Elektrotehnički fakultet
Univerzitet u Banjoj Luci

IZVJEŠTAJ PROJEKTNOG ZADATKA

iz predmeta

SISTEMI ZA DIGITALNU OBRADU SIGNALA

Student:
Tripić Nemanja, 11124/18

Mentori:
prof. dr Mladen Knežić
prof. dr Mitar Simić
dipl. inž. Damjan Prerad
ma Vedran Jovanović

Januar, Februar 2024. godine

1. Opis projektnog zadatka

U sklopu projektnog zadatka potrebno je realizovati sistem za dodavanje muzičkih gitarskih efekata u audio signal. Za realizaciju ovog sistema potrebno je koristiti razvojno okruženje *ADSP*-21489 kroz programski paket *CrossCore Embedded Studio* pomoću kojeg se kreira projekat, zatim piše i kompajlira kod, a zatim i spušta na razvojnu ploču. Pisanje koda za ovo razvojno okruženje podrazumijeva korištenje programskog jezika *C*. Osim toga potrebno je sve efekte koji se izaberu, kao i same audio signale, realizovati u programskom jeziku *Python,* a to u svrhu poređenja rezultata dobijenih na dva načina realizacije radi profilisanja koda, te mjerenja performansi.

2. Izrada projektnog zadatka

Pri izradi projektnog zadatka izabrani su sljedeći efekti: *delay, distortion, wah-wah, phaser, reverb.* U narednim pasusima biće ukratko teorijski opisan svaki od efekata, uz to će biti dati detalji realizacije, objašnjenja korištenih parametara, osim toga biće priložen kod realizovan u *CCES* za pokretanje na ploči *ADSP-21489.* Više detalja o svakom efektu, realizacija u *python*-u itd., može se naći na repozitorijumu na kojem je i ovaj izvještaj.

2.1 *Delay* (kašnjenje) efekat

Ovaj audio efekat spada u grupu efekata zasnovanih na kašnjenju, odatle i njegov naziv. Zapravo to je jedan od najjednostavnijih i u suštini osnovni efekat iz ove grupe, većina drugih efekata je zasnovana na njemu. Za realizaciju može se koristiti FIR filtar i ova realizacija biće korištena u slučaju ovog projektnog zadatka. Za realizaciju efekta koristi se jednačina diferencija na osnovu koje se onda programski može jednostavno realizovati efekat:

$$y(n) = x(n) + gx(n - M) \qquad (2.1.1)$$

Iz jednačine diferencija može se vidjeti da postoje dva parametra pri realizaciji, a to su M i g. Parametar M označava broj odmjeraka za koji je signal zakašnjen iz relacije $M = t * Fs$, gdje je t vrijeme kašnjenja signala u sekundama, a Fs frekvencija odmjeravanja, dok je g faktor pojačanja zakašnjelog signala, tj. često se kaže odnos amplituda reflektovanog i direktnog signala. Obično uzima vrijednosti između 0 i 1.

Sada slijedi prikaz realizacije efekta kašnjenja u *CCES.* Na slici 2.1.1 prikazan je kod u C programskom jeziku kojim je na ploči *ADSP-21489* realizovan efekat. Na početku se alocira memorija za

pomoćni bafer *x_delay* u koji ćemo smjestiti zakašnjele odmjerke ulaznog signala, i to se radi jednom *for* petljom pri čemu u pomoćni signal smještamo odmjerke iz originalnog niza počevši od odmjerka označenog sa parametrom M i to radimo *if* provjerom unutar petlje. Nakon toga u drugoj *for* petlji, smještaju se odmjerci u izlazni niz, kombinujući originalne odmjerke i zakašnjene uz skaliranje zakašnjelih odmjeraka sa parametrom *g*. Ukoliko se, koristeći makroe iz *cycle_count.h* zaglavlja, izmjeri broj potrebnih ciklusa za izvršavanje efekta, za slučaj osnovne implementacije efekta kašnjenja dobiće se da je utrošeno 5 945 835 ciklusa. Detaljnijom analizom rada efekta i koda koji je napisan, može se doći do prve jednostavne optimizacije. Pošto je poznato da je grananje unutar *for* petlji nešto što narušava performanse *DSP* koda, potrebno ga je, ako je ikako moguće, izbjeći ili uprostiti. Upravo to se može uraditi sa prvom *for* petljom unutar efekta kašnjenja. Grananje se može potpuno izbaciti, a petlja može da broj od M umjesto od 0.

```
void delay(float *x, float *y, int M, float g)
{
 float *x_delay = (float *)heap_calloc(index, LEN, sizeof(float));
 int k = 0;
 int i;


 for(int i = 0; i < LEN; i++)
 {
  if(i > M - 1)
  {
   x_delay[i] = x[k];
   k++;
  }
 }

 for(i = 0; i < LEN; i++)
 {
  y[i] += x[i] + g * x_delay[i];
 }
 heap_free(index, x_delay);
}
```

