series. Mathematics of Computation, 19(90):297–301.

[12] R. E. Bryant, "Graph-based algorithms for Boolean functions manipulation", IEEE Trans. Comput., Vol. C-35, No. 8, 1986, 667 - 691.