Elektrotehnički fakultet
Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student:
Štrbac Damjan 1139/22

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

Januar 2026. godine

1. Opis projektnog zadatka

U sklopu projektnog zadatka potrebno je realizovati sistem za kompresiju grayscale slika korištenjem diskretne kosinusne transformacije. Kompresija je bazirana na JPEG standardu za kompresiju.

Proces kompresije se sastoji iz sljedećih koraka:

Ekstrakcija Y komponente: Budući da se radi o monohromatskoj (grayscale) kompresiji, proces započinje izolacijom kanala luminanse (Y). Time se eliminišu informacije o hrominansi (boji), a slika se svodi na dvodimenzionalnu matricu intenziteta svjetlosti, čime se početna količina podataka odmah redukuje na najbitniju vizuelnu komponentu.

Segmentiranje slike na blokove 8×8 i centriranje vrijednosti: Slika se dijeli na nezavisne blokove dimenzija 8×8 piksela koji predstavljaju osnovne jedinice obrade. Vrijednosti piksela unutar svakog bloka (inicijalno u opsegu [0,255]) se centriraju oko 0 oduzimanjem vrijednosti 128, čime se dobija simetričan opseg([−128,127]).

Primjena 2D diskretne kosinusne transformacije (DCT): Na svaki pripremljeni blok primjenjuje se 2D DCT, čime se signal konvertuje iz prostornog domena (vrijednosti piksela) u frekvencijski domen (spektralni koeficijenti). Rezultat je matrica u kojoj gornji lijevi ugao sadrži niskofrekventne komponente koje nose većinu energije slike, dok se ka donjem desnom uglu nalaze visokofrekventni detalji manjeg vizuelnog značaja.

Kvantovanje koeficijenata: Svaki DCT koeficijent dijeli se odgovarajućim elementom iz definisane matrice kvantizacije, a rezultat se zaokružuje na najbliži cijeli broj. Ovim postupkom se prigušuju ili potpuno eliminišu (svode na nulu) visokofrekventni koeficijenti na koje je ljudsko oko manje osjetljivo. U ovom koraku dolazi do gubitka informacija.

Cik-cak (Zig-zag) permutacija: Kvantovana matrica se linearno iščitava cik-cak putanjom, počevši od DC koeficijenta (gore-lijevo). Cilj ove permutacije je grupisanje nula, koje su dominantne u visokofrekventnom dijelu matrice, na kraj jednodimenzionalnog niza, što značajno povećava efikasnost *Run-Length* kodovanja.

Kodovanje koeficijanata: DC koeficijent (1. element niza) se koduje kao razlika u odnosu na DC koeficijent prethodnog bloka, dok se preostala 63 AC koeficijenta koduju RLE metodom (*Run-*

Length Encoding) koja efikasno zapisuje duge nizove nula nastalih kvantizacijom. Konačni niz simbola (parovi: broj nula, vrijednost) podvrgava se Hafmanovom kodovanju, gdje se češćim simbolima dodjeljuju kraći binarni kodovi, čime se formira finalni *bitstream* i generiše .jpg datoteka.

2. Izrada projektnog zadatka

2.1 Predprocesiranje

Ova faza obuhvata učitavanje izvorne slike u BMP formatu i reorganizaciju piksela iz standardnog rasterskog zapisa u strukturu pogodnu za blokovsku obradu. Proces započinje dekompozicijom slike na tri nezavisna kanala boja (R, G i B), koji se inicijalno učitavaju kao jednodimenzionalni nizovi.

S obzirom na to da JPEG standard operiše nad blokovima veličine 8×8 piksela, podaci se permutuju tako da se u memoriji sekvencijalno zapisuju kompletni blokovi. Algoritam iterira kroz sliku blok po blok, a unutar svakog bloka pikseli se slažu linearno. U slučajevima kada dimenzije slike nisu cjelobrojno djeljive sa 8, primjenjuje se tehnika proširivanja (engl. padding) replikacijom ivičnih piksela (engl. clamping).

Nakon formiranja linearnih blokova, vrši se formatiranje ulaznih bafera prilagođeno arhitekturi digitalnog signalnog procesora (DSP), gdje se R kanal zadržava u formatu linearnih blokova i kopira se direktno u izlazni memorijski prostor, dok se G i B kanali isprepliću u jedan zajednički memorijski blok. Struktura ovog bloka je organizovana tako da se podaci alterniraju u segmentima od 32 uzorka (bajta). Ovo je urađeno jer C7x DSP arhitektura posjeduje dva integrisana Streaming Engine-a (SE). Ovi namjenski hardverski blokovi služe za automatsko upravljanje tokovima podataka, preuzimajući zadatak dovlačenja podataka iz memorije u vektorske registre nezavisno od glavne procesorske jedinice. Ovakva organizacija memorije (interleaving) omogućava Streaming Engine-ima da, koristeći definisane obrasce pristupa, efikasno razdvoje i paralelizuju učitavanje G i B komponenti, čime se maksimizira memorijska propusnost i osigurava da izvršne jedinice DSP-a budu konstantno snabdjevene podacima bez čekanja (stall).

Treba napomenuti da se ova pripremna faza, koja uključuje učitavanje, segmentaciju i reorganizaciju memorije, u potpunosti izvršava na ARM Cortex-A72 procesoru (Host). Nakon što su

podaci formatirani u odgovarajuće R i GB bafere, oni se putem dijeljene memorije prosljeđuju C7x DSP jezgru, koje je zaduženo za izvršavanje numerički intenzivnih operacija kompresije. Cortex-A72 je također zadužen za serijalizaciju kompresovanih podataka dobijenih od strane C7x DSP-a u JFIF format. U svrhu analize dobijenih rezultata i validacije performansi, kao ulazni signal je korištena referentna slika *grad.bmp*, koja se može naći u izvornom kodu.

2.2 Ekstrakcija Y komponente

Nakon prebacivanja podataka na DSP, pristupa se konverziji prostora boja. Kako je cilj projekta kompresija monohromatskih (grayscale) slika, potrebno je iz originalnog RGB prostora izdvojiti samo komponentu osvjetljenja, odnosno luminancu (Y). Prema JPEG standardu (ITU-T T.81), luminanca se računa kao težinska suma R, G i B komponenti:

$$Y = 0.299 \cdot R + 0.587 \cdot G + 0.114 \cdot B \tag{2.1}$$

S obzirom na to da C7x DSP arhitektura postiže maksimalne performanse pri radu sa cjelobrojnom aritmetikom, implementacija ne koristi operacije sa pokretnim zarezom. Umjesto toga, koeficijenti su skalirani faktorom 256, čime se omogućava izvođenje operacija korištenjem brze cjelobrojne aritmetike i bit-shift operacija. Aproksimirana formula implementirana na DSP-u glasi:

$$Y = \frac{77 \cdot R + 150 \cdot G + 29 \cdot B}{256} \tag{2.2}$$

Nakon izračunavanja Y komponente, vrši se centriranje opsega vrijednosti oko 0. Originalni 8-bitni podaci su u opsegu [0,255] (neoznačeni cijeli brojevi). Kako bi se pripremili za diskretnu kosinusnu transformaciju (DCT), koja optimalno radi sa podacima centriranim oko nule, od svake vrijednosti piksela oduzima se vrijednost 128. Ovim postupkom se dobija označen opseg vrijednosti $[-128,127]$, koji se smješta u izlazni bafer spreman za transformaciju.