Slika 2.1.1 - Implementacije efekta kašnjenja u *CCES*

Ako se sada izbroje utrošeni ciklus, dobije se broj 5 711 737, što je poboljšanje od oko 200 hiljada ciklusa, što nije mnogo, ali je ipak poboljšanje.

Dalje ako posmatramo drugu *for* petlju (prva *for* petlja ostaje u svojoj poboljšanoj verziji). Da se primjetiti da je za prvih M odmjeraka, nepotrebno uzimati u obzir odmjerke iz *x_delay* niza, što može unijeti nepotrebna računanja, već samo iz originalnog niza. Petlja se može razložiti na dvije jednostavnije petlje

```
for(int i = 0; i < LEN; i++)
{
 if(i > M - 1)
 {
  x_delay[i] = x[k];
  k++;
 }
}
```

Slika 2.1.2 – Prva optimizacija efekta kašnjenja

```
for(i = 0; i < M; i++)
{
 y[i] = x[i];
}

for(i = M; i < LEN; i++)
{
 y[i] = x[i] + g * x_delay[i];
}
```

Slika 2.1.3 – Druga optimizacija efekta kašnjenja

Mjerenjem ciklusa dođe se do broja 5 684 200, što jeste napredak za nekih 35 hiljada ciklusa, ali to i nije nešto značajno. Ukoliko se iskoristi direktiva *#pragma optimize for speed* ili se podesi optimizacija kompajlera za brzinu kroz podešavanja u *CCES*, dobijaje se 4 063 507 što je poboljšanje za oko 1.5 miliona ciklusa. Na kraju možemo iskoristiti vektorizaciju petlji, pomoću direktive *#pragma SIMD_for* i dođe se do broja ciklusa 2 280 297. Ipak ukoliko se pogledaju konačni rezultati obrade u izlaznom nizu uoči se da odmjerci nisu ono što očekujemo na osnovu obrade u pajtonu (svaki drugi odmjerak je 0). Pokaže se da je ovo posljedica korištenja *SRAM* memorije za sve nizove u funkciji, pri vektorizaciji petlji. Ukoliko se koristi *DRAM* memorije pri vektorizaciji dobijaju se korektni rezultati. Motiv za prvobitno korištenje *SRAM* memorije jeste to što je ona brži tip memorije (ali manjeg kapaciteta), pa je zato alociran hip na *SRAM* memoriji i na početku rada alociran ulazni i izlazni bafer na ovom hipu, te su tamo smještani ulazni i izlazni odmjerci, kao i međurezultati po potrebi. Ukoliko želimo koristiti vektorizaciju petlji, treba koristiti *DRAM* memoriju, rezultat je 4 059 824. Očigledno je da to neće donijeti bolje rezultate u odnosu na slučaj kada se koristi *SRAM* uz sve izložene optimizacije (naravno bez vektorizacije petlji), Rezultati su u tabeli.

,,naivna'' implementacija	I poboljšanje	II poboljšanje	kompajlerske optimizacije	**DRAM* uz vektoriz. petlji
5 945 835	5 711 737	5 684 200	4 063 507	4 059 824

Tabela 2.1.1 - Pregled broja ciklusa efekta kašnjenja

2.2 Distortion (distorzija) efekat

Efekat distorzije spada u grupu efekata sa nelinearnom obradom. Termin nelinearna obrada podrazumijeva sve algoritme obrade signala koji ne zadovoljavaju princip linearnosti. Ono što rade nelinearni algoritmi obrade, jeste da unose dodatne spektralne frekvencijske komponente kojih nema u originalnom signalu. A za slučaj audio signala, to znači da će se promijeniti boja zvuka, što slušalac može jasno da čuje. U suštini ono što distorzija radi ulaznom signalu jeste da ga nelinearno transformiše tzv. nelinearnom krivom pojačanja. Konkretno kod efekta distorzije ta kriva je obično neka verzija eksponencijalne krive i postoji nekoliko funkcija kojim se može izvršiti distorzija. Jedna od najčešće korišćenih i ona koja je korištena u ovom zadatku je sljedeća:

