Multidiszciplináris tudományok, 4. kötet. (2014) 1. sz. pp. 31-38.

FIR SZŰRŐK TELJESÍTMÉNYÉNEK JAVÍTÁSA C/C++-BAN

Lajos Sándor
Mérnöktanár, Miskolci Egyetem, Matematikai Intézet,
Ábrázoló Geometriai Intézeti Tanszék
3515 Miskolc, Miskolc-Egyetemváros, e-mail: LajosS@abrg.uni-miskolc.hu

Összefoglalás

A digitális jelfeldolgozás során, ha fontos a lineáris fázis jelleggörbe, FIR szűrőket alkalmazunk. A nagy fokszámú FIR szűrők megvalósítása viszont jelentős számítási teljesítményt igényel, mely a cikkben bemutatott optimalizálási módszerek segítségével a töredékére csökkenthető.

Kulcsszavak: *FIR szűrő, optimalizálás, C/C++.*

Abstract

In digital signal processing FIR filters are used, when the linear phase characteristic is important. The realization of high order FIR filters uses lots of CPU power, which can be reduced by the proposed optimization methods.

Keywords: *FIR filter, optimization, C/C++.*

1. Bevezetés

A digitális jelfeldolgozásban alkalmazott szűrők két típusra oszthatók:

- Véges impulzusválaszú FIR (**F**inite **I**mpulse **R**esponse)
- Végtelen impulzusválaszú IIR (**I**nfinite **I**mpulse **R**esponse).

Mindkét típusnak vannak előnyei. Egy IIR szűrő megvalósítása kevesebb memóriát és számítást igényel, mint egy hasonló tulajdonságokkal rendelkező FIR szűrő. Másrészről a FIR szűrők fázis jelleggörbéje lehet lineáris, vagyis a szűrők nem torzítják a jel fázisát. Ez utóbbi tulajdonságuk miatt sok esetben előnyben részesítik a FIR szűrőket [1].

2. FIR szűrők megvalósítása

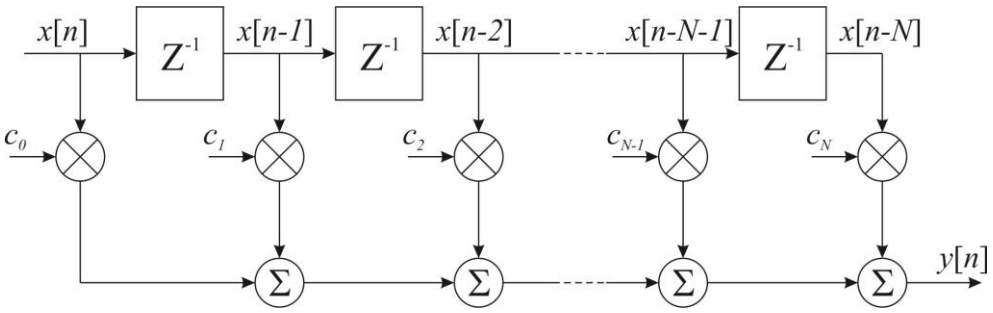
A FIR szűrők direkt megvalósítási formája az 1. ábrán látható. Ennek a szűrőstruktúrának az output/input viszonya az alábbi egyenlettel írható le [2]:

$$y[n] = \sum_{i=0}^{N} c_i x[n-i] \tag{1}$$

ahol:

- $x[n]$ az bemenő jel n-edik mintája,
- $y[n]$ a kimenő jel n-edik mintája,

- N a szűrő fokszáma (rendje),
- $c_{0...N}$ a szűrőegyütthatók.



1. ábra. *A FIR szűrők direkt megvalósítási formája.*

Az (1) egyenlet alapján egy FIR szűrő megvalósításához két *N+1* elemű vektor skaláris szorzatát kell kiszámítanunk. Ehhez a programkódban mintánként *N+1* darab szorzás és *N+1* darab összeadás szükséges. Bizonyos feladatokhoz nagy (akár több ezer) fokszámú szűrőket kell alkalmazni. Ilyenkor már a vektorok skaláris szorzása is jelentős számítási teljesítményt igényel, nem beszélve a minták tárolásához szükséges járulékos számításokról.

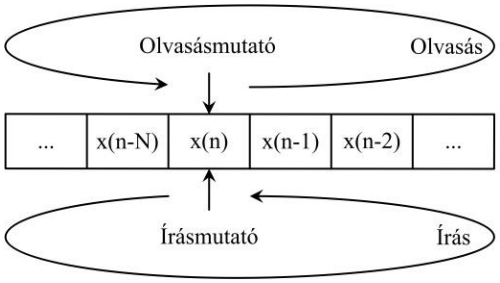
A következő fejezetekben szereplő programkódok tesztelése során megmértük, hogy mennyi processzorciklus szükséges 44 100 darab véletlenszerű minta feldolgozásához.

