Elektrotehnički fakultet
Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student:
Maksimović Marko 1104/21

Mentori:
prof. dr Mladen Knežić
prof. dr Mitar Simić
ma Vedran Jovanović
dipl. inž. Dimitrije Obradović

Januar 2026. godine

1. Opis projektnog zadatka

Tema projektnog zadatka jeste kompresija slike korištenjem diskretne kosinusne transformacije (*Discrete Cosine Transform - DCT*). Potrebno je realizovati sistem za kompresiju *grayscale* slika upotrebom DCT algoritma. Kompresija se bazira na JPEG standardu, što znači da prati korake propisane JPEG standardom *ISO/IEC 10918* [1]. Koraci su navedeni u specifikaciji projektnog zadatka, te se ovdje neće ponavljati.

Prije nego što se započne kompresija slike, potrebno je izvršiti **transformaciju prostora boja** iz RGB u YCbCr. YCbCr je prostor boja koji se sastoji od tri komponente:

- Y – *luma*, osvjetljenje piksela.
- Cb – *chroma blue*, hroma plave boje. Izražava razliku između plave i luma komponente piksela.
- Cr – *chroma red*, hroma crvene boje. Izražava razliku između crvene i luma komponente piksela.

Za *grayscale* reprezentaciju slike dovoljno je posmatrati samo Y komponentu. Tipično se za Cb i Cr komponente vrši pododmjeravanje (*subsampling*), što znači da se one ne čuvaju za svaki piksel, nego za svaki drugi ili četvrti piksel. Ovo daje dobre rezultate jer je ljudsko oko osjetljivo na crno-bijele detalje, ali slabije na detalje u boji.

Konverzija RGB prostora boja u YCbCr prostor boja moguća je na više načina, te je definisano više standarda, zavisno od primjene. Konkretno, JPEG standard koristi ITU-R BT.601 standard [2], te se konverzija vrši prema sljedećim formulama:

$$\begin{cases} Y &= 0.299R &+0.587G &+0.114B & \\ C_b &= -0.1687R &-0.3313G &+0.5B &+128 \\ C_r &= 0.5R &-0.4187G &-0.0813B &+128 \end{cases} \tag{1.1}$$

Prije nego što se primjeni DCT, za dobijanje validnih rezultata potrebno je Y komponente centrirati oko nule (jer DCT vrši poređenje sa kosinusnim funkcijama koje su takođe centrirane oko nule).

Diskretna kosinusna transformacija je tehnika transformacije signala iz vremenskog domena u frekvencijski domen. Postoje različite formulacije DCT transformacije, ali je nama od interesa **dvodimenzionalna DCT tipa II** koja se definiše kao linearna transformacija:

$$F(u,v) = \alpha(u)\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f\,(x,y)\cos\left(\frac{\pi}{N}\left(x+\frac{1}{2}\right)u\right)\cos\left(\frac{\pi}{N}\left(y+\frac{1}{2}\right)v\right), u, v = 0,1,\ldots,N-1 \quad (1.2)$$

pri čemu su faktori normalizacije:

$$\alpha(0) = \frac{1}{\sqrt{N}}, \alpha(k) = \sqrt{\frac{2}{N}}(k \geq 1). \quad (1.3)$$

Transformacija je ortonormalna. Možemo je zapisati u matričnom obliku. Neka je C DCT matrica dimenzija N x N definsana sa:

$$C_{u,x} = \alpha(u)\cos\left(\frac{\pi}{N}\left(x+\frac{1}{2}\right)u\right) \quad (1.4)$$

Tada DCT transformaciju možemo zapisati kao:

$$F = CfC^T \quad (1.5)$$

Kako vrijedi ortonormalnost, vrijedi da je inverzna transformacija:

$$f = C^T FC \quad (1.6)$$

Konceptualno, DCT transformacija funkcioniše tako što svaki blok slike poredi sa N različitih frekvencija po obe ose bloka slike. Što je veće poklapanje generisane frekvencije i slike, to je veća vrijednost koeficijenta F. Argumenti *u* i *v* definišu harmonike vertikalne i horizontalne frekvencije, respektivno.

DCT se primjenjuje na blokove slike fiksne veličine 8 x 8 piksela, te je prije primjene DCT transformacije potrebno izvršiti segmentaciju slike na blokove veličine 8 x 8 piksela.

Nakon izvšene DCT transformacije nad svakim blokom, potrebno je izvršiti samu kompresiju. Bez koraka kompresije, IDCT bi nam omogućila rekonstrukcije kompletne slike, bez gubitaka (*lossless*

compression). Cilj JPEG kompresije jeste smanjenje podataka potrebnih za reprezentaciju informacija sa gubitkom (*lossy compression*). Zato se nakon DCT-a vrši **kvantizacija koeficijenata**, a zatim i **kodovanje.**

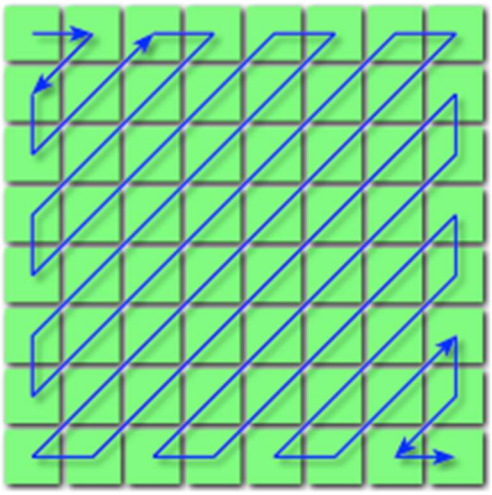
Kvantizacija se vrši koeficijent po koeficijent, primjenom matrice kvantizacije. Cilj matrice kvantizacije jeste da se određeni DCT koeficijenti strožije kvantuju (koeficijenti koji se nalaze "prema" donjem desnom dijelu bloka DCT koeficijenata, jer oni sadrže manje vrijednosti koje ljudsko oko neće primjetiti, pošto se radi o visokim frekvencijama po jednoj ili obe ose), a blaže one koeficijente koji su bliže gornjem lijevom ćošku bloka DCT koeficijenata, jer se tu nalaze bitnije frekvencije koje ljudsko oko lakše primjeti. Manipulacijom vrijednostima u matrici kvantizacije direktno se utiče na faktor kompresije, ali i na kvalitet ostvarene kompresije.

Kvantizovani koeficijenti dobijaju se prema formuli:

$$\hat{F}(u,v) = round\left(\frac{F(u,v)}{Q(u,v)}\right) \tag{1.7}$$

Iako JPEG kompresija dopušta upotrebu proizvoljnih matrica kvantizacije, ovaj rad koristi standardne tabele preporučene u **Annex K ISO/IEC 10918-1** [1] .

Da bi omogućili efikasnu serijalizaciju kvantizovanih koeficijenata DCT transformacije, potrebno je izvršiti permutaciju istih kako bi se postigao redoslijed pogodan za kodovanje. Ova vrsta permutacije naziva se **cik-cak** (engl. *zig-zag*) **permutacija.** Cik-cak permutacija zasniva se na čitanju koeficijenata iz matrice u niz prema predefinisanom redoslijedu, tako da se koeficijenti poredaju od nižih ka višim frekvencijama. Ilustracija redoslijeda čitanja koeficijenata iz matrice data je na sljedećoj slici.



Slika 1.1 - *Ilustracija čitanja koeficijenata*

Kodovanje se vrši nakon cik-cak permutacije i ima za cilj predstavljanje DCT koeficijenata na način koji zauzima minimalan memorijski prostor. Potrebno je da razlikujemo dvije vrste DCT koeficijenata, AC i DC koeficijente. **DC koeficijent** je prvi koeficijent u cik-cak nizu (frekvencija 0 po obe dimenzije pa on opisuje "jednosmjernu" frekvencijsku komponentu bloka) i izražava prosječan nivo intenziteta u bloku. **AC koeficijenti** su svi ostali koeficijenti i izražavaju promjene intenziteta, tj. detalje i teksture različitih frekvencija.

Razlikovanje koeficijenata je bitno jer se kodovanje istih vrši na različit način. DC koeficijenti se koduju **prediktivno,** što znači da se za svaki blok umjesto vrijednosti DC koeficijenta upisuje razlika DC koeficijenta trenutnog bloka i prošlog bloka. Na ovaj način se omogućuje izražavanje informacije o vrijednosti DC koeficijenta sa manjim brojem bita, jer je uobičajeno izražena sličnost između DC koeficijenata susjednih blokova. S druge strane, AC koeficijenti se koduju upotrebom *run-length kodovanja* (*Run Length Encoding – RLE*). RLE kodovanje se zasniva na identifikaciji sekvenci ponavljajućih karaktera, pa se izražava informacija o dužini takvog niza i ponavljajućeg karaktera. Konkretno, kod JPEG kompresije su česte sekvence uzastopnih nula u AC koeficijentima, pa se koristi namjenska tehnika RLE kodovanja koja identifikuje samo sekvence uzastopnih nula. Tako identifikovane

sekvence se prema JPEG standardu izražavaju na sljedeći način.
Svaka identifikovana sekvenca nula se izražava kao par

$$[(r, s), c] \tag{1.8}$$

pri čemu članovi *r* i *s* zauzimaju po 4 bita svaki, te se u literaturi takođe može naći i sljedeća specifikacija formata za ove vrijednosti: "RRRRSSSS". Član *r* izražava dužinu (*run*) sekvence nula i može da sadrži vrijednosti 0-15. Član *s* (engl. *size*, *category*) izražava broj bita potrebnih za reprezentaciju vrijednosti *c* koja slijedi nakon identifikovane sekvence i koja je različita od nula. Vrijednost *c* se reprezentuje varijabilnim brojem bita, specifikovanih u okviru vrijednosti *s,* a sama vrijednost se navodi u formi komplementa jedinice.