$$y(n) = \frac{x}{|x|}(1 - e^{-\frac{x^2}{|x|}}) \qquad (2.2.1)$$

Osnovni parametri distorzije su pojačanje i miks. Prije primjene funkcije iznad, na ulazni signal, potrebno ga je pojačati određenim pojačanjem. Što se tiče miksa to je obično drugi podesivi parametar distorzije i on određuje koliko će u rezultatnom signalu biti udjela originalnog signala, a koliko obrađenog, po principu *(1 – mix) * original + mix * obrađeni*. Dakle miks ima vrijednost između 0 i 1. Često prije obrade radi normalizacija signala, kao i normalizacija konačnog obrađenog signala, da bi se izbjegla dodatna izobličenja uzrokovana odsjecanjem audio signala, zbog toga što izlazi izvan opsega vrijednosti koje je moguće fizički reprodukovati na zvučniku.

Što se tiče implementacije u *CCES,* osnovna implementacije je data na slici 2.2.1. Dakle na početku i na kraju imamo dio koda koji se odnosi na normalizaciju, i taj dio koda za sada nećemo posmatrati, ali on ulazi u ukupan broj ciklusa koje mjerimo. U suštini efekat se svodi na dvije petlje, prva u kojoj na ulazni signal dodajemo pojačanje, i drugo u kojoj se vrši sama distorzija po formuli (2.2.1). Implementacijom na ovaj način, dobija se trajanje efekta u ciklusima 24 532 441.

Prva stvar koja se može primjetiti jeste da u drugoj petlji, kada se računa *x_distort* ima veliki broj pristupa nizu *x_gain*. To je dosta neefikasno, jer je *x_gain* u eksternoj *SRAM* memoriji, i stalni pristupi ovoj memoriji su spori. Ovaj problem se lako rješava tako što jednom pročitamo trenutnu vrijednost niza

x_gain i smjestimo u promjenljivu. Slično možemo i apsolutnu vrijednost izračunati i smjestiti u promjenljivu i slično. Ovim poboljšanjem koda, dobijamo i popratno smanjenje broja ciklusa potrebnih za izvršavanje na 21 555 208 (oko 3 miliona manje ciklusa).

```
void distortion(float* x, float* y, float gain, float mix)
{

 float* x_distort = (float* )heap_calloc(index, LEN, sizeof(float));
 float* x_gain = (float* )heap_calloc(index, LEN, sizeof(float));

 /* .. kod za normalizaciju ulaznog niza .. */


 for(int i = 0; i < LEN; i++)
 {
  x_gain[i] = gain * x[i];
 }

 for(int i = 0; i < LEN; i++)
 {
  x_distort[i] = (x_gain[i] / fabsf(x_gain[i])) *
                    (1 - (expf(-1 * ((x_gain[i] * x_gain[i]) / fabsf(x_gain[i])))));
  y[i] = (1 - mix) * x[i] + mix * x_distort[i];

 }

 /* ..kod za normalizaciju izlaznog niza.. */

 heap_free(index, x_distort);
 heap_free(index, g_gain);
}
```

Slika 2.2.1 – Implementacija efekta distorzije u *CCES*

Sljedeća ideja je da se ostvari dvostruka dobit, i na polju brzine izvršavanja, a i na polju zauzeća memorije. Moguće je ukloniti nizove *x_gain i x_distort* te raditi direktno na ulaznom nizu *x*. Na ovaj način ćemo izbaciti dinamičku alokaciju memorije, te jednu petlju. Broj utrošenih ciklusa je sada 15 876 344 ciklusa, a to je za oko 2.5 miliona manje ciklusa (i dva niza dužine *LEN* manje u *SRAM* memoriji). Kod je na slici 2.2.3. Pošto sam iscrpio sve mogućnosti unapređenja koda, sada se može pribjeći kompajlerskim optimizacijama (12 182 598 ciklusa) te vektorizaciji petlji - *#pragma SIMD_for* (7 442 106). Ipak ovaj posljednji rezultat nije korektan. Sličan problem se dešava sa vektorizacijom petlji, kao kod efekta kašnjenja (izlazni rezultati nisu korektni), pa će ovaj metod biti provjeren na *DRAM* memoriji (uz naravno sve prethone optimizacije koda). Korištenjem *DRAM* memorije i vektorizacije petlji dobije se rezultat od 7 557 717, što jeste bolje nego najbolji korektan rezultat na *SRAM* memoriji (12 182 598 ciklusa), pa se u slučaju ovog efekta može uzeti u obzir korištenje *DRAM* memorije uz upotrebu vektorizacije petlji. Tabela sa pregledom broja ciklusa za pojedine korake je data Tabelom 2.2.1.

```
for(int i = 0; i < LEN; i++)
{
 float x_original = x[i];
 float x_value = x_gain[i];
 float abs_x = fabsf(x_value);
 float exp_x = expf(-1 * ((x_value * x_value) / abs_x));

 x_distort[i] = (x_value / abs_x) *(1 - exp_x);
 y[i] = (1 - mix) * x_original + mix * x_distort[i];

}
```