2.3 Dvodimenzionalna diskretna kosinusna transformacija

Cilj ovog koraka je transformacija signala iz prostornog domena (vrijednosti piksela) u frekventni domen. DCT posjeduje svojstvo kompakcije energije, što znači da se većina informacija slike (niskofrekventni sadržaj) koncentriše u gornjem lijevom uglu matrice (DC koeficijent), dok visokofrekventni detalji, na koje je ljudsko oko manje osjetljivo, opadaju prema donjem desnom uglu. Dvodimenzionalna DCT tipa II se definiše kao linearna transformacija koja mapira prostorne

odmjerke u koeficijente:

$$F(u,v)=\alpha(u)\,\alpha(v)\sum_{x=0}^{N-1}\sum_{y=0}^{N-1} f(x,y)\cos\left(\frac{\pi}{N}\left(x+\frac{1}{2}\right)u\right)\cos\left(\frac{\pi}{N}\left(y+\frac{1}{2}\right)v\right) \tag{2.3}$$

gdje su vrijednosti u i v date u rasponu od 0 do N – 1, a faktori normalizacije su:

$$\alpha(0)=\frac{1}{\sqrt{N}} \tag{2.4}$$

$$\alpha(k)=\sqrt{\frac{2}{N}},(k\geq 1) \tag{2.5}$$

Ovaj pristup je računarski zahtijevan, te je u realizaciji ovog projekta iskorišteno svojstvo separabilnosti transformacija, što omogućava da se 2D DCT transformacija izvede putem 2 operacije matričnog množenja. DCT matrica dimenzija NxN je definisana kao:

$$C_{u,x}=\alpha(u)\cos\left(\frac{\pi}{N}\left(x+\frac{1}{2}\right)u\right) \tag{2.6}$$

Sada se 2D DCT može zapisati kao proizvod:

$$F=C\,f\,C^{T} \tag{2.7}$$

Implementacija u projektu ovaj proces razbija na dva koraka, prvo se računa međurezultat množenjem ulaznog bloka sa transponovanom matricom transformacije, pa se zatim matrica transformacije množi sa međurezultatom. Ulazni 8-bitni podaci se konvertuju u format sa pokretnim zarezom radi očuvanja preciznosti tokom množenja, čime se smanjuje greška zaokruživanja.

2.4 Kvantizacija DCT koeficijenata

Nakon transformacije u frekventni domen, slijedi proces kvantizacije. Ovo je jedini nepovratni (lossy) korak u JPEG algoritmu, čiji je cilj smanjenje preciznosti visokofrekventnih koeficijenata na koje je ljudsko oko manje osjetljivo (psiho-vizuelna redundansa). Standardni postupak podrazumijeva dijeljenje svakog od 64 DCT koeficijenta sa odgovarajućom vrijednošću iz kvantizacione tabele:

$$\hat{F}(u,v)=round\left(\frac{F(u,v)}{Q(u,v)}\right) \tag{2.8}$$

Međutim, operacija dijeljenja je u pogledu procesorskih ciklusa izuzetno skupa, pa je

kvantizacija implementirana kao množenje sa recipročnim vrijednostima. Tabela recipročnih vrijednosti je unaprijed izračunata i statički alocirana u memoriji.

2.5 Cik-Cak permutacija

Nakon kvantizacije, podaci su i dalje organizovani u 2D matrici 8×8. Kako bi se optimizovala efikasnost narednog koraka (entropijskog kodovanja), neophodno je reorganizovati koeficijente u jednodimenzionalni niz na način koji grupiše nule.

Kako bi se postiglo navedeno koristi se Cik-Cak (eng. *Zig-Zag*) skeniranje. Ovaj algoritam kreće od DC koeficijenta (gornji lijevi ugao), te se pomjera dijagonalno kroz niske frekvencije i završava na najvišim frekvencijama (donji desni ugao).

Budući da je većina visokofrekventnih koeficijenata nakon kvantizacije postala nula, ovaj poredak osigurava da se te nule nađu na kraju niza u dugačkim sekvencama, što je idealno za Run-Length Encoding (RLE).

2.6 Kodovanje dužinom niza (RLE – Run Length Encoding)

Nakon linearizacije podataka Zig-Zag skeniranjem, ulazni niz se sastoji od DC koeficijenta (prosječno osvjetljenje bloka) i niza AC koeficijenata u kojem dominiraju nule. U ovoj fazi vrši se entropijsko kodovanje koje ima za cilj uklanjanje statističke redundanse. Proces se razlikuje za DC i AC koeficijente.

Diferencijalno kodovanje DC koeficijenata: S obzirom na to da susjedni blokovi slike često imaju sličan prosječni intenzitet, DC koeficijenti su visoko korelisani. Umjesto kodovanja apsolutne vrijednosti, koduje se razlika između DC koeficijenta trenutnog i prethodnog bloka:

$$DIFF = DC_i - DC_{i-1} \tag{2.9}$$

Ova razlika je obično vrlo mala, čime se omogućava efikasnija kompresija.

Run-Length Encoding (RLE) za AC koeficijente: AC koeficijenti, sortirani Zig-Zag putanjom, karakteristični su po dugim nizovima nula, posebno na kraju bloka. RLE algoritam zamjenjuje te nizove

parovima vrijednosti (Run,Size), gdje:

- Run: Broj uzastopnih nula koje prethode nenultom koeficijentu (0-15).
- Size/Amplitude: Vrijednost nenultog koeficijenta koji prekida niz

Ukoliko niz nula pređe 15, koristi se specijalni simbol ZRL (Zero Run Length) koji označava 16 uzastopnih nula. Kraj bloka, nakon kojeg slijede isključivo nule, označava se simbolom EOB (End of Block).U realizaciji projekta primijenjen je pristup korištenjem vektorskih predikata i bit-manipulacije, čime je eliminisana potreba za provjerom svake nule pojedinačno.

2.7 Huffmanovo kodovanje i formiranje izlaznog toka

Finalna faza JPEG kompresije podrazumijeva pretvaranje RLE simbola u konačni niz bitova (bitstream) korištenjem Huffmanovog kodovanja. JPEG standard (ISO/IEC 10918-1) definiše preporučene Huffmanove tabele za luminancu koje daju optimalne rezultate za većinu prirodnih slika. Tabele se generišu dinamički prilikom prve inicijalizacije sistema i keširaju u memoriji radi bržeg pristupa.

Generisani niz bajtova na DSP-u sadrži isključivo kompresovane podatke o slici, ali ne i meta-podatke neophodne da bi standardni preglednici slika prepoznali format. A72 procesor preuzima taj niz i vrši proces enkapsulacije u JFIF (JPEG File Interchange Format). To podrazumijeva dodavanje standardnih markera koji definišu parametre slike.

3. Optimizacije

3.1 Optimizacija ekstrakcije Y komponente i centriranja oko 0

Inicijalna implementacija je radila sa skalarnim cjelobrojnim podacima, te se cijeli proces odvijao na nivou jedne slike:

```
void extractYComponent(BMPImage *img, YImage *y_out)
{
    // Prije petlje je priprema podataka
    int i;
    for (i = 0; i < total_pixels; i++) {
        int y_temp =
              coeff_r * r[i]
            + coeff_g * g[i]
            + coeff_b * b[i];

        y_temp >>= 8;
        y_temp -= 128;
        y[i] = (int8_t) y_temp;
    }
}

Broj ciklusa: 207 251 788
```

Prvi korak mi je bio vektorizacija koda, tako da funkcija koristi vektorske intrinzike C7x kompajlera. Konkretno, pošto je dužina vektorskog registra C7x DSP-a jednaka 512 bit-a, vektorske operacije su radjene sa 32 cjelobrojna podatka duzine 16 bita. Implementacija funkcije je sada:

```
void extractYComponent(BMPImage *img, YImage *y_out)
{

    for (i = 0; i < num_vectors; i++) {
        uchar32 r_in = vec_r[i];
        uchar32 g_in = vec_g[i];
        uchar32 b_in = vec_b[i];

        short32 r_s = convert_short32(r_in);
        short32 g_s = convert_short32(g_in);
        short32 b_s = convert_short32(b_in);

        short32 y_temp = (r_s * coeff_r) + (g_s * coeff_g) + (b_s * coeff_b);

        y_temp = y_temp >> 8;

        y_temp = y_temp - val_128;

        vec_y[i] = convert_char32(y_temp);
    }
}
Broj ciklusa: 126 802 524
```

Analizom aseblerkih anotacija date funkcije može se vidjeti da kompajler ima problema pri prevođenju instrukcije za bitski pomjeraj. Kompajler nije to prepoznao kao vektorsu instrukciju, te je svaki od 32 elementa vektorskog tipa posebno pomjerio:

```
EXT     .L2     B0,0x30,0x30,BL1  ; [B_L2] Izvlači jedan element iz vektora
(skalarno)
SHRW    .L2     BL1,0x8,BM0       ; [B_L2] Šiftuje taj element (skalarno)
VPUTH   .C2     BM0,0,VB0         ; [B_C]  Vraća element nazad u vektor
...
VPUTH   .C2     BM0,0x1,VB0       ; I tako za indeks 1...
...
VPUTH   .C2     BM0,0x1f,VB0      ; Sve do indeksa 31
```