Entropijsko kodovanje slijedi nakon kodovanja DCT koeficijenata upotrebom prediktivnog i RLE kodovanja. Primjenjuje se na kodovane DC koeficijente i *(r, s)* parove kodovanih AC koeficijenata. Originalni JPEG standard definiše mogućnost upotrebe **Huffmanovog kodovanja,** kao i **aritmetičkog kodovanja** za postizanje entropijskog kodovanja, međutim, u praksi je to gotovo uvijek Huffmanovo kodovanje, te je ono i realizovano u okviru ovog rada.

Huffmanov kod je prefiksni kod, što znači da nijedan kod nije prefiks drugog koda, te je kao takav **jednoznačno dekodabilan**. Dekoder može da dekoduje i jednoznačno identifikuje svaki kod bez upotrebe separatora između kodova, što direktno doprinosi efikasnosti kompresije i smanjenju veličine fajla.
U okviru ovog rada nije vršeno generisanje namjenskih Huffmanovih kodova, već su korištene standardne preporučene tabele istih, dostupne u **Annex K.3 ISO/IEC 10918-1** [1].

Posljednji korak jeste **serijalizacija** samog kompresovanog sadržaja. Iako je JPEG kompresija nezavisna od formata serijalizacije sadržaja, uobičajena je upotreba JFIF (*JPEG File Interchange Format*) standardizovanog formata. JFIF format koristi niz markera koji imaju ulogu strukturisanja dokumenta i skladištenja metapodataka.

Bitno je napomenuti da je potrebno voditi računa o jednoznačnosti vrijednosti u okviru serijalizovanog fajla, kako bi dekoder mogao da odredi semantiku svake vrijednosti. Zato je potrebno prilikom serijalizacije, nakon svake pojave 0xFF vrijednosti izvršiti dopunjavanje sa vrijednošću 0x00, kako bi dekoder uspješno razlikovao vrijednost 0xFF od markera koji počinju sa vrijednošću 0xFF. Ovaj postupak se naziva *byte stuffing.*

2. Izrada projektnog zadatka

2.1. Razvojna ploča

Projektni zadatak realizovan je na razvojnoj ploči *Texas Instruments J721 EXSKG01EVM*. Osnovu ove platforme čini *TDA4VM System on a Chip* (*SoC*) procesor koji integriše više različitih procesorskih jedinica. Konkretno, od interesa za ovaj projektni zadatak su *Cortex-A72 Central Processing Unit* (*CPU*), kao i *Texas Instruments C71x Digital Signal Processor* (*DSP*).

2.2. Natural C implementacija

Riješenje projektnog zadatka prvo je realizovano kao monolitna aplikacija, pisana prirodnim (*natural*) C jezikom, projektovana i implementirana za izvršavanje na *General Purpose CPU* procesorima. Cilj ove implementacije jeste primjena teorijskih koncepata na kojima počiva JPEG kompresija, te uspostavljanje referentne tačke za validaciju rezultata dobijenih u kasnijim fazama razvoja.

Ova implementacija dostupna je u direktorijumu *natural_c*, te se može izgraditi (*build*) kao *jpeg_enc_nat_c* target upotrebom *CMake* meta-build sistema.

2.3. Texas Instruments implementacija

Nakon ostvarenog riješenja za *General Purpose CPU*, isto je *port*-ovano na TI razvojnu ploču.

Aplikacija se osloni na *Texas Instruments Processor Software Development Kit* (*PSDK*) *RTOS* za TDA4VM procesor, konkretno u verziji 09_02_00_05. Za uspješno kompajliranje aplikacije, potrebno je modifikovati TI *build system*. Modifikacija je dostupna u vidu *git patch*-a koji se primjenjuje na *PSDK*, te korisniku olakšava proces konfiguracije okruženja. Detaljno uputstvo o podešavanju okruženja, kompajliranju, prebacivanju kompajliranih artefakata na ploču i pokretanju aplikacije

dostupno je u okviru README.md fajla na repozitorijumu, te se ovdje neće detaljno razmatrati.

Izvršavanje aplikacije realizovano je u **heterogenom okruženju**, što podrazumijeva kooperativni rad i distribuciju zadataka između procesorskih jedinica različitih arhitektura. Izvršavanje započinje na procesoru Cortex-A72 (u terminologiji se koristi izraz *host* strana). Na *host* strani se izvršava Linux operativni sistem, te kao takav poznaje koncept fajl sistema i može da otvori fajl u kojem se nalazi slika. Host program učitava sliku, vrši ekstrakciju sirovih podataka u formi RGB piksela, te priprema podatke za slanje na C71x stranu, poštujući dogovoreni format podataka (definisan u zaglavlju *jpeg_compression.h*). A72 strana vrši **sinhroni poziv** servisa na C71x strani, što znači da se izvršavanje na *host* strani suspenduje dok se ne primi odgovor. Po prijemu odgovora, rezultat obrade se serijalizuje u datoteku formiranu po JFIF standardu.

Međuprocesorska komunikacija realizovana je upotrebom *Texas Instruments Vision Apps* radnog okvira (*Application Framework - AF*). Ovaj AF raspolaže komunikacionim mehanizmima koji apstrahuju *Inter-Processor Communication* (*IPC*), kao i *Remote Processor Messaging* (*RPMsg*) protokol i predstavljaju zaseban sloj iznad navedenih, te omogućuju programerima jednostavnije pisanje programa koji vrše komunikaciju između različitih procesora. Konkretno, za realizaciju ovog projekta iskorišten je *klijent-server* arhitekturni stil. A72 strana predstavlja klijenta u ovoj komunikaciji, dok C71x predstavlja server koji obezbjeđuje servise klijentu. Podaci se razmjenjuju putem dijeljene radne memorije, te je potrebno voditi računa o koherenciji keša, jer ona nija garantovana u heterogenom okruženju.

Glavni dio obrade podataka vrši se na procesoru TI C71x (u terminologiji se koristi izraz *device* strana). Ova strana obezbjeđuje servis za JPEG obradu slike. Obrada podataka je realizovana sa fokusom na iskorišćenje arhitekturnih prednosti i harverskih mogućnosti C71x procesora. Po završetku obrade, rezultat se putem dijeljene memorije vraća *host* strani.

Ova implementacija dostupna je u direktorijumu ti, te se može izgraditi (*build*) kao *make_ti_build* i *make_ti_debug_build* target upotrebom *CMake* meta-build sistema. Prvi target nema uključen kod za instrumentaciju i profilisanje, dok drugi ima.

Inicijalno riješenje, nakon *port*-ovanja na razvojnu ploču, je radilo korektno, ali sa niskim performansama izvršavanja po pitanju brzine. Zainteresovani čitalac može da pogleda isto odabirom

odgovarajućeg *commit*-a na git repozitorijumu.

Procesom inkrementalne optimizacije i iterativnog profilisanja koda, ostvaren je niz optimizacija koje su rezultovale boljim performansama, te je ostvaren lokalni minimum po pitanju broja ciklusa izvršavanja. Naknadne optimizacije su moguće, ali zahtijevaju dodatne vremenske resurse i napor.

3. Optimizacija

3.1. Profilisanje koda

Profilisanje koda nam pruža informaciju o procentualnoj zastupljenosti pojedinih funkcija i dijelova koda. Profilisanje nam pruža informaciju o potrošnji procesorskih ciklusa i/ili potrošnji memorijskog prostora.

Nakon *port*-ovanja koda na TI razvojnu ploču, izvršeno je profilisanje sa ciljem identifikacije uskog grla (*bottleneck*) performansi. Identifikacija uskog grla je korisna jer nam ukazuje na dio programa koji se najsporije izvršava i koji najviše doprinosi degradaciji performansi. Prirodno, upravo ovaj dio predstavlja polaznu tačku primjene optimizacionih tehnika jer je dobit najveća.

Treba napomenuti da je pravilo 80/20 prisutno i u kontekstu optimizacije softvera za *DSP* i da ono nalaže da fokus optimizacije treba da bude na 20% koda koji se izvršava 80% vremena (u praksi, petlje).

Nakon inicijalnog profilisanja, dobijeni su sljedeći rezultati. Svi rezultati profilisanja u ovom izvještaju su rezultat izvršavanja programa nad slikom *lena.bmp* dimenzija 512 x 512.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE         |                              CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion            |                           3,551,645 |
[C7x_1 ] | Segmentation (8x8)             |                           2,168,832 |
[C7x_1 ] | DCT Transform                  |                       3,082,797,792 |
[C7x_1 ] | Quantization                   |                          87,540,819 |
[C7x_1 ] | ZigZag Reorder                 |                           3,442,172 |
[C7x_1 ] | Huffman Encoding               |                           7,871,634 |
[C7x_1 ] ------------------------------------------------------------------------
[C7x_1 ] | TOTAL CYCLES                   |                       3,187,372,894 |
```

Na osnovu rezultata profilisanja, možemo da identifikujemo usko grlo. Ubjedljivo najveći broj ciklusa se troši na sprovođenje DCT transformacije, te ćemo se prvo fokusirati na optimizaciju iste.

3.2. Optimizacija DCT transformacije

DCT transformacija je inicijalno realizovana tako da se svaki put računaju DCT koeficijenti po formuli (1.2). Kompajler je konfigurisan da vrši *O3* optimizacije, što znači da se vrše agresivne optimizacije na nivou fajla. Detaljnim profilisanjem DCT funkcije dobijamo sljedeći uvid.

```
[C7x_1 ] | DCT Transform (Total)                    |          3,231,730,444 |
[C7x_1 ] |   -> Cos calc & mul                      |                    215 |
[C7x_1 ] |   -> Alpha calculation                   |                      9 |
[C7x_1 ] |   -> DCT coeff calculation               |                     18 |
```

Vidimo da se većina procesorskog vremena troši na računanje kosinusnog člana, ali i određivanje alfa faktora i računanje samog DCT koeficijenta troše takođe cikluse. Ono što se može primjetiti iz formule (1.2) jeste da nema potrebe da se svaki put iznova računaju kosinusni član i množenje sa alfa faktorom.

Posmatranjem formula (1.4) i (1.5) možemo zaključiti da je dovoljno jednom izračunati matricu C, a zatim je primjenjivati na različitim ulaznim blokovima. Ukoliko izvršimo računanje iste unaprijed i njene vrijednosti smjestimo u *lookup* tabelu, a DCT realizujemo u matričnoj formi, dobijamo značajno bolje rezultate.