Slika 2.2.2 – Prvo poboljšanje koda efekta distorzije

```
void distortion(float* x, float* y, float gain, float mix)
{
 int i;

 /* .. normalizacija ulaza ..*/

 for(i = 0; i < LEN; i++)
 {
  float x_original = x[i];
  float x_value = x_original * gain;
  float abs_x = fabsf(x_value);
  float exp_x = expf(-1 * ((x_value * x_value) / abs_x));


  y[i] = (1 - mix) * x_original + mix * ((x_value / abs_x) * (1 - exp_x));

 }


 /* .. normalizacija izlaza .. */
}
```

Slika 2.2.3 – Drugo poboljšanje koda efekta distorzije

„naivna'' implementacija	I poboljšanje	II poboljšanje	kompajlerske optimizacije	* vektorizacije petlji na DRAM
24 532 441	21 555 268	15 876 344	12 182 598	7 557 717

Tabela 2.2.1 – Pregled broja ciklusa za razne varijante koda efekta distorzije

2.3 *Wah-Wah* (vah-vah) efekat

Vah-vah efekat spada u grupu efekata sa vremenski promjenljivim filtrima. Efekti iz ove grupe dobijaju se tako što se u toku obrade signala mijenjaju parametri filtra. Karakterističan primjer efekta iz ove grupe jeste vah-vah efekat. Za realizaciju efekta koristi se filtar propusnik opsega sa uskim propusnim opsegom. Ovakav filtar često se zove i pik filtar (eng. *peak*). Zatim, da bi se na osnovu ovog filtra kreirao vah-vah efekat, potrebno je „šetati“ centralnu frekvenciju po frekvencijskoj osi i provlačiti ulazni signal kroz te filtre i slati ih na izlaz.

Što se tiče implementacije ovog efekta u *CCES*, ona je sljedeća.

```
void wah_wah(float* x, float* y)
{
 int i,j;

 float a[3];
 float b[3];

 for(i = 2; i < LEN; i++)
 {
  for(j = 0; j < 3; j++)
  {
   a[j] = peak_a_coeff[(i-2)*3 + j];
   b[j] = peak_b_coeff[(i-2)*3 + j];
  }
  y[i] = b[0]*x[i] + b[1]*x[i-1] + b[2]*x[i-2] - a[1]*y[i-1] - a[2]*y[i-2];
 }
}
```

Slika 2.3.1 – Implementacija vah-vah efekta u *CCES*

Pošto su u pajtonu generisani i eksportovani koeficijenti, kod nije pretjerano kompleksan. Potrebno je samo na osnovu tih koeficijenata odgovarajućom jednačinom diferencija računati izlazne koeficijente. Dakle pošto je u pajtonu korištena funkcija *iir_peak* koja vraća vrijednosti koeficijenata *IIR* filtra, ovde imamo problem jer ta funkcija nije na raspologanju. Rješenje je da se kroz .h fajl koeficijenti izvezu kao niz, zatim da se taj fajl uključi u *CCES* projekat i tako dalje koristi. Pošto se filtar projektuje za svaku centralnu frekvenciju, a njih ima koliko je i dužina signala (*LEN*), a svaki filtar ima po tri koeficijenta (dakle *LEN * 3)* a to može biti veliki broj, ovo predstavlja opterećenje za memoriju. Razmišljanje u pogledu optimizacije memorije treba biti u pravcu boljeg rješenja za izvoz ili računanje ovih koeficijenata.