Rješenje je korištenje intrinzika koji govori kompajleru da to posmatra kao vektorsku operaciju. Operacija bitskog pomjeraja postaje:

```
vY_s = __shift_right(vY_s, (short32)8);

;*  20              VADDH   .L2     VBL2,VBL1,VBL1    ; [B_L2] |43|  ^
;*  21              VADDH   .L2     VBL2,VBL1,VBL1    ; [B_L2] |43|  ^
;*  22              VSHRH   .L2     VBL1,0x8,VBL1     ; [B_L2] |48|  ^
;*  23              VSUBH   .L2     VBL1,VBL0,VBM0    ; [B_L2] |50|  ^

Broj ciklusa: 33 362 101
```

Kao što možemo vidjeti u asemeblerskom kodu, skalarna instrukcija je zamijenjena sa vektorskom instrukcijom VSHRH. Refaktorisanjem funkcije da radi na blokovski način, kao i ostatak pipeline-a nije doveo do promjene u prosječnom broju ciklusa potrebnom za izvršavanje, ali je bio ključan za optimizaciju kasnijih koraka koda. Dodavanjem ključne riječi restrict na bafere:

```
const uint8_t * __restrict pRowR = img->r + (startY * width + startX);
const uint8_t * __restrict pRowG = img->g + (startY * width + startX);
const uint8_t * __restrict pRowB = img->b + (startY * width + startX);
int8_t * __restrict pDst = outputBuffer;

Broj ciklusa: 25 612 105
```

Daljom analizom asemblerskih anotacija vidimo da je još jedan od problema učitavanje podataka, tj. Konkretno .D jedinica. Ovo možemo riješiti konfiguracijom Streaming Engine-a da učitava RGB podatke, čime se smanjuje vrijeme čekanja procesora.

Implementacija funkcije:

```
void extractYComponentBlock4x8x8( int8_t * __restrict outputBuffer)
{   . . .
    #pragma MUST_ITERATE(8, 8, 8)
    for (i = 0; i < 8; i++)
    {
        getNextHalfBlock(&vR_s, &vG_s, &vB_s); // Funkija za SE

        vY_s = (vR_s * vCoeffR) + (vG_s * vCoeffG) + (vB_s * vCoeffB);

        vY_s = __shift_right(vY_s, (short32)8);
        vY_s = vY_s - vOffset;

        vY_out = __convert_char32(vY_s);
        *((char32 *)pDst) = vY_out;
        pDst += 32;
    }
}

 Broj ciklusa: 13 144 020
```

3.2 Optimizacija DCT-a

Algoritam je inicijalno implementiran po jednačini 2.12, gdje su unaprijed definisane matrice DCT-a.

```
static const float DCT_T[8][8] = {…};

static const float DCT_T_TRANSPOSED[8][8] = {…};

static void matrixMul8x8(float A[8][8], float B[8][8], float C[8][8]) {...}

void computeDCT(YImage *y_img, DCTImage *dct_out)
{

    for (y = 0; y <= height - 8; y += 8)
    {
        for (x = 0; x <= width - 8; x += 8)
        {
            for (i = 0; i < 8; i++)
            {
                for (j = 0; j < 8; j++)
                {
                    block[i][j] = (float)srcData[(y + i) * width + (x + j)];
                }
            }
        }
        matrixMul8x8(block, (float (*)[8])DCT_T_TRANSPOSED, temp);
        matrixMul8x8((float (*)[8])DCT_T, temp, result);
     }
}
 Broj ciklusa: 194 106 426
```

Inicijalna implementacija je primjenjivala DCT transformaciju na cijelu sliku. Nakon refaktorisanja koda, tako da se obrada vrši po blokovima, dobije se ubrzanje:

```
static void computeDCTBlock(const float * __restrict src_block, float *
__restrict dct_out)
{
    __attribute__((aligned(64))) float temp[64];
    __attribute__((aligned(64))) float result[64];
    int i;

    // Step A: Temp = Block * T'
    matrixMul8x8(src_block, DCT_T_TRANSPOSED, temp);

    // Step B: Result = T * Temp
    matrixMul8x8(DCT_T, temp, result);

    // Store Result
    #pragma MUST_ITERATE(64, 64, 64)
    for( i = 0; i < 64; i++)
    {
        dct_out[i] = result[i];
    }
}

 Broj ciklusa: 156 158 993
```

Dodavanjem inline-a na funkciju MatMul8x8 i konverzije predefinisanih matrica u linearni niz ne dobija se ubrzanje. Dodavanjem MUST_ITERATE i UNROLL anotacija na petlje matričnog množenja i restrict ključne riječi na pokazivače:

```
static inline void matrixMul8x8(const float * __restrict A, const float
* __restrict B, float * __restrict C)

#pragma MUST_ITERATE(8, 8, 8)
for ( i = 0; i < 8; i++)
{
      #pragma MUST_ITERATE(8, 8, 8)
      #pragma UNROLL(8)
      for ( j = 0; j < 8; j++)
      {
            float sum = 0.0f;

            #pragma MUST_ITERATE(8, 8, 8)
            #pragma UNROLL(8)
            for ( k = 0; k < 8; k++)
            {
                  sum += A[i*8 + k] * B[k*8 + j];
            }
            C[i*8 + j] = sum;
      }
}
 Broj ciklusa: 55 296 651
```

Ako se doda UNROLL i na vanjsku petlju, broj ciklusa skoči na 610 997 278. Potpuni unroll vanjske petlje je generisao veliki blok linearnog koda koij prelazi kapacitet keš memorije, što uzrokuje zastoje radi učitavanja instrukcija iz memorije. Implementacijom MatrixMul kao skalarni proizvod, te promjenom redoslijeda množenja:

```
static inline void matrixMul8x8(const float * __restrict A, const float
* __restrict B, float * __restrict C)
{
      int u, v, k;
      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++)
                  {
                        /* Pristup linearnom nizu: row*8 + col */
                        sum += A[u*8 + k] * B[v*8 + k];
                  }
                  C[u*8 + v] = sum;
            }
      }
}
/* Step A: Temp = Block * T' */
matrixMul8x8(DCT_T, block, temp);
/* Step B: Result = T * Temp */
matrixMul8x8(DCT_T, temp, result);

 Broj ciklusa: 120 240 649
```

Problem je u horizontalnoj redukciji, tj. sabiranju unutar vektora što onemogućava vektorizaciju. Ako promjenim redoslijed petlji i operacija dobijam:

```
for (i = 0; i < 8; i++)
{
      #pragma MUST_ITERATE(8, 8, 8)
      for (k = 0; k < 8; k++)
      {
            float a_val = A[i * 8 + k];
            #pragma MUST_ITERATE(8, 8, 8)
            #pragma UNROLL(8)
            for (j = 0; j < 8; j++)
            {
                  C[i * 8 + j] += a_val * B[k * 8 + j];
            }
      }
}

 Broj ciklusa: 70 145 657
```

Iako ovaj pristup teoretski omogućava bolju vektorizaciju, implementirana verzija rezultirala je degradacijom performansi zbog neefikasnog upravljanja memorijskim resursima. Direktna akumulacija vrednosti u odredišni niz unutar najdublje petlje nametnula je operaciju čitaj-modifikuj-piši. Ovakav pristup je doveo do saturacije jedinica za učitavanje i skladištenje podataka i uvođenja latencije u izvršni cjevovod. Nasuprot tome, standardna implementacija koristi procesorske registre za akumulaciju međurezultata, svodeći interakciju sa memorijom na samo jedan upis po izračunatom elementu.

Refaktorisanjem funkcije na granularnost od 4 bloka umjesto jednog:

```
void computeDCTBlock4x8x8(const int8_t * __restrict src_data, float *
__restrict dct_out)
{
    __attribute__((aligned(64))) float work_block[64];

    int k, i;
    #pragma MUST_ITERATE(4, 4, 4)
    for (k = 0; k < 4; k++)
    {
        const int8_t *current_src = src_data + (k * 64);
        float *current_dst = dct_out + (k * 64);

        #pragma MUST_ITERATE(64, 64, 64)
        for(i = 0; i < 64; i++)
        {
            work_block[i] = (float)current_src[i];
        }

        computeDCTBlock(work_block, current_dst);
    }
}
 Broj ciklusa: 37 409 343
```

Grupisanje obrade na 4 bloka donijelo je ubrzanje primarno kroz poboljšanu lokalnost instrukcija i podataka. Izvršavanje DCT transformacije u uskoj petlji osigurava da programski kod ostaje rezidentan u kešu, eliminišući zastoje uzrokovane ponovnim učitavanjem instrukcija koje se dešavaju kod pojedinačnih poziva. Takođe, ovakva struktura omogućava kompajleru da invariantne podatke (poput adresa DCT tabela) zadrži u registrima kroz sve iteracije, umjesto da ih ponovo učitava za svaki blok.