```
[C7x_1 ] | DCT Transform (Total)                    |              5,088,696 |
[C7x_1 ] |   -> First matmul (C x f)                |                    668 |
[C7x_1 ] |   -> Second matmul ((C x f) x C^T)       |                    406 |
```

Ovaj rezultat predstavlja poboljšanje za nepuna tri reda veličine.

Osiguravanjem memorijskog poravnanja nizova i matrica na adrese djeljive sa 64 postiže se određeno poboljšanje. Ukoliko se nizovi i matrice nalaze na adresama djeljivim sa 64, .D jedinici je

lakše da učita podatke jer se oni nalaze na adresama koje su djeljive sa dužinom podataka koje jedinica može da učita u komadu. Ovo se manifestuje zamjenom LDNW asemblerskih instrukcija sa LDW instrukcijama. Nakon osiguravanja poravnanja, dobijamo sljedeće rezultate.

```
[C7x_1 ] | DCT Transform (Total)                    |             5,035,617 |
[C7x_1 ] |   -> First matmul (C x f)                |                   665 |
[C7x_1 ] |   -> Second matmul ((C x f) x C^T)       |                   457 |
```

Algoritam za množenje matrica podrazumijeva pristup jednoj matrici po redovima, što je pogodono jer se matrice linearizuju prema *row-major* pravilu u C/C++ programskom jeziku, te se na taj način ostvaruje prostorna lokalnost reference podataka u kešu (C71x ima *split-cache* modifikovanu Harvard arhitekturu), a drugoj matrici po kolonama, što iz istog razloga nije pogodono jer se na taj način ne ostvaruje prostorna lokalnost reference.
Međutim, ukoliko bi se drugi operand matričnog množenja zapisao u transponovanoj formi (što već imamo na raspolaganju jer imamo *lookup* tabelu transponovane C matrice), tada bi mogli da realizujemo matrično množenje upotrebom isključivo pristupa pa redovima, što bi puno bolje iskoristilo keš memoriju. Ukoliko iskoristimo osobinu transponovanja matričnog proizvoda i asocijativnost, možemo da zapišemo DCT u sljedećem obliku.

$$C \times (C \times f^T)^T = C \times (f \times C^T) = C \times f \times C^T \tag{3.1}$$

Ovaj oblik je pogodan jer omogućuje realizaciju dva matrična množenja sa pristupom matricama samo po redovima. Na ovaj način se iskorištava prostorna lokalnost reference podataka i keširanje daje dobre performanse, što se zaključuje i po rezultatima profilisanja.

```
[C7x_1 ] | DCT Transform (Total)                    |             3,020,297 |
[C7x_1 ] |   -> First matmul (C x f^T)              |                   245 |
[C7x_1 ] |   -> Second matmul (C x (C x f^T)^T)     |                   262 |
```

Bitno je napomenuti da nakon ove modifikacije koda nastaju blage razlike (~10B) na binarnog nivou rezultata kompresije jer se greške kvantizacije prilikom množenja i sabiranja podataka tipa *float*

akumuliraju drugačijim redoslijedom zbog drugačijeg redoslijeda operacija. Vizuelno, rezultat je isti.

Dodavanjem *#pragma MUST_ITERATE(8, 8, 8)* pretprocesorske direktive na petlje ne postiže se nikakav efekat, jer funkcija ima fiksan broj iteracija (8) i koristi se *O3* nivo optimizacija, što znači da kompajler sam primjenjuje potrebne optimizacije nad petljom na osnovu broja iteracija.

```
void perform_dct_on_block(int8_t * restrict b_start, float * restrict dct_coeffs) {
  int u, v, k;
  float __attribute__((aligned(64))) intermediate[64];

  ASSERT_ALIGNED_64(dct_matrix_c);
  ASSERT_ALIGNED_64(dct_matrix_c_T);

  float sum = 0.0f;
  #pragma MUST_ITERATE(8, 8, 8)
  for(u = 0; u < 8; u++)
    #pragma MUST_ITERATE(8, 8, 8)
    for(v = 0; v < 8; v++) {
      sum = 0.0f;
      #pragma MUST_ITERATE(8, 8, 8)
      #pragma UNROLL(8)
      for(k = 0; k < 8; k++)
        sum += dct_matrix_c[u * 8 + k] * b_start[v * 8 + k];

            intermediate[u * 8 + v] = sum;
    }



  #pragma MUST_ITERATE(8, 8, 8)
  for(u = 0; u < 8; u++)
    #pragma MUST_ITERATE(8, 8, 8)
    for(v = 0; v < 8; v++) {
      sum = 0.0f;
      #pragma MUST_ITERATE(8, 8, 8)
      #pragma UNROLL(8)
      for(k = 0; k < 8; k++)
        sum += dct_matrix_c[u * 8 + k] * intermediate[v * 8 + k];

            dct_coeffs[u * 8 + v] = sum;
    }
```

```
}
```

Dodavanjem *#pragma UNROLL(8)* dobijamo gore performanse.

```
[C7x_1 ] | DCT Transform (Total)                    |             5,630,953 |
[C7x_1 ] |   -> First matmul (C x f)                |                   626 |
[C7x_1 ] |   -> Second matmul ((C x f) x C^T)       |                   801 |
```

Analizom asemblerskih fajlova, dolazimo do zaključka da se odmotavanjem stvara velik pritisak na registre (*Register is live too long*) što dovodi do degradiranja performansi.

Dalji napredak postižemo **vektorizacijom** obrade. Koristimo tipove iz *runtime* biblioteke (*floatn*, pri čemu *n* mora biti stepen dvojke ne veći od 16). C71x procesor omogućuje *Single Instruction Multiple Data* (*SIMD*) obradu. Raspolaže vektorskim registrima koji su dužine 256b/512b, zavisno od konkretnog modela procesora (u slučaju odabrane razvojne ploče, to je 512b). Primjenom SIMD operacija, dobijamo poboljšanje performansi.

```
[C7x_1 ] | DCT Transform (Total)                    |             1,594,111 |
[C7x_1 ] |   -> First matmul (C x f)                |                   623 |
[C7x_1 ] |   -> Second matmul ((C x f) x C^T)       |                   102 |
```

Programski kod iskorištava vektorske tipove i činjenicu da C71x kompajler zna da emituje SIMD asemblerske instrukcije za standardne C aritmetičke operatore nad vektorskim tipovima.

```
void perform_dct_on_block(int8_t * restrict b_start, float * restrict dct_coeffs) {
  //...

  #pragma MUST_ITERATE(8, 8, 8)
  for(i = 0; i < 8; i++) {
    char8 in_char = *((char8 *)&b_start[i * 8]);
    float8 in_vec = __convert_float8(in_char);
    float8 row_acc = (float8)0.0f;
    #pragma UNROLL(8)
    for(k = 0; k < 8; k++) {
      float pixel_val = in_vec.s[k];

      float8 c_row = *((float8 *)&dct_matrix_c_T[k * 8]);

      row_acc += c_row * pixel_val;
```

```
    }
    intermediate_regs[i] = row_acc;
  }

  #pragma MUST_ITERATE(8, 8, 8)
  for(i = 0; i < 8; i++) {
    float8 c_factors = *((float8 *)&dct_matrix_c[i * 8]);
    float8 row_acc = (float8)0.0f;
    #pragma UNROLL(8)
    for(k = 0; k < 8; k++) {
      float c_val = c_factors.s[k];
      float8 m_row = intermediate_regs[k];
      row_acc += m_row * c_val;
    }
    *((float8 *)&dct_coeffs[i * 8]) = row_acc;
  }

}
```

Za učitavanje skalara iz vektorskog registra koristimo pristup skalaru preko .C jedinice, što stvara pritisak na istu. Ovo se može zaključiti iz asemblerskog fajla.

```
Bound(.C2)          -     9*
```

Eventualno poboljšanje možemo postići ako učitavanje skalara ne vršimo preko .C jedinice, nego upotrebom .D jedinica, jer ih imamo više od .C jedinica. Postižemo manje opterećenje na .D jedinice i suptilno bolje rezultate.

```
Bound(.D)           5
```

```
[C7x_1 ] | DCT Transform (Total)                    |             1,522,486 |
[C7x_1 ] |   -> First matmul (C x f)                 |                   552 |
[C7x_1 ] |   -> Second matmul ((C x f) x C^T)        |                   103 |
```

Poboljšanje bi bilo moguće postići alokacijom bafera za *lookup* tabele u okviru L2 memorije (TI dozvoljava da se dio SRAM memorije koristi kao keš, a dio kao adresibilni SRAM), međutim, *Vision Apps* AF koristi namjenske linkerske skripte koje alociraju sav dostupni prostor u L1 i L2 za svoje potrebe i nisam uspio da izmjenim konfiguraciju istih da omogućim smještanje *lookup* tabela u SRAM

memoriju (kompajliranje nije uspješno). Zbog vremenskog ograničenja odustao sam od izmjene linkerskih skripti, ali sam se dosjetio da C71x ima harversku jedinicu koja omogućuje pribavljanje podataka prema predefinsanom pravilu, a pruža iste performanse pristupa kao L1 memorija – *Streaming Engine.*

Streaming Engine (*SE*) predstavlja perifernu hardversku jedinicu koja se može konfigurisati da radi kao *Direct Memory Access* (*DMA*) kontroler u kontekstu dobavljanja podataka iz memorije ka procesoru. Konfigurisanje iste zahtjeva postojanje inicijalizacionog koda čiji cilj je podešavanje parametara rada *SE* jedinice.