Što se tiče brzine, osnovni način implementacije daje broj ciklusa 17 590 162. Prijedlog poboljšanja je da se pokuša neki vid *loop unrolling-a,* to jest da se unutrašnja petlja koja ima tri iteracije, razmota i da se svaki koeficijent dohvati posebno u statičke nizove a i b dužine 3. Dobijeni rezultat na ovaj način u pogledu broja ciklusa iznosi 12 313 129 ciklusa, što je značajno poboljšanje (oko 5 miliona ciklusa).

```
float a[3];
float b[3];
for(i = 2; i < LEN; i++)
{
 a[0] = peak_a_coeff[(i-2)*3];
 a[1] = peak_a_coeff[(i-2)*3 + 1];
 a[2] = peak_a_coeff[(i-2)*3 + 2];

 b[0] = peak_b_coeff[(i-2)*3];
 b[1] = peak_b_coeff[(i-2)*3 + 1];
 b[2] = peak_b_coeff[(i-2)*3 + 2];

 y[i] = b[0]*x[i] + b[1]*x[i-1] + b[2]*x[i-2] - a[1]*y[i-1] - a[2]*y[i-2];

}
```

Slika 2.3.2 – Prvo poboljšanje razmotavanjem petlje

Pošto je *a[0]* koeficijent uvijek 1 što znači da su svi koeficijenti normalizovani sa ovim koeficijentom, njega ne treba ni računati (on stoji uz *y[i]*). Također, pokaže se da je *b[1]* uvijek 0, pa ni njega ne treba računati, i u tom slučaju broj ciklusa je 10 602 354(ušteda dodatnih 1.7 miliona ciklusa). Sličan broj ciklusa dobije i kada bi se koeficijenti direktno dohvatali (bez posrednih statičkih nizova), broj ciklusa je tada 10 602 460. Ova implementacija je prikazana na sljedećoj slici:

```
for(i = 2; i < LEN; i++)
{

 y[i] = peak_b_coeff[(i-2)*3] * x[i] + peak_b_coeff[(i-2)*3 + 1] * x[i-1] + peak_b_coeff[(i-2)*3 + 2]*x[i-2]
   - peak_a_coeff[(i-2)*3 + 1] * y[i-1] - peak_a_coeff[(i-2)*3 + 2] * y[i-2];

}
```

Slika 2.3.3 – Alternativno rješenje sa direktnim dohvatanjem koeficijenata

Možda je očekivano da ovakav pristup da bolje rezultate, ali pošto su koeficijenti filtra u *DRAM* memoriji, a ulazni i izlazni niz u *SRAM* memoriji, a u jednačini diferencija naizmjenično dohvatamo odmjerak ulaza, pa koeficijent filtra, tu dolazi do overheda i nemamo bolje rezultate. Ipak izloženo rješenje sa slike 2.3.2 je sa preglednijim kodom, a po pitanju performansi nema značajne razlike, pa se to rješenje uzima u obzir. Na ovo rješenje sada možemo primjeniti kompajlerske optimizacije i posmatrati šta će se desiti. Dobijaju se dobri rezultati, jer je sada broj utrošenih ciklusa 8 504 467. (za slučaj sa slike 2.3.3 primjenom kompajlerskih optimizacija dobije se 8 504 230 ciklusa, što je približan rezultat). Ostaje

još da kao i u prethodnim efektima, primjenim vektorizaciju petlji, koja ne daje neke posebne rezultate. Broj ciklusa se svede na 8 319 648. Ovde nema problema prilikom vektorizacije petlji i korištenja *SRAM* memorije, pa s obzirom da je ona brža od *DRAM* ostaćemo pri njoj. U sljedećoj tabeli se nalazi pregled utrošenih ciklusa za vah vah efekat.

,,naivna'' implementacija	poboljšanje sa razmotavanjem petlje	poboljštanje sa računanjem a[0] i b[1] unaprijed	kompajlerske optimizacije	vektorizacije petlji (*SRAM)*
17 590 162	12 313 129	10 602 354	8 504 467	8 319 648

Tabela 2.3.1 – Pregled broja ciklusa tehnika optimizacije vah vah efekta

2.4 *Phaser* (fejzer) efekat

Pripada istoj grupi efekata kao i vah-vah efekat, i u suštini jako su slični, i na neki način inverzni jedan u odnosu na drugog. Princip funkcionisanja je isti, postoji centralna frekvencija koja je promjenljiva i na osnovu koje se kreiraju filtri, kroz koje se onda propušta ulazni signal. Razlika je u tome što se kod fejzer efekta, umjesto filtra propusnika opsega, koristi filtar nepropusnik opsega sa centralnom frekvencijom i jako uskim opsegom ili tzv. noč (eng. *notch)* filtar. To je filtar koji, idealno, propušta sve komponente frekvencija osim jedne, a to je komponenta centralne frekvencije, koju jako slabi.