3.3 Optimizacija kvantizacije DCT koeficijenata

Algoritam je inicijalno radio kvantizaciju na nivou cijele slike, tako što je množio cijelu sliku sa inverznom matricom kvantizacije koja je već predefinisana u kodu. Time je operacija dijeljenja zamijenjena mnogo efikasnijoj operaciji množenja.

```
void quantizeImage(DCTImage *dct_img, QuantizedImage *q_img)
{   // Inicijalizacija parametara prije petlje

    for (y = 0; y <= height - 8; y += 8)
    {
        for (x = 0; x <= width - 8; x += 8)
        {
            // Iterate inside the 8x8 block
            for (i = 0; i < 8; i++)
            {

                float *blockRowSrc = &srcData[(y + i) * width + x];
                int16_t *blockRowDst = &dstData[(y + i) * width + x];

                const float *tblRow = &RECIP_LUMINANCE_QUANT_TBL[i * 8];


                for (j = 0; j < 8; j++)
                {
                    float dctVal = blockRowSrc[j];
                    float recipQ = tblRow[j];

                    float scaled = dctVal * recipQ;

                    blockRowDst[j] = (int16_t)roundf(scaled);
                }
            }
        }
    }
}

Broj ciklusa: 901 158 936
```

Refaktorisanjem kvantizacije da radi blokovsku obradu:

```
void quantizeBlock(float *dct_block, int16_t *quant_block)
{
    for ( i = 0; i < 64; i++)
    {
        float dctVal = dct_block[i];
        float recipQ = RECIP_LUMINANCE_QUANT_TBL[i];

        float scaled = dctVal * recipQ;

        quant_block[i] = (int16_t)roundf(scaled);
    }
}

 Broj ciklusa: 747 994 027
```

Dodavanjem MUST_ITERATE i UNROLL na petlju nije promjenio broj ciklusa. Problem predstavlja roundf instrukcija koja onemogućava kompajleru da primjeni pipeline optimizacije na petlju:

```
;*   SOFTWARE PIPELINE INFORMATION
;*      Loop found in file              :
/home/damjans/Desktop/SDOS/Projekat/jpeg-image-compression/dsp_port/
jpeg_compression/src/quantization.c
;*      Loop source line                : 38
;*      Loop opening brace source line  : 39
;*      Loop closing brace source line  : 49
;*      Disqualified loop: Loop contains a call
;*      Disqualified loop: Loop contains non-pipelinable instructions
```

To sam riješio korištenjem vektorskih tipova i ugrađene funkcije za konverziju iz podataka sa pokretnim zarezom u cjelobrojne podatke:

```
void quantizeBlock(float *dct_block, int16_t *quant_block)
{
    #pragma MUST_ITERATE(4, 4, 4)
    for ( i = 0; i < 4; i++) {
        float16 val = v_dct[i];
        float16 q   = v_quant[i];

        float16 scaled = val * q;

        v_out[i] = __convert_short16(scaled);
    }
}
 Broj ciklusa: 15 810 794
```

Dodavanjem ključne riječi restrict na ulazne i izlazne pokazivače:

```
void quantizeBlock(float * __restrict dct_block, int16_t * __restrict
quant_block) {…}

 Broj ciklusa: 6 264 466
```

Nakon pomenutih optimizacija asemblerske anotacije za petlju su:

```
;*      Loop source line                : 31
;*      Loop opening brace source line  : 31
;*      Loop closing brace source line  : 43
;*      Known Minimum Iteration Count   : 4
;*      Known Maximum Iteration Count   : 4
;*      Known Max Iteration Count Factor : 4
;*      Loop Carried Dependency Bound(^) : 0
;*      Partitioned Resource Bound      : 2 (pre-sched)
;*      Searching for software pipeline schedule at ...
;*         ii = 2  Schedule found with 4 iterations in parallel
(pruning)
;*      Partitioned Resource Bound(*)   : 2 (post-sched)
;*      Resource Partition (may include "post-sched" split/spill moves):
```

Refaktorisanjem funkcije na granularnost od 4 bloka umjesto jednog:

```
void quantizeBlock4x8x8(float * __restrict dct_macro_block,
                        int16_t * __restrict quant_macro_block) {…}

 Broj ciklusa: 2 853 688
```

Povećanje performansi je posledica smanjenja redudantnih memorijskih pristupa i maksimizacije ponovne upotrebe podataka. Dok pojedinačna funkcija zahteva ponovno dobavljanje vektora kvantizacione tabele iz memorije za svaki 8x8 blok, optimizovana verzija primenjuje tehniku promocije konstanti u registre. Vektori tabele se učitavaju u registre procesora samo jednom na početku makro-bloka i zadržavaju se tokom obrade sva četiri pod-bloka.

3.4 Optimizacija cik-cak permutacije

Inicijalna implementacija algortima je radila na nivou slike:

```
void performZigZag(QuantizedImage *q_img, int16_t *zigzag_out)
{   // Inicijalizacija parametara prije petlje
    for (blockY = 0; blockY < height; blockY += 8)
    {
        for (blockX = 0; blockX < width; blockX += 8)
        {
            int16_t *blockOutputPtr = &zigzag_out[blockIndex * 64];

            for (i = 0; i < 64; i++)
            {
                int zzPos = ZIGZAG_ORDER[i];
                int localRow = zzPos / 8;
                int localCol = zzPos % 8;
                int srcIndex = (blockY + localRow) * width + (blockX +
localCol);
                blockOutputPtr[i] = srcData[srcIndex];
            }
            blockIndex++;
        }
    }
}
 Broj ciklusa: 125 275 536
```

Sa refaktorisanjem na blokovsku obradu:

```
void performZigZagBlock(int16_t *quant_block, int16_t *zigzag_block)
{
    int i;
    for (i = 0; i < 64; i++)
    {
        uint8_t src_idx = ZIGZAG_ORDER[i];

        zigzag_block[i] = quant_block[src_idx];
    }
}
 Broj ciklusa: 88 909 908
```

Prvobitna implementacija vršila je kompleksnu re-kalkulaciju adresa unutar najdublje petlje, zahtevajući operacije celobrojnog dijeljenja i množenja sa širinom slike za svaki koeficijent kako bi se mapirale 1D ZigZag koordinate u 2D prostor slike. Nova implementacija eliminiše ovaj trošak operišući nad prethodno izdvojenim linearnim blokom od 64 elementa. Ovakav pristup svodi adresiranje na jednostavan indirektni pristup nizu, smanjujući broj instrukcija po pikselu. Dodatno, rad sa malim, kontigualnim blokom osigurava veću prostornu lokalnost podataka (*Spatial Locality*), garantujući da se svi koeficijenti nalaze u kešu, čime se eliminiše kašnjenje uzrokovano pristupom glavnoj memoriji. Nakon dodavanja restrict na ulazne i izlazne pokazivače:

```
void performZigZagBlock(const int16_t * __restrict quant_block, int16_t
* __restrict zigzag_block) {…}


 Broj ciklusa: 23 775 488
```

Korištenjem vektorskih intrinzika za permutaciju:

```
static inline uchar64 __vperm_wrapper(uchar64 mask, uchar64 src1,
uchar64 src2)
{
    uchar64 p1 = __vperm_vvv(mask, src1);
    uchar64 p2 = __vperm_vvv(mask, src2);

    __vpred pred = __cmp_gt_pred(convert_char64(mask), (char64)63);

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

Budući da jedan blok 8x8 blok 16 bitnih koeficijenata prevazilazi kapacitet 512-bitnog vektorskog registra, kreirana je funkcija koja simulira permutaciju na dva izvorna vektora. Upotrebom permutacionog intrinzika i pre-kalkulisanih maski, procesor vrši paralelno ispremeštanje bajtova unutar vektorskih registara. Primjenom permutacije nad istom maskom dobijaju se 2 privremena vektora, jedan koji sadrži validne podatke za indekse 0 – 63 i drugi koji sadrži validne podatke za indekse 64 – 127. Predikatski vektor vrši korekciju tako što poredi vrijednosti u masci sa graničnom vrijednošću 63. Ovaj predikat služi kao kontrolni signal za __select instrukciju koja spaja validne bajtove iz dva privremena vektora u konačan rezultat, eliminišući potrebu za uslovnim skokovima. Opisani postupak se izvršava iterativno za donji i gornju polovinu izlaznog bloka, nakon čega se formira izlazni vektor.