```
// --- Streaming Engine Setup ---
  __SE_TEMPLATE_v1 se_tmplt = __gen_SE_TEMPLATE_v1();
  se_tmplt.ELETYPE = __SE_ELETYPE_8BIT;
  se_tmplt.VECLEN  = __SE_VECLEN_8ELEMS;
  se_tmplt.ICNT0   = 64;
  se_tmplt.ICNT1   = num_blocks * 64;
  se_tmplt.DIM1    = 64;
  se_tmplt.PROMOTE = __SE_PROMOTE_OFF;

  __SE0_OPEN(input_buffer, se_tmplt);
```

Pored SE jedinica, C71x raspolaže i sa Streaming Address Generator (SA) jedinicama, koje takođe predstavljaju periferne hardverske jedinice, sa zadatkom efikasnog računanja adresa u ugnježdenim petljama (do 6 dimenzija). Upotreba istih je slična kao upotreba SE jedinica.

```
// --- Streaming Address Generator setup ---
  __SA_TEMPLATE_v1 sa_tmplt = __gen_SA_TEMPLATE_v1();
  sa_tmplt.VECLEN  = __SA_VECLEN_8ELEMS;
  sa_tmplt.DIMFMT  = __SA_DIMFMT_2D;
  sa_tmplt.ICNT0   = 64;
  sa_tmplt.ICNT1   = num_blocks * 64;
  sa_tmplt.DIM1    = 64;

  __SA0_OPEN(sa_tmplt);
```

Nakon konfiguracije *SE* jedinice da dobavlja podatke za procesiranje (centrirane Y komponente) i *SA*

jedinice da računa adrese u izlaznom baferu, dobija se značajno poboljšanje performansi.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                          |                CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion                             |             3,543,646 |
[C7x_1 ] | Segmentation (8x8)                              |             2,172,128 |
[C7x_1 ] | DCT Transform (Total)                           |               896,888 |
[C7x_1 ] | Quantization                                    |            87,582,097 |
[C7x_1 ] | ZigZag Reorder                                  |             3,453,757 |
[C7x_1 ] | Huffman Encoding                                |             7,911,389 |
[C7x_1 ] ------------------------------------------------------------------------
[C7x_1 ] | TOTAL CYCLES                                    |           105,559,905 |
[C7x_1 ] ========================================================================
```

Za postizanje ovog rezultata, bilo je potrebno uložiti vrijeme i strpljenje za pronalaženje optimalne konfiguracije petlji empirijskim procesom isprobavanja različitih parametara *pragma* direktiva. Optimalan rezultat postiže se odmotavanjem vanjskih petlji faktorom 4. Odmotavanje faktorom 8 dovodi do degradiranja performansi jer diskvalifikuje petlju za *Software Pipelining* zbog prevelikog broja instrukcija u tijelu petlje.

3.3. Optimizacija konverzije prostora boja

Konverzija prostora boja može da se optimizuje primjenom vektorske obrade. Vektorska implementacija, slično kao i kod DCT implementacije, koristi SIMD instrukcije i vektorske tipove podataka. Fuzijom konverzije prostora boja, centriranja oko nule i dobavljanja podataka postižemo bolje performanse jer se podaci ne moraju smještati u memoriju pa ponovo obrađivati, već se sve dešava dok su podaci još u registrima procesora.

```
extern "C" void fetch_next_block(int8_t* y_output) {

  char32 * vec_y_out = (char32 *) y_output;
```

```
    short32 coeff_r = (short32) 77;
    short32 coeff_g = (short32) 150;
    short32 coeff_b = (short32) 29;
    short32 val_128 = (short32) 128;

    // Upper half of one block. We load it and upcast it to 512b (32 shorts).
    uchar32 r_in_1 = vec_r_src[0];
    uchar32 g_in_1 = vec_g_src[0];
    uchar32 b_in_1 = vec_b_src[0];

    // Upcasting uchar32 to short32 to take up one V register in whole. We need this upcast because of Y conversion and
multiplying by short.
    short32 r_s_1 = __convert_short32(r_in_1);
    short32 g_s_1 = __convert_short32(g_in_1);
    short32 b_s_1 = __convert_short32(b_in_1);

    // Perform SIMD instructions
    short32 y_temp_1 = (r_s_1 * coeff_r) + (g_s_1 * coeff_g) + (b_s_1 * coeff_b);

    y_temp_1 = (y_temp_1 >> 8) - val_128;

    // We write out the result, but downcast it to char32 first.
    vec_y_out[0] = __convert_char32(y_temp_1);

    // We repeat this for the lower half of the block.
    uchar32 r_in_2 = vec_r_src[1];
    uchar32 g_in_2 = vec_g_src[1];
    uchar32 b_in_2 = vec_b_src[1];

    // Upcast
    short32 r_s_2 = __convert_short32(r_in_2);
    short32 g_s_2 = __convert_short32(g_in_2);
    short32 b_s_2 = __convert_short32(b_in_2);

    short32 y_temp_2 = (r_s_2 * coeff_r) + (g_s_2 * coeff_g) + (b_s_2 * coeff_b);

    y_temp_2 = (y_temp_2 >> 8) - val_128;

    vec_y_out[1] = __convert_char32(y_temp_2);

    // We have laoded two char32 vectors while loading a single block, so we increment pointers by 2
    vec_r_src += 2;
    vec_g_src += 2;
    vec_b_src += 2;
}
```

Data funkcija vrši učitavanje podataka iz memorije, smještanje istih u vektorske registre, te obradu i smještanje u izlazni bafer. Funkcija je pisana u C++ programskom jeziku, te uvezana sa ostatkom C koda iz razloga što je za naredni korak optimizacije neophodan C++ programski jezik.
Rezultati optimizacije su sljedeći.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                        |                 CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion                           |              2,394,118 |
[C7x_1 ] ------------------------------------------------------------------------
```

3.4. Optimizacija blokovske obrade

Generalno postoje dva pristupa blokovskoj obradi slike. Jedan je da se svaka faza sprovodi nad svim blokovima slike prije prelaska na sljedeću fazu, što je pogodno za instrukcijski keš, ali nije dobro za prostornu lokalnost reference podataka. Ovaj pristup se naziva *horizontalno procesiranje*.

Trenutna formulacija algoritma zasniva se na horizontalnom procesiranju.. Ovo nije optimalno jer zahtjeva postojanje velikih bafera u memoriji za skladištenje međurezultata između svake dvije faze. Ovakvi baferi se smještaju u DDR memoriji što dovodi do penala performansi.

Drugi pristup naziva se *vertikalno procesiranje* i zasniva se na sprovođenju svih faza nad jednim blokom prije prelaska na obradu narednog bloka. Ovaj pristup iskorištava prostornu lokalnost reference podataka i omogućava efikasan pristup podacima unutar jednog bloka tokom svih faza obrade, ali po cijenu pogoršanih performansi keširanja instrukcija.

Prelaskom na vertikalno procesiranje, dobijamo bolju prostornu lokalnost reference bloka koji se obrađuje, ali pogoršavamo performanse dobavljanja narednog bloka iz memorije. Kako se sada blokovi slike keširaju, *SE* jedinica više nije potrebna za efikasno dobavljanje bloka slike za DCT obradu. Njena upotreba sada postaje smislena prilikom dobavljanja blokova iz memorije, te je konfigurišemo da radi kao DMA jedinica koja dobavlja blokove slika iz sporije DDR memorije u male bafere bliže procesoru.

Ograničen broj *SE* jedinica C7x procesora predstavlja prepreku. Da bi se iskoristila vektorska obrada na optimalan način, idealno je da se R, G i B komponente nalaze u zasebnim baferima iz kojih procesor čita komponente i sprovodi vektorske operacije nad istima. Međutim, na raspolaganju imamo samo 2 *SE* jedinice. Iskorištavanjem tih jedinica tako da jedna učitava samo R komponente, druga G komponente, a učitavanje B komponente ostavljamo .D jedinici, dobijamo poboljšane performanse u odnosu na učitavanje podataka isključivo preko .D jedinica.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                          |                CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion                             |             1,468,416 |
[C7x_1 ] ------------------------------------------------------------------------
```

Međutim, ako modifikujemo način na koji se podaci na *host* strani upisuju u dijeljenu memoriju tako da se R komponenta upisuje u jedan bafer, a komponente B i G u drugi bafer **isprepleteno**, tj. da se upisuju 32 bajta G, a zatim 32 bajta B komponente, postižemo situaciju u kojoj možemo da iskoristimo obe *SE* jedinice na *device* strani. Ekstrakcija B, odnosno G komponenti iz bafera je jednostavna, jer vektorske operacije podržavaju ekstrakciju donjeg (*lo*), odnosno gornjeg (*hi*) dijela vektora. Na ovaj način postižemo dodatno poboljšanje performansi.

Potrebno je prvo konfigurisati *SE* jedinice da znaju odakle i prema kojim pravilima da dobavljaju podatke, te iste i "otvoriti".

```
extern "C" void fetch_setup(uint8_t* r_vec, uint8_t* gb_vec, uint64_t image_length) {

  __SE_TEMPLATE_v1 se_params = __gen_SE_TEMPLATE_v1();

  se_params.ELETYPE = __SE_ELETYPE_8BIT;
  se_params.VECLEN  = __SE_VECLEN_32ELEMS;
  se_params.ICNT0   = image_length;
  se_params.DIM1    = 0;
  se_params.ICNT1   = 0;
  se_params.DIM2    = 0;
  se_params.ICNT2   = 0;
```

```
  __SE_TEMPLATE_v1 se_params2 = __gen_SE_TEMPLATE_v1();
  se_params2.ELETYPE = __SE_ELETYPE_8BIT;
  se_params2.VECLEN  = __SE_VECLEN_64ELEMS;
  se_params2.ICNT0   = image_length * 2;
  se_params2.DIM1    = 0;
  se_params2.ICNT1   = 0;
  se_params2.DIM2    = 0;
  se_params2.ICNT2   = 0;


  __SE0_OPEN((void*)r_vec, se_params);
  __SE1_OPEN((void*)gb_vec, se_params2);

}
```