Realizacija fejzer efekta u *CCES* je gotovo identična implementaciji vah vah efekta, stoga nema potrebe detaljno razmatrati pojedinosti implementacije. Jedina razlika u odnosu na vah vah efekat jeste što se koeficijenti filtra dohvataju iz drugih .h fajlova. A to su *iir_notch_a.h* i *iir_notch_b.h*. Osim toga razlika je što se kod fejzer efekta preporučuje dvostruko filtriranje, to jest da se filtriranje obavi jednom na osnovu promjenljivih centralnih frekvencija smješenih u niz trougoanog oblika i da se rezultati ne smjeste u konačni izlazni niz, već u pomoćni niz. A onda da se filtriranje obavi na identičan način, ali sada je ulaz, umjesto originalnog ulaznog signala, prethodno izračunati pomoćni niz, a izlaz je izlazni signal. Realizacija efekta je na slici 2.4.1. Poboljšanje se može postići razmotavanjem petlje, kao što smo vidjeli kod vah vah efekta, ali ovde se može ići i korak dalje te umjesto dvije *for* petlje, kako je na prvu bilo zamišljeno, sve smjestiti u jednu petlju, kao na slici 2.4.2. Kao i kod vah-vah efekta, nije bilo problema sa vektorizacijom petlji, pa ostajemo pri korištenju *SRAM* memorije. Ali pokaže se da kada se koristi samo jedna petlje vektorizacije petlje ne daje bolje rezultate u odnosu na situaciju kada se koriste samo kompajlerske optimizacije.

```
float a[3];
float b[3];

for(int i = 2; i < LEN; i++)
{
 for(j = 0; j < 3; j++)
 {
  a[j] = notch_a_coeff[(i-2)*3 + j];
  b[j] = notch_b_coeff[(i-2)*3 + j];
 }

 y_first[i] = b[0]*x[i] + b[1]*x[i-1] + b[2]*x[i-2] - a[1]*y_first[i-1] - a[2]*y_first[i-2];
}

for(int i = 2; i < LEN; i++)
{
 for(j = 0; j < 3; j++)
 {
  a[j] = notch_a_coeff[(i-2)*3 + j];
  b[j] = notch_b_coeff[(i-2)*3 + j];
 }

 y[i] = b[0]*y_first[i] + b[1]*y_first[i-1] + b[2]*y_first[i-2] - a[1]*y[i-1] - a[2]*y[i-2];

}
```

Slika 2.4.1 – Implementacija fejzer efekta

```
for(int i = 2; i < LEN; i++)
{
 a[1] = notch_a_coeff[(i-2)*3 + 1];
 a[2] = notch_a_coeff[(i-2)*3 + 2];

 b[0] = notch_b_coeff[(i-2)*3 + 0];
 b[1] = notch_b_coeff[(i-2)*3 + 1];
 b[2] = notch_b_coeff[(i-2)*3 + 2];

 y_first[i] = b[0]*x[i] + b[1]*x[i-1] + b[2]*x[i-2] - a[1]*y_first[i-1] - a[2]*y_first[i-2];
 y[i] = b[0]*y_first[i] + b[1]*y_first[i-1] + b[2]*y_first[i-2] - a[1]*y[i-1] - a[2]*y[i-2];
}
```

Slika 2.4.2 – Poboljšanje razmotavanjem petlje i spajanjem u jednu petlju

„naivna'' implementacija	poboljšanje sa razmotavanjem petlje	poboljštanje sa računanjem a[0] unaprijed	poboljšanje sa jednom for petljom	kompajlerske optimizacije
35 676 735	25 079 922	23 298 447	17 660 848	10 757 170

Tabela 2.4.1 – Pregled broja ciklusa za razne tehnike optimizacije fejzer efekta

	prije optimizacija	kompajlerske op.	vector. petlji
jedna petlja	17 660 848	10 757 170	11 507 170
dvije petlje	23 298 447	19 454 853	17 597 976

Tabela 2.4.2 – Pregled broja ciklusa za slučaj korištenja jedne i dvije petlje

Iz rezultata se vidi da slučaj kada modifikujemo kod da koristi jednu petlje dobijamo bolje rezultate i na početku, ali i kada se uključe kompajlerske optimizacije. Međutim vektorizacijom petlji, u slučaju dvije petlje, primjeti se napredak, ali to je još uvijek lošiji rezultat nego sa jednom petljom. Sa druge strane kod implementacije sa jednom petljom,rezultat sa vektorizacijom je lošiji nego bez nje, pa se ova tehnika ovde neće koristiti. To se može objasniti velikom zavisnošću podatka unutar te jedne petlje, pa kompajler nije uspio da je paralelizuje, već je samo trošio dodatno vrijeme pokušavajući da to uradi.