```
void performZigZagBlock(const int16_t * __restrict quant_block, int16_t
* __restrict zigzag_block)
{
    const short32 *input_vec_ptr = (const short32 *)quant_block;
    short32 *output_vec_ptr      = (short32 *)zigzag_block;

    short32 v_src0_s = input_vec_ptr[0];
    short32 v_src1_s = input_vec_ptr[1];

    uchar64 v_src0_u = as_uchar64(v_src0_s);
    uchar64 v_src1_u = as_uchar64(v_src1_s);

    uchar64 v_res0_u = __vperm_wrapper(perm_mask_lo, v_src0_u,
v_src1_u);
    uchar64 v_res1_u = __vperm_wrapper(perm_mask_hi, v_src0_u,
v_src1_u);

    output_vec_ptr[0] = as_short32(v_res0_u);
    output_vec_ptr[1] = as_short32(v_res1_u);
}

 Broj ciklusa: 6 320 703
```

Maske za prvu i drugu polovinu izlaznih elemenata se generišu u fazi inicijalizacije, Refaktorisanjem funkcije na granularnost od 4 bloka umjesto jednog dobijamo:

```
void performZigZagBlock4x8x8(const int16_t * __restrict src_macro,
int16_t * __restrict dst_macro)
{
    // Inicijalizacija parametara prije petlje
    #pragma MUST_ITERATE(4, 4, 4)
    #pragma UNROLL(4)
    for (k = 0; k < 4; k++)
    {
        // Procesiranje jednog bloka ovdje
    }
}
#endif

 Broj ciklusa: 2 421 407
```

Dodatno ubrzanje pri prelasku na granularnost od 4 bloka rezultat je efikasnijeg softverskog pajplajninga i boljeg iskorišćenja instrukcijskog paralelizma. Kod obrade pojedinačnog bloka, procesor je prinuđen na čekanje usled zavisnosti podataka. Procesiranjem 4 bloka unutar jedne funkcije, kompajler je u mogućnosti da ispreplete instrukcije. Procesiranjem 4 bloka unutar jedne funkcije, kompajler je u mogućnosti da ispreplete instrukcije.

3.5 Optimizacija RLE kodovanja

Inicijalna implementacija je radila na nivou cijele slike:

```
int32_t performRLE(int16_t *zigzag_data, int32_t width, int32_t height,
RLESymbol *rle_out, int32_t max_capacity) {…}

 Broj ciklusa: 895 213 739
```

Izmjenom na pristup obrade blok po blok:

```
int32_t performRLEBlock(int16_t *block, RLESymbol *rle_out, int32_t
max_capacity, int16_t *last_dc_ptr) {…}

 Broj ciklusa: 183 318 939
```

Prelazak sa monolitne implementacije na blokovski pristup rezultirao je značajnim ubrzanjem algoritma. Inicijalna implementacija je patila od visokog pritiska na registre usled velikog broja varijabli unutar glavne petlje. Takođe, repetetivno izračunavanje baznih adresnih blokova unutar petlje je stvaralo nepotreban aritmetički teret. Izolacijom logike na nivo 8x8 bloka je pojednostavilo analizu protoka podataka za kompajler. Ovo je omogućilo promociju svih privremenih varijabli u registre, eliminišući Load/Store operacije ka steku, pojednostavljenje adresiranja i poboljšanu lokalnost podataka.

Dodatna optimizacija je postignuta refaktorisanjem funkcija na granularnost od 4 bloka:

```
int32_t performRLEBlock4x8x8(const int16_t * __restrict
macro_zigzag_buffer, RLESymbol * __restrict rle_out,
                     int32_t max_capacity,
                     int16_t *last_dc_ptr){…}

 Broj ciklusa: 154 305 108
```

Prelazak sa granularnosti pojedinačnog bloka na makro-blok (4x8x8) rezultirao je značajnom redukcijom broja ciklusa, primarno usled amortizacije režijskih troškova poziva funkcije, tj. eliminacijom redudantnih prologa i epiloga i poboljšanom lokalnosti instrukcija.

Inicijalna funkcija za proračun amplitudskog koda je u sebi sadržavala uslovno grananje

```
static inline uint16_t getAmplitudeCode(int16_t val) {
    if (val > 0) {
        return (uint16_t)val;
      } else {
        return (uint16_t)(val - 1);
    }
}
```

Zamjena uslovnog grananja bitovskom aritmetikom:

```
static inline uint16_t getAmplitudeCode(int16_t val)
{
    int16_t mask = val >> 15;
    return (uint16_t)(val + mask);
}
```

Ova optimizacija nije rezultirala značajnom redukcijom broja ciklusa, što ukazuje da je kompajler već primjenio konverziju uslovnog grananja i preveo ga u linearni niz instrukcija bez kontrolnih hazarda. Značajno ubrzanje od 30 miliona ciklusa postignuto je optimizacijom funkcije za dohvatanje bitske dužine zamjenom iterativnog softverskog algoritma namjenskom hardverskom instrukcijom. Originalni kod koristi while petlju za izračunavanje logaritma osnove 2 (bitske dužine), što je rezultiralo varijabilnom latencijom izvršavanja i čestim hazardima grananja.

```
static inline uint8_t getBitLength(int16_t val) {
    if (val == 0) return 0;

    int16_t absVal = (val < 0) ? -val : val;

    uint8_t bits = 0;
    while (absVal > 0) {
        bits++;
        absVal >>= 1;
    }
    return bits;
}
```

Nova implementacija koristi C7x intrinzike za računanje apsolutne vrijednosti i pozicije najvećeg bita:

```
static inline uint8_t getBitLength(int16_t val)
{
    if (val == 0) return 0;

    int32_t v32 = (int32_t)__abs(val); // Convert to 32-bit and take
absolute value

    // Use the norm intrinsic to find the position of the highest set
bit
    return (uint8_t)(32 - __norm(v32) - 1);
}

 Broj ciklusa: 130 165 519
```

Implementacijom dvoprolaznog (2-pass) algoritma, postignuto je ubrzanje RLE kodiranja, zasnovano na principu razdvajanja kontrolnog toka od obrade podataka.
Prva faza koristi linearnu pretragu za detekciju ne-nula koeficijenata. Zahvaljujući odsustvu kompoksne logike, omogućeno je kompajleru da optimizuje ekstrakciju indeksa aktivnih koeficijanata:

```
nz_count = 0;
for (k = 1; k < 64; k++) {
      if (current_block_ptr[k] != 0) {
            nz_indices[nz_count++] = k;
      }
}
```

Druga faza iterira isključivo kroz prethodno prikupljene indekse, čime se eliminiše prazan hod petlje za nulte koeficijente. Umjesto 63 iteracije, sa uslovnim skokovima te svaki blok, petlja se sad izvršava N puta (gdje je N broj ne-nula elemenata, tipično manje od za JPEG kompresiju):

```
for (i = 0; i < nz_count; i++)
{
      int curr_k = nz_indices[i];
      int16_t val = current_block_ptr[curr_k];

      int zero_run = curr_k - last_k - 1;


      if (zero_run >= 16) {
            int num_zrl = zero_run >> 4;
            zero_run    = zero_run & 0xF;
            while (num_zrl > 0)
            {
                  if (total_symbols >= max_capacity)
                        return -1;
                  RLESymbol *z = &rle_out[total_symbols++];
                  z->symbol   = 0xF0;
                  z->code     = 0;
                  z->codeBits = 0;
                  num_zrl--;
            }
      }
      if (total_symbols >= max_capacity)
            return -1;

      uint8_t size = getBitLength(val);
      RLESymbol *ac = &rle_out[total_symbols++];
      ac->symbol   = (uint8_t)((zero_run << 4) | size);
      ac->code     = getAmplitudeCode(val);
      ac->codeBits = size;
      last_k = curr_k;
}
 Broj ciklusa: 56 459 506
```

Dodatna poboljšanja se mogu dobiti korištenjem C7x intrinzika. Prethodna implementacija oslanjala se na privremeni bafer na steku za čuvanje indeksa ne-nula koeficijenata, što je generisalo saobraćaj pri upisivanju i čitanju iz memorije. Kod se može optimizovati koristešnjem vektoriskih intrinzika za direktnu konverziju predikata u 64-bitne skalarne bit-maske, te korištenjem ctz (eng. Count Trailing Zeroes) instrukcije za lociranje sljedećeg koeficijenta.