Zatim je dobavljanje blokova moguće pozivom sljedeće funkcije.

```
extern "C" void fetch_next_blocks(int8_t* y_output, uint16_t num_blocks) {

  char32 * vec_y_out = (char32 *) y_output;

  short32 coeff_r = (short32) 77;
  short32 coeff_g = (short32) 150;
  short32 coeff_b = (short32) 29;
  short32 val_128 = (short32) 128;

  uint16_t i = 0;
  for(i = 0; i < num_blocks; i++) {
    // Fetch the R components from the first one
    uchar32 r_in_1 = strm_eng<0, uchar32>::get_adv();
    uchar64 gb_in_1 = strm_eng<1, uchar64>::get_adv();

    // Upcasting
    short32 r_s_1 = __convert_short32(r_in_1);
    short32 g_s_1 = __convert_short32(gb_in_1.lo);
    short32 b_s_1 = __convert_short32(gb_in_1.hi);

    short32 y_temp_1 = (r_s_1 * coeff_r) + (g_s_1 * coeff_g) + (b_s_1 * coeff_b);
    y_temp_1 = (y_temp_1 >> 8) - val_128;

    vec_y_out[2*i + 0] = __convert_char32(y_temp_1);

    // We repeat everything, but for the second half of the block
```

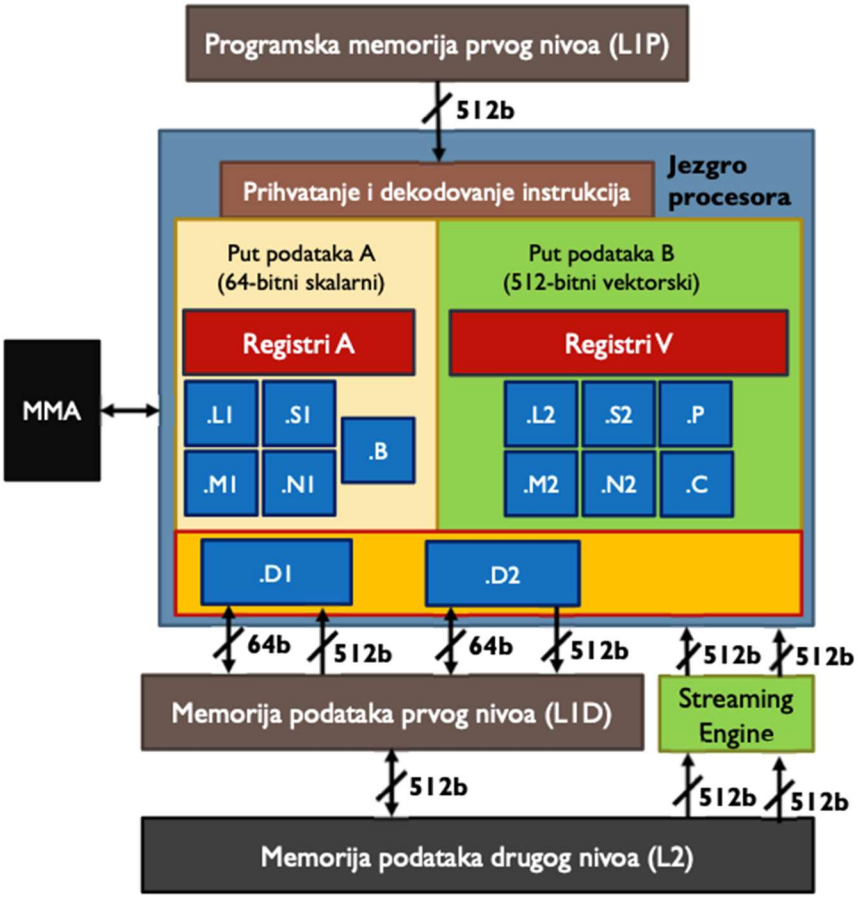
```
    uchar32 r_in_2 = strm_eng<0, uchar32>::get_adv();
    uchar64 gb_in_2 = strm_eng<1, uchar64>::get_adv();

    short32 r_s_2 = __convert_short32(r_in_2);
    short32 g_s_2 = __convert_short32(gb_in_2.lo);
    short32 b_s_2 = __convert_short32(gb_in_2.hi);

    short32 y_temp_2 = (r_s_2 * coeff_r) + (g_s_2 * coeff_g) + (b_s_2 * coeff_b);
    y_temp_2 = (y_temp_2 >> 8) - val_128;

    vec_y_out[2*i + 1] = __convert_char32(y_temp_2);

  }

}
```

Ovo dovodi do dobrih perfomansi.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                        |                CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion (segm, Y conv, cent.)     |               623,723 |
[C7x_1 ] | DCT Transform (Total)                         |               846,312 |
[C7x_1 ] | Quantization                                  |            85,948,365 |
[C7x_1 ] | ZigZag Reorder                                |             1,916,970 |
[C7x_1 ] | Huffman Encoding                              |             7,864,028 |
[C7x_1 ] ------------------------------------------------------------------------
[C7x_1 ] | TOTAL CYCLES                                  |            97,199,398 |
[C7x_1 ] ========================================================================
```

Vidimo da sve faze obrade benefituju od keširanja blokova.

Možemo da izvedemo jedan interesantan zaključak. Vidimo da su performanse DCT obrade iste, iako više ne koristimo *SE* jedinice za dobavljanje blokova za obradu, već se vrši keširanje. Ovo je konzistentno sa mikroarhitekturom C7x procesora, jer je L1 keš na istoj "udaljenosti" (isti penal performansi pri dobavljanju) kao i *SE* jedinica.



Slika 3.1 – ***Arhitektura C7000 procesora***

3.5. Optimizacija kvantizacije

Kvantizacija DCT koeficijenata funkcioniše na vrlo jednostavan način. Osnovni zadatak je podijeliti vrijednost DCT koeficijenta (*floating point* vrijednosti) sa vrijednošću iz matrice kvantizacije, te rezultat zaokružiti na najbliži cijeli broj. Međutim, iako vrlo jednostavan, ovaj proces nailazi na par ograničenja po pitanju performansi.

```
void quantize_block(float *dct_block, int16_t* out_quantized_block) {
  uint32_t i = 0;
  for(i = 0; i < 64; i++) {
    out_quantized_block[i] = (int16_t)roundf(dct_block[i] / std_lum_qt[i]);
  }
}
```

Pored poziva *roundf* funkcije, ovdje zapravo postoji još jedan funkcijski poziv. To se može vidjeti iz asemblerskih anotacija.

```
;*----------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*      Loop found in file          : /home/markos/Desktop/jpeg-compression/ti/service/src/quantization.c
;*      Loop source line            : 6
;*      Loop opening brace source line   : 6
;*      Loop closing brace source line   : 8
;*      Disqualified loop: Loop contains a call
;*      Disqualified loop: Loop contains non-pipelinable instructions
;*      Disqualified loop: Loop contains a call
;*      Disqualified loop: Loop contains non-pipelinable instructions
;*----------------------------------------------------------------------*
```

U pitanju je poziv **ugrađene funkcije** (funkcija *runtime* biblioteke). Ovaj funkcijski poziv nastaje automatski zbog operacije dijeljenja. Kompajler zamjenjuje operaciju dijeljenja sa softverskom emulacijom iste, te pravi funkcijski poziv prilikom prevođenja. Ovo za posljedicu ima diskvalifikaciju petlje za *Software pipelining.*

Možemo da formulišemo dijeljenje na drugačiji način.

$$\frac{F(u,v)}{Q(u,v)} = F(u,v) \cdot \left(\frac{1}{Q(u,v)}\right) \tag{3.2}$$

Reformulacija dijeljenja dovodi do poboljšanja performansi.

```
[C7x_1 ] | Quantization                                |            17,781,234 |
```

Međutim, i dalje je potrebno vršiti dijeljenje za računanje recipročne vrijednosti članova matrice

kvantizacije. S obzirom na činjenicu da su u pitanju konstantne vrijednosti koje se računaju svaki put iznova, logičan slijed je da redefinišemo tabelu koeficijenata matrice kvantizacije da sadrži već izračunate recipročne vrijednosti, a zatim da vršimo množenje istih sa DCT koeficijentima. Za računanje tabele iskorištena je *Python* skripta koja generiše potrebne C fajlove, a koji se uključuju u *build* sistem.

Postižemo određeno poboljšanje performansi.

```
[C7x_1 ] | Quantization                                  |            15,892,559 |
```

I dalje je prisutan poziv funkcije *roundf* što spriječava sprovođenje *Software pipelining*-a. Direktnim definisanjem logike za zaokruživanje dobijamo značajno poboljšanje performansi, jer je *Software pipeline* uspješno odrađen.

```
[C7x_1 ] | Quantization                                  |               383,882 |
```

Kao i kod optimizacija prethodnih koraka obrade, i ovdje primjena SIMD obrade donosi značajno poboljšanje. Da bi izbjegli grananje prilikom zaokruživanja, koristimo predikaciju vektora i selekciju upotrebom **intrinzičnih funkcija**.

```
void quantize_block(float* dct_block, int16_t* out_quantized_block) {
   ASSERT_ALIGNED_64(dct_block);
   ASSERT_ALIGNED_64(out_quantized_block);

   uint8_t i = 0;
   float16* input = (float16*)dct_block;
   short16* out = (short16*)out_quantized_block;
   float16* dct_table = (float16*)std_lum_qt_recip;

   float16 zeros = (float16)0.0f;
   float16 plus_half = (float16)0.5f;
   float16 minus_half = (float16)-0.5f;

   #pragma MUST_ITERATE(4, 4, 4)
   #pragma UNROLL(2)
   for(i = 0; i < 4; i++) {
      float16 in = *(input + i);
      float16 dct_in = *(dct_table + i);
      float16 result = in * dct_in;

      // Use __vpred for storing an array of bools
      __vpred pred = __cmp_ge_pred(result, zeros);
      float16 offset = __select(pred, plus_half, minus_half);

      *(out + i) = __convert_short16(result + offset);
   }

}
```

Međutim, profilisanjem ipak dobijamo loše performanse. Pregledom asemblerskih anotacija, identifikujemo glavni uzrok.