2.4 *Reverberation* (reverberacije) efekat

Efekat reverberacije u tekstu projektnog zadatka smješten je u grupu specijalnih efekata, a često se u literaturi smješta i u grupu prostornih efekata [1]. Jedan od najpoznatijih efekata iz ove grupe je efekat poznat pod nazivom reverberacije. Jednostavna realizacija je korištenjem FIR filtra koji se može opisati sljedećom jednačinom diferencija:

$$y(n) = x(n) + \alpha x(n-R) + \alpha^2 x(n-2R) + \cdots + \alpha^{N-1} x(n-(N-1)R) \qquad (2.5.1)$$

Na ovaj način simulira se prvo direktni put talasa od izvora ka slušaocu, a zatim i višestruke refleksije talasa koje pristižu sa određenim kašnjenjem i oslabljene do slušaoca. Svaka refleksija koja je više zakašnjena, tj. kasnije stiže do slušaoca, je više oslabljena zbog stepenovanja parametra α.Parametar R određuje nakon koliko vremena se pojavljuje prva refleksija, tj. koliko je zakašnjena. A parametar N određuje koliko refleksija će se desiti za konkretni odmjerak. Ovi parametri su fiksni za sve odmjerke, iako realno oni mogu biti promjenljivi, ali to i jeste glavni nedostatak ovog načina realizacije reverberacija.

Što se tiče implementacije reverberacija u *CCES,* slično kao u pajtonu, sve se svodi na dvije *for* petlje, gdje jedna prolazi kroz sve odmjerke ulaza, a druga ide od 0 do parametra *N* i određuje broj odmjeraka koji će biti zakašnjeni i sabrani sa ulaznim odmjerom uz određeno slabljenje. Kod je na slici:

```
void reverb(float* x, float* y, float alpha, int N, int R)
{
 for(int i = 0; i < LEN; i++)
 {
  for(int k = 0; k < N; k++)
  {
   y[i] += pow(alpha, k) * ((i > k*R) ? x[i - k*R] : 0);
  }
 }

    /* .. normalizacija izlaznog niza .. */
}
```

Slika 2.5.1 – Implementacija efekta reverberacija

Međutim, ako se malo razmisli šta zapravom radimo ovim efektom može se doći do boljih implementacija u pogledu performansi na *ADSP*. Unutrašnju petlju možemo pojednostaviti, tako što znamo da za *k = 0* u izlazni signal, uvijek dodajemo samo originalni ulazni, bez kašnjenja (*y[i] = x[i]).* Na ovaj način se broj ciklusa može spustiti sa 90 miliona, na 85 miliona. Kod je na slici 2.5.2. Dalje možemo primjetiti sljedeću stvar. Pošto parametri efekta *N* i *R* tipično nisu veliki brojevi, tj. dosta su manji od ukupnog broja odmjeraka signala, većina vremena u unutrašnjoj petlji potroši se na provjeru uslova koji je već ispunjen. Dakle može se prvo provjeriti da li je trenutna vrijednost odmjerka veća od umnoška *N* i *R,* te ako jeste nema potrebe provjeravati uslov u unutrašnoj petlji, a ako nije onda se može provjeriti taj uslov i ići dalje u skladu sa tim. Kod je na slici 2.5.3. U suštini iskorištena je ideja da je bolje imati dvije identične petlje, jednu u *if* grani, drugu u *else* grani, nego imati grananje unutar petlje. Doduše ovde još uvijek ostaje jedno grananje unutar petlje, ali bar je jedna petlja bez grananja i to dosta znači za performanse. Na ovaj način se potroši 82 miliona ciklusa, što je oko 3 miliona uštede u odnosu na prethodni kod.

```
for(int i = 0; i < LEN; i++)
{
 y[i] = x[i];
 for(int k = 1; k < N; k++)
 {
  if(i > k*R)
  {
   y[i] += pow(alpha, k) * x[i - k*R];
  }
 }
}
```