```
// Load the block as two vectors of 32 short elements
short32 v_lo = *((short32 *)&current_block_ptr[0]);
short32 v_hi = *((short32 *)&current_block_ptr[32]);

// Generate predicates indicating zero elements
__vpred pred_lo_zeros = __cmp_eq_pred(v_lo, (short32)0);
__vpred pred_hi_zeros = __cmp_eq_pred(v_hi, (short32)0);

// Convert predicates to scalar bitmasks
// Each bit corresponds to one byte in the vector
uint64_t raw_lo = (uint64_t)__create_scalar(pred_lo_zeros);
uint64_t raw_hi = (uint64_t)__create_scalar(pred_hi_zeros);

// Reduce byte-level mask to short-level mask
// A short is zero only if both of its bytes are zero
uint64_t mask_const = 0x5555555555555555ULL;
uint64_t is_zero_lo = raw_lo & (raw_lo >> 1) & mask_const;
uint64_t is_zero_hi = raw_hi & (raw_hi >> 1) & mask_const;

// Invert masks to identify non-zero elements
uint64_t nz_mask_lo = (~is_zero_lo) & mask_const;
uint64_t nz_mask_hi = (~is_zero_hi) & mask_const;

// Exclude the DC coefficient which was already processed
nz_mask_lo &= ~1ULL;
int last_k = 0;

while (nz_mask_lo != 0)
{
      // Find next non-zero coefficient
      int bit_idx = ctz_64(nz_mask_lo);
      nz_mask_lo &= (nz_mask_lo – 1);

      .
      .
      .
      // Emit AC symbol
      uint8_t size = getBitLength(val);
      RLESymbol *ac = &rle_out[total_symbols++];
      ac->symbol = (uint8_t)((zero_run << 4) | size);
      ac->code = getAmplitudeCode(val);
      ac->codeBits = size;
      last_k = curr_k;
}

 Broj ciklusa: 41 993 197
```

3.6 Optimizacija Hafmanovog kodovanja

Inicijalna optmizacija algoritma nije došla iz promjene samog načina izvršavanja algoritma, već iz promjene granularnosti blokova obrade prethodnih koraka. Hafman nije promjenjen da radi blokovsku obradu već u oba slučaja radi na nivou cijele slike. Prije promjene granularnosti, kad je svaki korak kompresije radio na nivou sliku, potreban broj ciklusa je bio 1 649 675 685. Nakon izmjene na blokovsku obradu podataka broj potrebnih ciklusa za Hafmanov algoritam je 223 330 099. Ovo ubrzanje se može objasniti pomoću bolje memorijske lokalnosti, jer je prelazak na blokovsku obradu, uz eliminaciju suvišnih memorijskih transakcija i struktura za čuvanje međureprezentacija smanjio pritisak na memorijsku magistralu. Ovo je omogućilo da se veći procenat ulaznih RLE podataka zadrži u keš memoriji, što je smanjilo kašnjenje uneseno čekanju na pristup podacima.

Značajno poboljšanje performanse je postignuto promjenom sistema za bitski upis u bafer. Umesto učestalog upisa u memoriju nakon svakih 8 prikupljenih bita, uvedena je strategija odloženog upisa korišćenjem 64-bitnog akumulatorskog registra. Ovim se frekvencija skupih operacija pristupa memoriji i provere „byte stuffing-a“ znazno smanjuje:

```
// Original
typedef struct { ... uint32_t accumulator; ... } BitWriter;
// Novi
typedef struct { ... uint64_t accumulator; ... } BitWriter;
```

Originalna implementacija funkcije za upis u akumulator je sadržala while petlju za pražnjenje bajtova, stvarajući strukturu petlja-u-petlji koja je onemogućavala kompajleru da efikasno primjeni optimizacije na glavnu petlju kodera. Prelaskom na fiksni prag pražnjenja (32 bita), while petlja je zamenjena jednim uslovnim grananje.

```
static inline void putBits(BitWriter* bw, uint16_t data, uint8_t
numBits)
{
    data &= (0xFFFF >> (16 - numBits));

    bw->accumulator |= ((uint64_t)data << (64 - bw->bitCount -
numBits));
    bw->bitCount += numBits;

    if (bw->bitCount >= 32)
    {
        flushBlock(bw);
    }
}
```

Dodatno je logika za „byte stuffing” odmotana što omogućava linearno izvršavanje koda bez skokova unazad što smanjuje broj promašaja grananja.

```
static inline void flushBlock(BitWriter* bw)
{
    // Extract the upper 32 bits
    uint32_t chunk = (uint32_t)(bw->accumulator >> 32);

    // Shift accumulator and update bit count
    bw->accumulator <<= 32;
    bw->bitCount -= 32;

    // Pointer to the current write position
    uint8_t * __restrict p = &bw->buffer[bw->size];

    // Split the chunk into bytes
    uint8_t b3 = (chunk >> 24) & 0xFF;
    uint8_t b2 = (chunk >> 16) & 0xFF;
    uint8_t b1 = (chunk >> 8)  & 0xFF;
    uint8_t b0 =  chunk        & 0xFF;

    // Write bytes and insert stuffing after 0xFF
    *p++ = b3; if (b3 == 0xFF) *p++ = 0x00;
    *p++ = b2; if (b2 == 0xFF) *p++ = 0x00;
    *p++ = b1; if (b1 == 0xFF) *p++ = 0x00;
    *p++ = b0; if (b0 == 0xFF) *p++ = 0x00;