```
;*----------------------------------------------------------------------------*
;*   SOFTWARE PIPELINE INFORMATION
;*
;*     Loop found in file            : /home/markos/Desktop/jpeg-compression/ti/service/src/quantization.c
;*     Loop source line              : 15
;*     Loop opening brace source line   : 15
;*     Loop closing brace source line   : 26
;*     Known Minimum Iteration Count    : 4
;*     Known Maximum Iteration Count    : 4
;*     Known Max Iteration Count Factor : 4
;*     Loop Carried Dependency Bound(^) : 24
;*     Partitioned Resource Bound       : 2 (pre-sched)
;*
;*     Searching for software pipeline schedule at ...
;*        ii = 24 Schedule found with 2 iterations in parallel
;*
;*     Partitioned Resource Bound(*)    : 2 (post-sched)
;*
```

Vidimo da je usko grlo *Loop Carried Dependency Bound* koji predstavlja teorijski minimalnu vrijednost *Initiation Interval* (*II*) vrijednosti s obzirom na zavisnost podataka između iteracija petlje. Ova vrijednost je uzrokovana *aliasing*-om, koji nastaje kao posljedica toga što kompajler mora konzervativno da pristup optimizaciji i ne smije da pretpostavi da pokazivači na izlazni i ulazni bafer zaista pokazuju na različite memorijske lokacije. Kompajler u softverski pajplajnovanoj petlji ne smije da započne instrukcije učitavanja ulaznih podataka naredne iteracije prije nego što se završi operacija skladištenja rezultata u izlazni bafer prošle iteracije.

Srećom, riješenje je trivijalno. Dovoljno je uvesti ključnu riječ *restrict* na parametre funkcije da se kompajler informiše da pokazivači ne pokazuju na preklapajuće memorijske lokacije. Pored ključne riječi *restrict*, mogu se koristiti i odgovarajuće kompajlerske opcije, kao i *pragma* direktive, međutim, one funkionišu na nivou programa, dok *restrict* ključna riječ omogućuje finu granularnost.

Parametri *pragma* direktiva nad petljama su, takođe, određeni empirijskim procesom.

Poboljšanje je vidno.

```
[C7x_1 ] | Quantization                                |               156,904 |
```

Za sprovođenje dalje optimizacije, analiziramo asemblerske anotacije.

```
;*    Resource Partition (may include "post-sched" split/spill moves):
;*
;*                                A-side   B-side
;*    Bound(.C2)                         -      1
;*    Bound(.P2)                         -      0
;*    Bound(.D)                         2*      -
;*    Bound(.M .N .MN)                     0      1
;*    Bound(.L .S .LS)                     0      1
;*    Bound(.L .S .C .LS .LSC)             0      2*
;*    Bound(.L .S .C .M .LS .LSC .LSCM)    0      2*
;*    Bound(.L .S .C .M .D .LS .LSC .LSCM .LSCMD)  1      2*
;*
;*    Done
```

Problem je što prilikom obrade svakog bloka učitavamo i DCT koeficijente (16 *float* vrijednosti, što je ukupno 512 bita), kao i matricu kvantizacije (16 *float* vrijednosti, ukupno 512 bita). Ovo prevazilazi količinu podataka koje .D jedinica može da dobavi u jednom ciklusu, jer .D jedinica može da vektorsku stranu opsluži sa 512 bita po ciklusu. Imamo dvije instrukcije učitavanja koje se moraju sekvencijalno izvršiti.

Poboljšanje možemo da postignemo ako vršimo obradu više blokova slike od jednom, a matricu kvantizacije promovišemo u registre.

Reformulacijom da se vrši obrada dva bloka u jednom pozivu funkcije već dobijamo bolje performanse.

```
[C7x_1 ] | Quantization                                |              113,293 |
```

3.6. Hibridni pristup procesiranju

Da bismo iskoristili najbolje od oba pristupa blokovskom procesiranju, koristićemo hibridni pristup koji se zasniva na vertikalnoj obradi grupe blokova. Na ovaj nacin, kombinujemo pogodnost manjih bafera blizu procesora za reprezentaciju grupe blokova, ali ostvarujemo i mogućnost keširanja *lookup* tabela u fazama obrade.

Reformulacijom obrade tako da se obrađuju dva bloka u jednoj grupi već postižemo poboljšanja performansi.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                        |                CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion (segm, Y conv, cent.)     |               622,899 |
[C7x_1 ] | DCT Transform (Total)                         |               810,352 |
[C7x_1 ] | Quantization                                  |               113,207 |
[C7x_1 ] | ZigZag Reorder                                |             1,917,113 |
[C7x_1 ] | Huffman Encoding                              |             7,904,246 |
[C7x_1 ] ------------------------------------------------------------------------
[C7x_1 ] | TOTAL CYCLES                                  |            11,367,817 |
[C7x_1 ] ========================================================================
```

Od ove izmjene najviše benefituju one funkcije koje imaju potrebu učitavanja *lookup* tabele pri obradi, kao što je kvantizacija.

Empirijskim procesom isprobavanja različitih veličina grupe blokova, identifikujemo lokalni minimum za veličinu grupe od 32 bloka.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                        |                CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion (segm, Y conv, cent.)     |               611,039 |
[C7x_1 ] | DCT Transform (Total)                         |               808,296 |
[C7x_1 ] | Quantization                                  |                62,567 |
[C7x_1 ] | ZigZag Reorder                                |             1,895,445 |
[C7x_1 ] | Huffman Encoding                              |             7,866,986 |
[C7x_1 ] ------------------------------------------------------------------------
[C7x_1 ] | TOTAL CYCLES                                  |            11,244,333 |
[C7x_1 ] ========================================================================
```

3.7. Optimizacija cik-cak permutacije

Analizom asemblerskih anotacija za cik-cak optimizaciju, možemo da vidimo da je najveće ograničenje *Loop Carried Dependency Bound*. Da bi spriječili *aliasing*, uvodimo *restrict* ključnu riječ na parametre funkcije, što je dovelo do poboljšanja.

[C7x_1] | ZigZag Reorder	|	690,955 |

Trenutna implementacija vrši skalarnu permutaciju prema predefinsanom nizu indeksa pristupa u cik-cak poretku. Analizom asemblerskih anotacija možemo da identifikujemo da se opterećuje A (skalarna) strana.

Znamo da C7x procesori imaju namjensku jedinicu koja vrši permutaciju vektorskih elemenata - .C jedinica. Međutim, postoji ograničenje za naš slučaj upotrebe. Naš cilj je permutacija niza od 64 elementa tipa *uint16_t.* Ovaj niz ima ukupnu dužinu od 1024 bita, dok je dužina jednog vektorskog registra 512 bita. Da bi se prevazišlo ovo ograničenje, potrebno je izvršiti segmentaciju ulaznog niza u dva niza dužine 64 elementa tipa *uint8_t,* te izvršiti parcijalne permutacije nad istima i naposlijetku objediniti dobijene rezultate.

Osnovna ideja je da se adresira svaki bajt, te da se od jednog cik-cak niza indeksa naprave dva. Prvi sadrži prvih 32 elementa originalnog niza, ali je proširen na takav način da je svaki element originalnog niza praćen njegovim inkrementom (jer sada adresiramo pojedinačne bajte). Drugi sadrži preostalih 32 elementa originalnog niza, proširenih na isti način.

```
void init_zigzag(void) {
  uint8_t temp_lo[64];
  uint8_t temp_hi[64];
  int i;

  for (i = 0; i < 32; i++) {
    uint8_t src_idx = zigzag_map[i];
    temp_lo[2 * i]     = (uint8_t)(src_idx * 2);
    temp_lo[2 * i + 1] = (uint8_t)(src_idx * 2 + 1);
  }

  for (i = 32; i < 64; i++) {
```

```
        uint8_t src_idx = zigzag_map[i];
        int j = i - 32;
        temp_hi[2 * j]     = (uint8_t)(src_idx * 2);
        temp_hi[2 * j + 1] = (uint8_t)(src_idx * 2 + 1);
    }

    perm_mask_lo = *((uchar64 *)temp_lo);
    perm_mask_hi = *((uchar64 *)temp_hi);
}
```

Zatim se vrši permutacija prve i druge polovine ulaznog bloka. Permutacija se vrši dva puta, jednom za prvu polovinu cik-cak niza indeksa, drugi put za drugu polovinu. Indeksi veći od 63 za indeksiranje elemenata u polovinama blokova neće biti problem, jer *___vperm_vvv* intrinzik koristi adresiranje po modulu dužine vektora. Na kraju se rezultati objedinjuju na osnovu toga da li prava vrijednost elementa pripada prvom, ili drugom nizu.

```
static inline uchar64 vperm_byte_wrapper(uchar64 mask, uchar64 src_lo, uchar64 src_hi) {
    // Input data is short32. We can't operate directly on shorts, so we perform permutation on byte level.
    // We use extended zig-zag table which addresses each byte of input array and permutates it.
    // Because this extended zig-zag table has length of 128, we need to perform this permutation twice, once for each
half of zigzag table.

    uchar64 p1 = __vperm_vvv(mask, src_lo);
    uchar64 p2 = __vperm_vvv(mask, src_hi);

    // p2 has the upper addresses.
    // p1 has the lower addresses.
    // __vperm_vvv reads addreses by modulo of 64, so it never "leaves" the input array.
    // We need to determine which array holds the right value for each index.
    // For indexes lower than 64, p1 has the correct value, and vice versa.
    __vpred pred = __cmp_gt_pred(mask, (uchar64)63);

    return __select(pred, p2, p1);
}
```

Ova funkcija se poziva iz funkcije za procesiranje blokova sa spremljenim segmentiranim nizovima. Poboljšanje performansi je značajno.

[C7x_1]	ZigZag Reorder	22,196

3.8. Optimizacija kodovanja

Kodovanje zahtjeva manipulaciju podataka na bitskom nivou, kao što je opisano u glavi 1. Ovo zahtjeva definisanje namjenskih struktura podataka i funkcija koje sprovode operacije na nivou bita.