Slika 2.5.2 – Prvo poboljšanje, izbacivanje prve iteracije

```
for(int i = 0; i < LEN; i++)
{
        y[i] = x[i];
        if(i > N*R)
        {
                for(int k = 1; k < N; k++)
                {
                        y[i] += pow(alpha, k) * x[i - k*R];
                }
        } else
               {
                for(int k = 1; k < N; k++)
                {
                        if(i > k*R)
                        {
                         y[i] += pow(alpha, k) * x[i - k*R];
                        }
                }
        }
}
```

Slika 2.5.3 – Drugo poboljšanje efekta reverberacija

,,naivna'' implementacija	poboljšanje sa vađenjem prve iteracija van petlje	poboljštanje sa vađenjem uslova van petlje	kompajlerske optimizacije	vektorizacija petlji (*DRAM*)
90 282 822	85 241 243	82 602 613	72 61 829	73 654 014

Tabela 2.5.1 – Pregleda broja ciklusa za razne varijante optimizacije efekta reverberacija

Vektorizacijom petlji dolazi do pogrešnih rezultata obrade pri *SRAM* memoriji, slično kao za prva dva efekta, pa je poslednja optimizacija dodavanje kompajlerskih optimizacija i broj utrošenih ciklusa je 72 161 829. Sa druge strane ako se koristi *DRAM* memorija, moguće je koristiti vektorizaciju i data je broj ciklusa 73 654 014, što je lošiji rezultat od najboljeg na *SRAM-u*, pa nema potrebe uvoditi *DRAM* memoriju i vektorizaciju. Tabela 2.5.1 sadrži pregled broja ciklusa za pojedine varijante optimizacije..

3. Zaključak

Nakon obrade svih efekata i primjene raznih tehnika optimizacije može se reći sljedeće. Prilikom analize koda krenuti od neke osnovne implementacije, a zatim se upoznati sa radom *DSP* sistema uopšte, a naročito konktretnog sistema na kojem se radi, te pokušati preurediti kod na način da se dobiju što bolje performanse. Ovakvim načinom rada kod svih algoritama, u ovom zadatku, došlo se određenih poboljšanja (i do 40% kod nekih efekata prije uvođenja kompajlerskih optimizacija, kao npr. fejzer). Zatim

svakako treba iskoristiti mogućnosti kompajlera i uvesti kompajlerske optimizacije kroz podešavanja, jer se tako mogu dobiti još značajnija poboljšanja, ali prije kompajerskih optimizacija uvijek treba pokušati sa nekim vidom preuređenja koda, korištenja, ako je to moguće, ugrađenih specifičnih funkcija ploče i slično. Na kraju treba voditi računa i o memoriji koja se koristi pri radu. U ovom izvještaju vidi se da je to itekako uticalo na performanse, ali i ispravnost rezultata. Vektorizacija petlji nije se uvijek mogla izvršiti ako se koristila eksterna *SRAM* memorija, već se morala iskoristiti sporija eksterna *SDRAM(DRAM)* memorija, dakle i na to treba obratiti pažnju. U ovom zadatku nisu korištene mogućnosti interne memorije, recimo programske memorije, koja bi mogla dobijenti značajna poboljšanja i to je sljedeći korak o kojem se može razmišljati u pogledu unapređenja performansi ovog projekta. Ideja bi mogla biti ping-pong baferi manjeg kapaciteta i korištenje *DMA,* gdje bi se recimo koeficijenti filtara mogli smještati na programsku memoriju, jer su fiksni, a odmjerci mogli dohvatati u manjim blokovima, ili možda realizacija metoda *overlap-add* ili *overlap-save* za neke metode, gdje bi ulazne odmjerke smještali u manji bafer koji može stati u programsku memoriju i tako računali izlazne odmjerke. Također, kod primjene vah-vah i fejzer efekta, trebalo bi razmišljati o boljem načinu realizacije *iir* filtara, jer bi kako se povećava ulazni niz i ukupna veličina fajlova sa koeficijentima smještenim na ovaj način rasla i postala preveliko opterećenje za memoriju. Trenutni način realizacije nije pretjerano efikasana i samo je iskorištena osnovna ideja, za demonstraciju rada efekata, svakako bi u daljem radu trebalo naći neko bolje rješenje.

1. Literatura

[1] Vladimir Risojević. *Multimedijalni sistemi*. Univerzitet u Banjoj Luci, Elektrotehnički fakultet, 2018.

[2] Udo Zolzer. DAFX: *Digital Audio Effects.* John Willey and Sons Ltd, 2011.

[3] Nastavni materijali sa predmeda Sistemi za digitalnu obradu signala i Osnovi digitalne obrade signala