    // Update total output size
    bw->size = (int32_t)(p - bw->buffer);
}
```

Navedene optimizacije su smanjile broj potrebnih ciklusa za izvršavanje na 83 306 184. Dodavanjem restrict ključne riječi na ulazni i izlazni bafer, te uklanjanje suvišnih međubafera je dovelo do smanjenja ciklusa na 71 708 555.

3.7 Moguća poboljšanja

Trenutna implementacija DCT algoritma koristi definiciju 2D DCT-a putem matričnog množenja, što rezultira algoritamskom visokom algoritamskom složenošću. Moguće ubrzanje bi se postiglo implementacijom Separabilnog Fast DCT algoritma (npr. Arai-Agui-Nakajima). Ovaj pristup razlaže 2D transformaciju na dva 1D prolaza i zamenjuje matrično množenje "leptir" dijagramom toka podataka, smanjujući broj potrebnih množenja sa 512 na svega ~100 po bloku, uz minimalni gubitak preciznosti.

Još jedan moguća optimizacija za DCT jeste prelazak na aritmetiku sa fiksnim zarezom ili sa cijelobrojnim podacima. U slučaju zadržavanja matričnog množenja veliko poboljšanje možemo dobiti

upotrebom C7x MMA (Matrix Multiply Accelerator) jedinice koja je specijalizovana za množenje matrica.

Trenutna implementacija kvantizacije koristi direktnu vektorsku konverziju iz podataka sa pokretnim zarezom u cjelobrojne podatke što povećava grešku kvantizacije. Implementiranjem drugog načina zaokruživanja bi se mogla smanjiti ova greška. Takođe, prelaskom na format sa nepokretnim zarezom bi mogao dovesti do manje greske ujedno sa manjim vremenom izvršavanja.

Spajanjem kvantizacije i cik cak skeniranja bi podrazumijevalo da se kvantizovani koeficijenti upisuju direktno na njihove finalne cik cak pozicije u izlaznom baferu, što eliminiše prolaz čitanja/pisanja kroz memoriju i ubrzava vrijeme izvršavanja algoritma.

Trenutna implementacija cik cak algortima tretira sve kvantizovane koeficijente kao 16-bitne vrednosti, što zahteva dva vektorska registra za smeštanje jednog 8x8 bloka. S obzirom na to da distribucija AC koeficijenata u JPEG kompresiji naginje ka malim vrednostima (često nula), buduća optimizacija bi uključivala dinamičko pakovanje podataka. Proverom opsega vrednosti unutar bloka, podaci koji staju u 8-bitni opseg mogli bi se komprimovati u jedan vektor. Ovo bi omogućilo izvršavanje ZigZag permutacije u jednom taktu korišćenjem proste permutacione instrukcije, eliminišući potrebu za skupom vektorskom logikom i smanjujući saobraćaj ka memoriji.

Trenutna implementacija RLE kodovanja koristi niz struktura za skladištenje RLE simbola. Iako logičan za ljude, ovaj format dovodi do neefikasnog pristupa memoriji jer onemogućava kontiualne upise i čitanja. Buduća optimizacija podrazumevala bi prelazak na Structure of Arrays (SoA) paradigmu, gde se simboli, kodovi i dužine čuvaju u tri zasebna linearna niza. Ova promena bi eliminisala potrebu za strukturnim poravnavanjem (padding) i smanjila "cache thrashing".

4. Analiza rezultata i evaluacija kvaliteta kompresije

Evaluacija performansi algoritama za kompresiju slike sa gubicima (eng. lossy compression) predstavlja složen inženjerski problem koji zahteva uspostavljanje ravnoteže između dva suprotstavljena cilja: minimizacije veličine podataka i maksimizacije vernosti rekonstruisanog signala. Kako JPEG standard uvodi ireverzibilne promene u frekvencijskom domenu, neophodno je kvantifikovati nastalu degradaciju korišćenjem objektivnih matematičkih metrika. U ovoj analizi su korištene sljedeće kategorije metrika:

1. Metrike vjernosti signala: MSE i PSNR koje mjere apsolutnu razliku u intenzitetu piksela.
2. Metrike perceptivnog kvaliteta: SSIM, koja modeluje način na koji ljudski vizuelni sistem percipira strukturu promjene.
3. Metrike efikasnosti kompresije: CR i BPP, koje mjere stepen redukcije podataka.
4. Vizuelna dijagnostika greške: Mapa razlika (Difference Map) koja omogućava prostornu lokalizaciju degradacije i identifikaciju specifičnih artefakata koji mogu biti maskirani u usrednjenim statistilkim vrijednostima.

Srednja kvadratna greška

Srednja greška (Mean squared error – MSE) je fundementalna metrika u obradi signala koja kvantifikuje prosječnu snagu greške šuma unijetog kompresijom. Za originalnu sliku *I* i rekonstruisanu sliku *K* dimenzija *M*x*N,* MSE se definiše kao:

$$MSE = \frac{1}{M \cdot N} \sum_{i=0}^{M-1} \sum_{j=0}^{N-1} [I(i,j) - K(i,j)]^2 \tag{4.1}$$

gdje *I(i,j)* i *K(i,j)* predstavljaju vrijednosti intenziteta piksela na koordinatama *(i,j)*. Vrijednost 0 ukazuje na savršenu rekonstrukciju.

Odnos vršnog signala i šuma

Odnos vršnog signala i šuma (Peak Signal-To-Noise Ratio – PSNR) predstavlja logaritamsku reprezentaciju između maksimalne moguće snage signala i snage koruptivnog šuma (predstavljenog kroz MSE). Izražava se u decibelima:

$$PSNR = 10 \log_{10}\left(\frac{MAX_I^2}{MSE}\right) = 20 \log_{10}\left(\frac{MAX_I}{\sqrt{MSE}}\right) \tag{4.2}$$

Gdje je *MAX* maksimalna moguća vrijednost piksela (za 8-bitnu sliku to je 255). Vrijednosti iznad 40

dB se smatraju rekonstrukcijom visokog kvaliteta, u rasponu od 30 do 40 db prihvatljivog kvaliteta, a vrijednosti manje od 20 dB ukazuju na značajnu degradaciju slike.

Indeks strukturne sličnosti

Za razliku od MSE i PSNR koji tretiraju greške kao apsolutne razlike, indeks strukturne sličnosti (Structural Similarity Index Measure – SSIM) se zasniva na pretpostavci da je ljudski vizuelni sistem visoko adaptiran za izvlačenje strukturnih informacija iz vizuelne scene. SSIM mjeri sličnosti između dve slike kroz tri komponente: osvjetljenje (l), kontrast (c) i strukturu (s):

$$SSIM(x,y)=[l(x,y)^{\alpha}]\cdot[c(x,y)]^{\beta}\cdot[s(x,y)]^{\gamma} \tag{4.3}$$

Odnos kompresije

Odnos kompresije (Compression Ratio – CR) se definiše kao odnos veličine originalnog dokumenta i veličine kompresovanog dokumenta:

$$CR=\frac{Originalna\,veličina}{Kompresovana\,veličina} \tag{4.4}$$

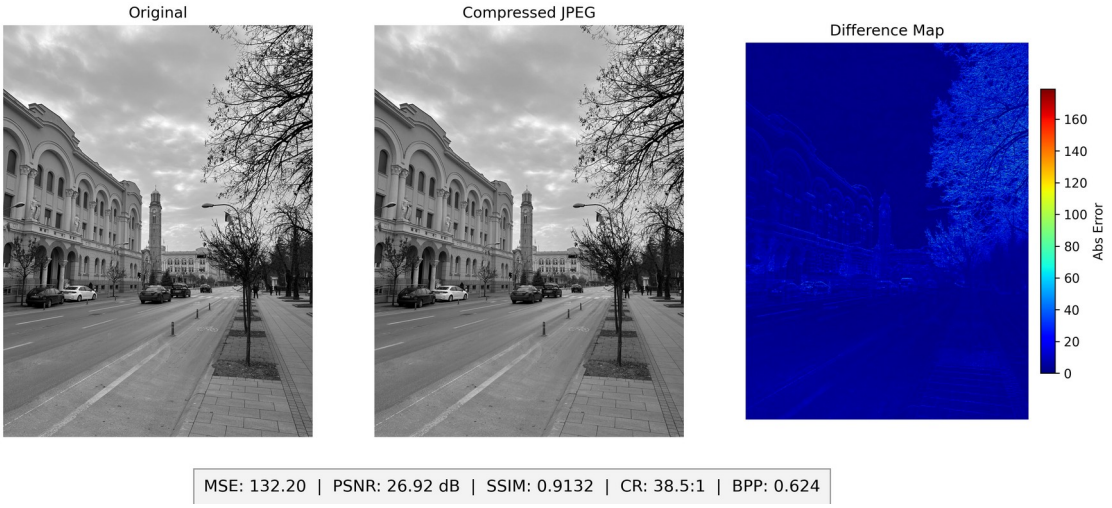
Broj bita po pikselu

Predstavlja normalizovanu mjeru koja omogućava poređenje efikasnosti nezavisno od rezolucije slike:

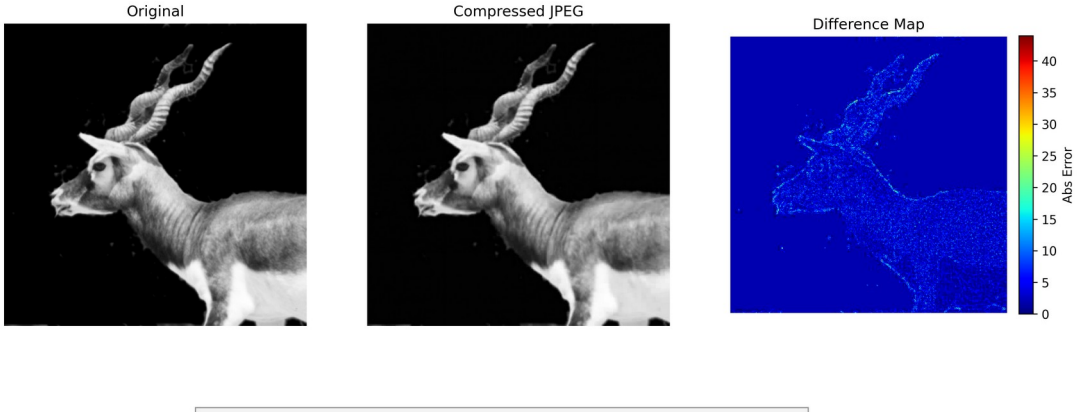
$$BPP=\frac{Kompresovana\,veličina}{Ukupan\,broj\,piksela} \tag{4.5}$$

Rezultati eksperimentalne evaluacije