```
typedef struct {
    uint8_t *buffer;   // Buffer in which we write encoded coefficients
    uint32_t byte_pos;  // Current byte in the buffer
     uint32_t bit_pos;   // Current bit position in the current byte (0-7)
    uint8_t current;   // Current byte being constructed
} BitWriter;
```

Pokušao sam da deklarišem pomoćne funkcije za manipulacijom bitskih operacija kao *static inline,* međutim, nije bilo razlike u performansama. Ovo je i očekivano, jer je kompajler pokrenut sa *O3* nivoom optimizacije, što je prouzrokovalo agresivne optimizacije i *inline*-ovanje funkcija implicitno.

Izmjenom *BitWriter* strukture da se obrađuje podatak tipa *uin64_t* prije upisa u memoriju, postižu se poboljšanja jer se ne vrši skladištenje u memoriju toliko često.

[C7x_1] | Huffman Encoding	|	5,167,013 |

Pokušaj izmjene petlje za RLE kodovanje tako da se unaprijed odredi indeks zadnjeg ne-nula elementa u cilju smanjenja broja iteracija nije doveo do poboljšanja, već, štaviše, do pogoršanja performansi. Kompajleri više vole fiksan broj iteracija i niz optimizacija koje mogu da sprovedu na osnovu poznatog, fiksnog broja iteracija nadjačava trošak dodatnih iteracija.

Prelaskom na vektorsku obradu, moguće je paralelno identifikovati sve ne-nula elemente, te iterirati kroz niz dok god postoje ne-nula elementi. Za ekstrakciju i manipulaciju pojedinih elemenata koriste se bitske operacije. Poboljšanja su vidna.

[C7x_1] | Huffman Encoding	|	2,344,537 |

Uvođenjem kodovanja u hibridno procesiranje postižemo dodatna poboljšanja zbog keširanja grupe blokova i, samim tim, efikasnijim pristupom istim u cilju sprovođenja operacija u toku kodovanja.

Dolazimo do ostvarenog lokalnog minimuma projekta.

```
[C7x_1 ] ========================================================================
[C7x_1 ] | JPEG COMPRESSION STAGE                          |              CYCLES |
[C7x_1 ] ========================================================================
[C7x_1 ] | RGB -> Y Conversion (segm, Y conv, cent.)       |             606,181 |
[C7x_1 ] | DCT Transform (Total)                           |             826,581 |
[C7x_1 ] | Quantization                                    |              67,928 |
[C7x_1 ] | ZigZag Reorder                                  |              25,093 |
[C7x_1 ] | Huffman Encoding                                |           1,775,941 |
[C7x_1 ] ------------------------------------------------------------------------
[C7x_1 ] | TOTAL CYCLES                                    |           3,301,724 |
[C7x_1 ] ========================================================================
```

3.9. Neuspjeli pokušaji optimizacije

Pokušao sam dodatno da optimizujem DCT obradu upotrebom *SA* jedinica tako da ostvarim cirkularno adresiranje ulaznih i izlaznih bafera, međutim, nisam uspio da konfigurišem iste na pravi način. Zbog vremenskog ograničenja, odustao sam od ovog pristupa. Cilj je bio da se *SA* jedinice jednom konfigurišu, a koriste više puta u obradi.

```
void dct_init(int8_t num_blocks, uint32_t total_pixels) {
  __SA_TEMPLATE_v1 sa_tmplt = __gen_SA_TEMPLATE_v1();
  sa_tmplt.DIMFMT = __SA_DIMFMT_2D;
  sa_tmplt.VECLEN = __SA_VECLEN_8ELEMS;
  sa_tmplt.ICNT0  = num_blocks * 64;// total of num_blocks * 64 elements to fetch
  sa_tmplt.ICNT1  = total_pixels;
  sa_tmplt.DIM1   = 0;  // reset it to base address so we get circular addresing

  __SA0_OPEN(sa_tmplt);
  __SA1_OPEN(sa_tmplt);
}
```

Pokušao sam uključiti *O4* nivo optimizacije (na nivou programa), međutim, potrebne informacije je kompajler već imao u vidu ključnih riječi, raznih *pragma* direktiva i poziva ugrađene *__nasert*

funkcije, tako da *O4* nije doveo ni do kakvih poboljšanja. Pokušao sam uključiti i *–opt_for_speed=5*, ali ni to nije dovelo do poboljšanja.

Izmjena linkerskih skripti bi omogućila alokaciju bafera u SRAM memoriji i izbjegavanje nedeterminističkih rezultata profilisanja zbog keširanja, međutim, *Vision Apps* koristi svoje linkerske skripte koje su jako rigidne i čak i najmanje izmjene (na nivou par bajtova) dovode do nemogućnosti kompajliranja.

4. Moguća poboljšanja

Izmjena linkerskih skripti bi omogućila alokaciju bafera u SRAM memoriji i izbjegavanje nedeterminizma u kontekstu brzine izvršavanja.

C7x procesor raspolaže sa eksternom *Matrix Multiplication Accelerator* (*MMA*) jedinicom. Konfiguracijom ove jedinice bi mogli optimizovati matrično množenje koje je prisutno u DCT obradi.

Naprednom konfiguracijom *SE* jedinice bi se moglo omogućiti fleksibilnije adresiranje ulaznih bafera, te ne bi bilo potrebno raditi segmentaciju slike na *host* strani.

Realizacija algoritma upotrebom *fixed point* aritmetike bi omogućila efikasniju obradu podataka jer se *fixed point* reprezentacija efektivno realizuje upotrebom binarne aritmetike, te je kao takva dosta efikasnija za sprovođenje od FP aritmetike. Bitno je napomenuti da bi u ovom slučaju trebali voditi računa o dinamičkom opsegu formata podataka.

5. Analiza rezultata i evaluacija kvaliteta kompresije

Cilj analize rezultata i evaluacije kvaliteta kompresije jeste određivanje kvantifikacije nastale degradacije upotrebom objektivnih matematičkih metrika. JPEG kompresija je kompresija podataka sa gubicima (*lossy compression*) i kao takva uvodi ireverzibilne izmjene frekvencijskog spektra. Uvijek postoji kompromis između veličine kompresovanog sadržaja i njegove vjerodostojnosti originalu.
Koristićemo tri objektivne matematičke metrike.

- *Mean Squared Error – MSE*

Srednja kvadratna greška (Mean Squared Error - MSE) predstavlja jednu od osnovnih objektivnih mjera za procjenu kvaliteta rekonstruisane slike u odnosu na original. U kontekstu JPEG kompresije, MSE kvantifikuje prosječno kvadratno odstupanje intenziteta piksela dekodirane slike u odnosu na originalnu sliku.

$$MSE = \frac{1}{M \cdot N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - K(i,j)]^2 \tag{5.1}$$

Vrijednost u opsegu [0, 100] predstavlja prihvatljiv nivo za JPEG kompresiju.

- *Peak Signal-to-Noise Ration – PSNR*

PSNR (Peak Signal-to-Noise Ratio) predstavlja najčešće korištenu metriku za objektivnu evaluaciju kvaliteta slike nakon kompresije sa gubicima. Ova mjera definiše odnos između maksimalne moguće snage signala (originalne slike) i snage šuma (greške nastale kompresijom) koji utiče na vjernost njene reprezentacije.

$$PSNR = 10 \cdot \log_{10}\left(\frac{MAX_I^2}{MSE}\right) \tag{5.2}$$

Pri čemu je MAX maksimalna moguća vrijednost piksela (255 u našem slučaju).

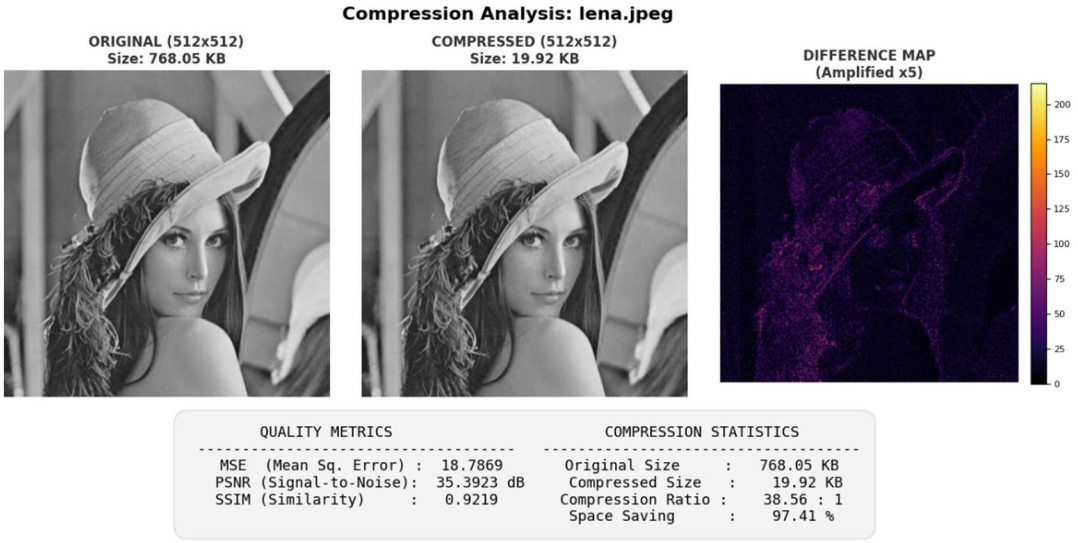
Vrijednost u opsegu [30, 40] dB predstavlja industrijski standard i prihvatljiv nivo odnosa snage signala i šuma.

- *Structural Similarity Index Measure – SSIM*

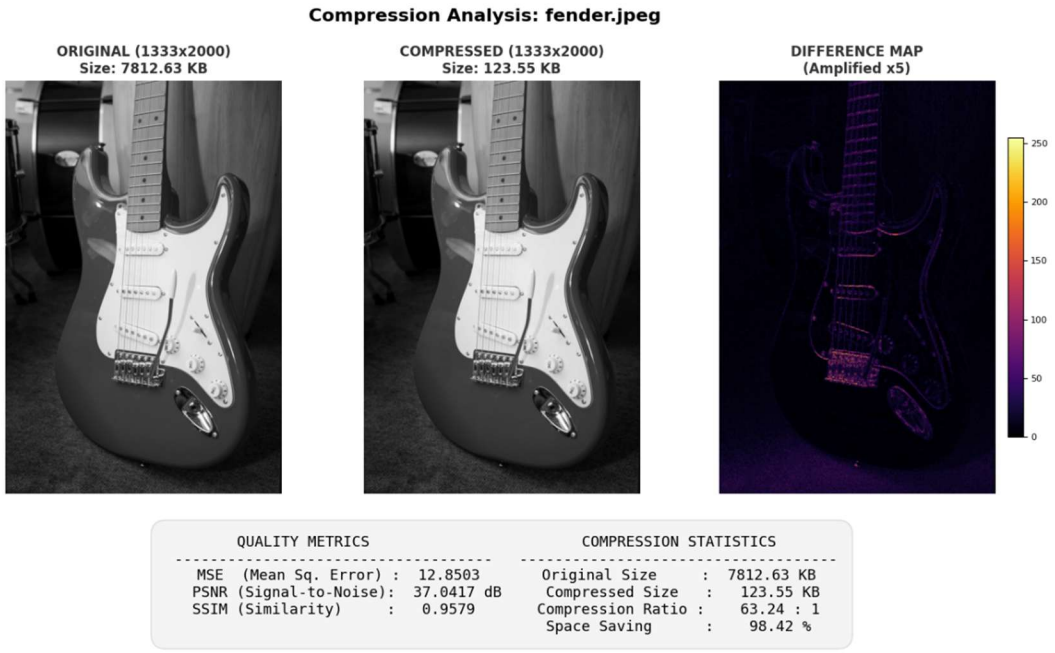
Indeks strukturalne sličnosti (Structural Similarity Index Measure - SSIM) je metoda dizajnirana da oponaša ljudski vizuelni sistem. Ljudsko oko je visoko prilagođeno izvlačenju strukturnih informacija iz vizuelne scene, a ne računanju grešaka u pojedinačnim tačkama.

SSIM mjeri sličnost između dvije slike poređenjem tri ključne karakteristike: luminancija, kontrast i struktura.

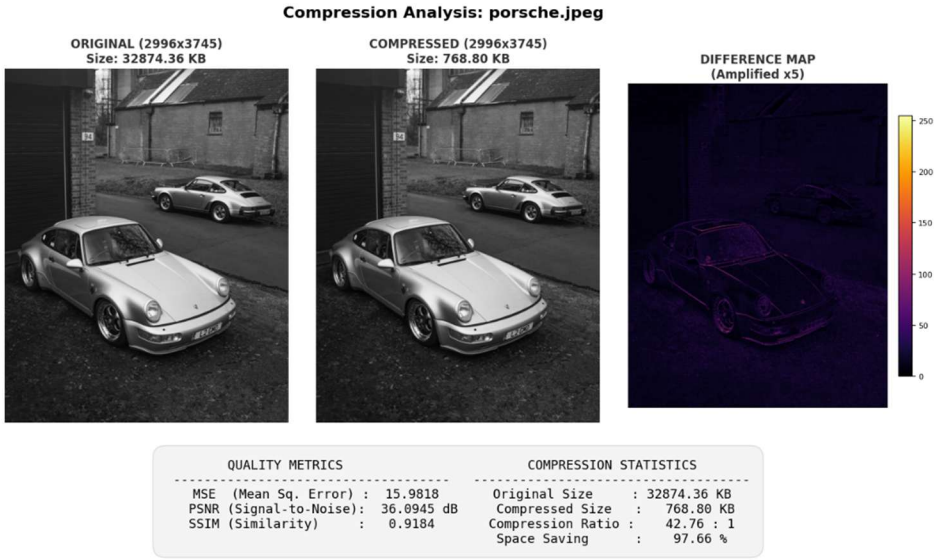
Vrijednost u opsegu [0.95, 1] predstavlja izvrsnu strukturalnu sličnost, dok se sve iznad 0.9 smatra dobrim rezultatom.



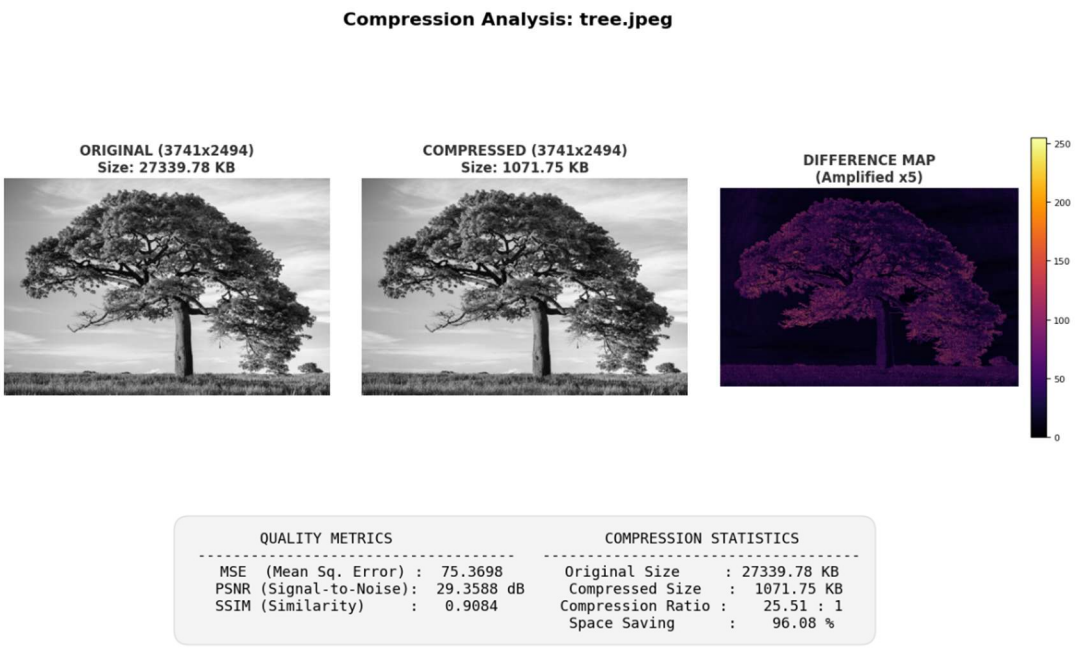
Slika 5.1 - ***Analiza lena.bmp***



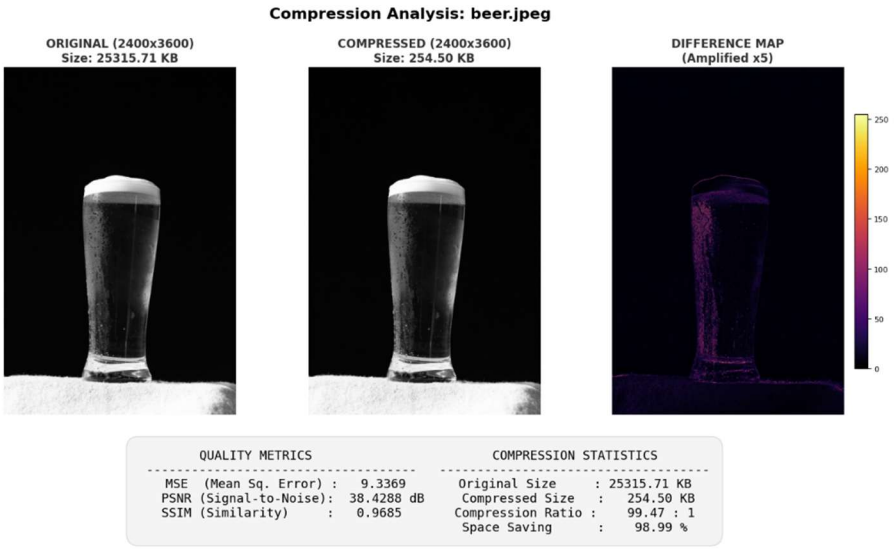
Slika 5.2 - ***Analiza fender.bmp***



Slika 5.3 - ***Analiza porsche.bmp***



Slika 5.4 - ***Analiza tree.bmp***



Slika 5.5 - ***Analiza beer.bmp***

6. Zaključak

Cilj ovog projekta je realizacija algoritma za JPEG kompresiju na platformi za digitalnu obradu signala, s fokusom na iskorištenje pogodonosti koje takva platforma pruža, te ostvarivanju maksimalnog stepena redukcije uz prihvatljiv kvalitet rekonstruisanog signala.

JPEG kompresija se zasniva na primjeni DCT transformacije, te zbog svoje tendencije da koncentriše energiju u niskofrekvencijskim komponentama predstavlja dobar izbor za reprezentaciju signala i kompresiju istog. Kao što je već rečeno u glavi 1, JPEG iskorištava činjenicu da ljudsko oko dobro identifikuje niskofrekvencijske vizuelne komponente, dok je slabije osljetljivo na visokofrekvencijske. Polazeći od ove činjenice, moguće je eliminisati kvantizacijom visokofrekvencijske komponente, a zadržati suštinu samog signala, odnosno slike.

Primjenom JPEG kompresije možemo da redukujemo količinu podataka potrebnih za reprezentaciju informacije, te omogućimo efikasnije skladištenje signala ili prenos istog preko nekog komunikacionog linka. Odnos kompresije je varijabilan i zavisi od frekvencijskog spektra ulazngo signala, ali je njegova prosječna vrijednost 40:1, čime se ulazni signala svode na nešto ispod 0.6 bita po pikselu.

Poznavajući prednosti DSP platforme na kojoj je projekat realizovan, uspješno je sproveden niz optimizacija koje su omogućile smanjenje broja ciklusa potrebnih za obradu slike za skoro 3 reda veličine (965,37 puta).

Za Full HD fotografiju je potrebno približno 35 miliona ciklusa, što se na gigahercnom procesoru prevodi na vremenski interval od 0,035 sekundi po slici. Ovaj rezultat nalaže da je moguće obraditi nepunih 29 Full HD slika u jednoj sekundi (28, 57 slika), što je pogodono za obradu u realnom vremenu.

7. Reference

[1] ISO/IEC, 1993. [Na mreži]. Available: https://www.w3.org/Graphics/JPEG/itu-t81.pdf.

[2] ITU-R, 1982. [Na mreži]. Available: https://en.wikipedia.org/wiki/Rec._601.