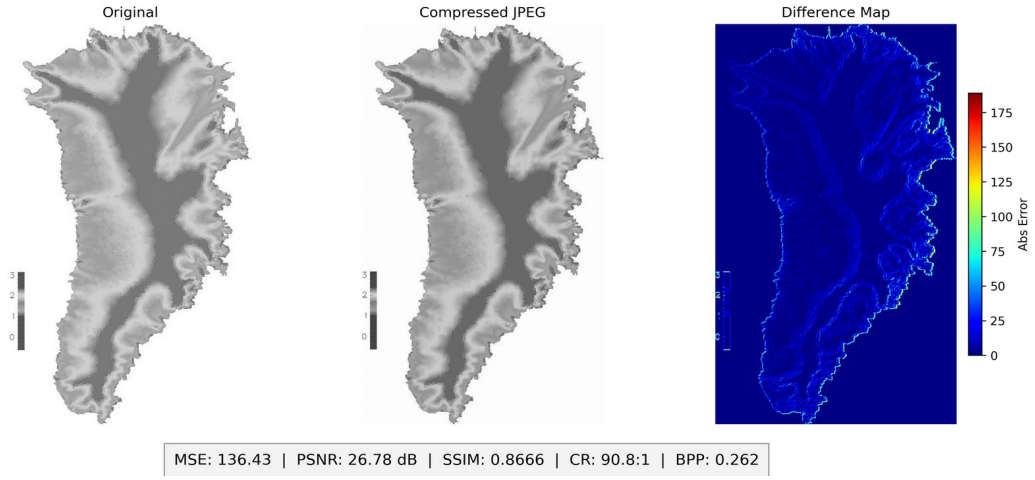
Testiranje je izvršeno na skupu referentnih slika različitih karakteristika (različiti nivoi detalja, kontrasta i tekstura), kako bi se ispitala robusnost algoritma i uticaj kvantizacije na vizuelni integritet signala. Dobijeni rezultati su analizirani kroz prethodno definisane metrike. U nastavku su prezentovani vizuelni rezultati kompresije za odabrane testne uzorke, nakon čega slijedi zbirna tabela koja objedinjuje izmjerene vrijednosti objektivnih metrika kvaliteta i parametara efikasnosti.



Slika 4.1 - *Komparativni prikaz slika i mape razlika za testni uzorak ,,grad.bmp“*



Slika 4.2 - *Komparativni prikaz slika i mape razlika za testni uzorak ,,blackbuck.bmp“*



Slika 4.2 - *Komparativni prikaz slika i mape razlika za testni uzorak ,,greenland.bmp“*

MSE: 26.85 | PSNR: 33.84 dB | SSIM: 0.8997 | CR: 50.9:1 | BPP: 0.472

Slika 4.4 – *Komparitivni prikaz slika i mape razlika za testni uzorak ,,lena.bmp“*

MSE: 149.12 | PSNR: 26.40 dB | SSIM: 0.7518 | CR: 38.7:1 | BPP: 0.621

Slika 4.5 – *Komparitivni prikaz slika i mape razlika za testni uzorak ,,offset_sample.bmp“*

MSE: 206.89 | PSNR: 24.97 dB | SSIM: 0.6878 | CR: 43.2:1 | BPP: 0.556

Slika 4.6 – *Komparativni prikaz slika i mape razlika za testni uzorak ,,sample_1920x1280.bmp“*

Tabela 4.1 – *Zbirni prikaz performansi kompresije za različite testne uzorke*

Naziv slike	MSE	PSNR [dB]	SSIM	CR	BPP
grad.bmp	132.20	26.92	0.9132	38.5 : 1	0.624
blackbuck.bmp	10.06	38.10	0.7401	79.5 : 1	0.302
greenland.bmp	136.43	26.78	0.8666	90.8 : 1	0.262
lena.bmp	26.85	33.84	0.8997	50.9 : 1	0.472
offset_sample.bmp	149.12	26.40	0.7518	38.7 : 1	0.621
sample_1920x1280.bmp	206.89	24.97	0.6878	43.2 : 1	0.556

Slika *blackbuck* postigla je najveći stepen kompresije (~80:1) i najveći PSNR (38.10 dB), što matematički sugeriše najmanju grešku. Međutim, njen SSIM je najniži (0.7401). Ovo je uzrokovano jer ova slika sadrži homogenu, tamnu pozadinu. U takvim regijama JPEG kompresija efikasno eliminiše AC koeficijente što dovodi do velike uštede prostora. Takođe, greške u tamnim pikselima su numerički male, pa su MSE i PSNR nizak. Međutim ljudsko oko je izuzezno osjetljivo na promjene glatkih površina, pa čak i mala greška ovdje stvara vidljive artefakte i blokvske granica.

Supotno tome, silka *grad* ima visok MSE i nizak PSNR, ali vizuelno i strukturno djeluje najkvalitetnije (Najveći SSIM). Ova slika je bogata visokim frekvencijama (ivice zgrada, prozori, granje, teksture), te JPEG kvantizacija ovdje pravi velike numeričke greške jer odbacuje ili grubo zaokružuje te frekvencije. Iako su pikseli numerički promjenjeni, glavna struktura je očuvana, SSIM (a i ljudsko oko) teže primjećuje ove greške, pa daje visoku ocjenu.

Navedeno potvrđuje da PSNR nije uvijek pouzdan indikator subjektivnog kvaliteta slika, za slike sa malo detalja kompresija mora biti pažljivija jer su artefakti lako uočljivi, dok se je za slike sa mnogo detalja moguće primjeniti agresivnu kvantizaciju. Slika *lena* predstavlja balansiran slučaj, gdje su sve metrike usaglašene.

Poseban segment analize posvećen je poređenju rezultata za isti vizuelni sadržaj prikazan u dvije različite rezolucije: *offset_sample* (1280x853) i *sample_1920x1280* (1920x1280), gdje je na prvu sliku primjenjeno i dopunjavanje do 856 piksela. Manja slika predsavlja down-sampled verziju originalne

scene. Proces smanjivanja rezolucije matematički djeluje kao niskopropusni filtar, gdje se visokofrekventni šum i najsitiniji detalji teksture gube, slika postaje „glađa“ sa manje nagli prelaza, što omogućava DCT transformaciji da efikasnije pakuje energiju u mali broj niskofrekventnih koeficijenata što rezultuje manjom greškom i boljim očuvanjem strukture. Nasuprot tome, veća slika sadrži visoke frekvencije koje kvantizaciona tabela agresivno odbacuje, unoseći veću grešku. Dopunjavanje slike replikacijom ivica je takođe smanjilo postotak greške, jer sada u vertikalnom pravcu nema promjene signala u tim redovima, što rezultari koeficijentima vrlo niske frekvencije koji se efikasno kompresuju.

5. Zaključak

Cilj ovog projekta je bio implementacija i evaluacija performansi JPEG algoritma za kompresiju slike na TI C7000 DSP arhitekturi, sa fokusom na analizu odnosa između stepena redukcije podataka i kvaliteta rekonstruisanog signala. JPEG standard se, kroz primjenu diskretne kosinusne transformacije (DCT), potvrdio kao dominantno rješenje za kompresiju fotografija sa kontinuiranim tonovima, omogućavajući efikasnu dekorelaciju piksela i koncentraciju energije u niskim frekvencijama. Eksperimentalna verifikacija je pokazala da implementirani enkoder uspješno balansira između memorijskih zahtjeva i vizuelnog integriteta, postižući prosječne odnose kompresije preko 40:1, čime se originalni fajlovi svode na manje od 0.6 bita po pikselu, što je od ključnog značaja za embedded sisteme sa ograničenim propusnim opsegom.

Glavna prednost analiziranog rješenja leži u njegovoj spektralnoj efikasnosti. Algoritam iskorištava nesavršenost ljudskog vizuelnog sistema, koji je manje osjetljiv na promjene u visokim frekvencijama, te agresivnom kvantizacijom tih komponenti drastično smanjuje veličinu fajla bez gubitka suštinskog sadržaja slike. U praktičnom dijelu rada, ovo je posebno potvrđeno implementacijom tehnike replikacije ivica prilikom nadopunjavanja blokova. Za razliku od standardnog popunjavanja nulama, ovaj pristup je spriječio stvaranje vještačkih visokih frekvencija na rubovima slike, čime je eliminisana pojava artefakata i dodatno poboljšana efikasnost kompresije u graničnim blokovima.

S druge strane, analiza je istakla i fundamentalna ograničenja blokovske obrade signala. Podjela slike na fiksne 8x8 blokove neminovno dovodi do diskontinuiteta na njihovim granicama, koji postaju vidljivi artefakti pri nižim bitskim protocima. Takođe, ireverzibilna priroda kvantizacije uzrokuje nepovratan gubitak informacija, što se manifestuje kroz Gibbs-ov fenomen (zamućenje i oscilacije) oko oštrih ivica. Ovo čini JPEG manje pogodnim za slike sa oštrim kontrastima, tekstom ili tehničkim crtežima, gdje preciznost ivica ima prioritet nad glatkim prelazima boja.