3. Körkörös puffer

Az 1. ábrán látható szűrőstruktúrához tárolnunk kell a bemenő jel előző *N* darab mintáját. Ezt a feladatot egy körkörös puffer segítségével oldjuk meg [3].

3.1. Körkörös írás, körkörös olvasás

Első lépésben a legegyszerűbb körkörös puffert fogjuk alkalmazni, amikor az adatoknak az írása és az olvasása is körkörösen történik (2. ábra).



2. ábra. *Puffer körkörös írással és olvasással.*

Ebben az esetben a FIR szűrő programkódja a következő:

```
void Fir(float *x, float *y, int ns, int N)
{
   register int i, j, rP;
   float o;

   for(i = 0;i < ns;i++)
   {
      o = 0;
      b[wP] = x[i];
      for(j = 0;j <= N;j++)
      {
         rP = wP+j;
         if(rP > N)
            rP -= N+1;
         o += b[rP] * c[j];
      }
      if(--wP < 0)
         wP = N;
      y[i] = o;
   }
}
```

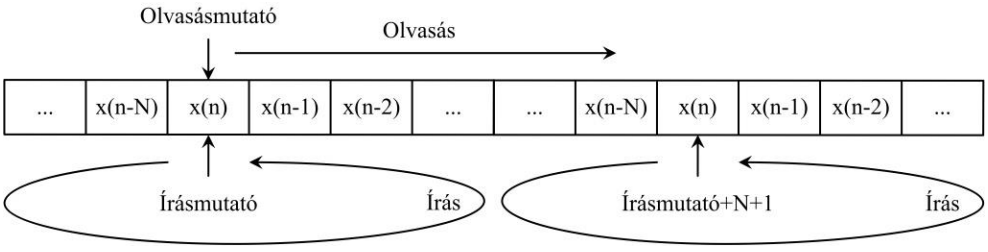
A függvény az x tömbben kapja az bejövő mintákat, az y tömbben adja vissza a feldolgozott mintákat. Az ns a feldolgozandó minták száma, N a szűrő fokszáma. A wP globális változó az körkörös puffer írásmutatója, az rP pedig az olvasásmutatója. A b globális tömb maga a körkörös puffer, a c globális tömb pedig a szűrőegyütthatókat tárolja.

A fenti kód végrehajtásához 199 269 ezer processzorciklus szükséges (5. ábra).

3.2. Körkörös írás lineáris olvasás

Az előzőekben vázolt puffer hátránya, hogy mind az írás, mind az olvasás során vizsgálnunk kell a puffermutatót, hogy az elérte-e a puffer végét. Ez a vizsgálat lassítja a feldolgozást.

Ezért megduplázzuk a puffer méretét és minden mintát kétszer írunk be. Ebben az esetben lehetővé válik, hogy az olvasás lineáris legyen, nincs szükség az olvasásmutató vizsgálatára (3. ábra).



3. ábra. *Puffer körkörös írással, lineáris olvasással.*

Ebben az esetben a FIR szűrő programkódja a következő:

```
void Fir(float *x, float *y, int ns, int N)
{
   register int i, j, rP;
   float o;

   for(i = 0;i < ns;i++)
   {
      o = 0;
      b[wP] = b[wP + N+1] = x[i];

      for(j = 0;j <= N;j++)
      {
         rP = wP+j;
         o += b[rP] * c[j];
      }
      if(--wP < 0)
         wP = N;
      y[i] = o;
   }
}
```

A fenti kód végrehajtásához már csak 73 484 ezer processzorciklus szükséges (5. ábra).

4. Ciklusok kibontása

Egy FIR szűrő programkódja két egymásba ágyazott ciklusból áll. A külső ciklus a feldolgozandó mintákon lép végig, míg a belső ciklus végzi a két vektor skaláris szorzását. A ciklusok adminisztrálása azonban időigényes feladat. Minden egyes iteráció során léptetni kell a ciklusváltozót, majd vizsgálni kell, hogy elérte-e a ciklus végét. Ezek a ciklusadminisztrációs feladatok csökkenthetők a ciklusok kibontásával [4].

Jelen esetben a cikluskibontást úgy valósítjuk meg, hogy a ciklusmagban négy műveletet végzünk el, és a ciklusváltozót négyesével léptetjük.

Csak a belső ciklust kibontva a végrehajtáshoz szükséges processzorciklusok száma 71 160 ezerre csökken (5. ábra). Mindkét ciklust kibontva már csak 68 751 ezer processzorciklus szükséges a végrehajtáshoz (5. ábra). Ebben az esetben a FIR szűrő programkódja a következő:

```
void Fir(float *x, float *y, int ns, int N)
{
   register int i, j, rP;
   float o, o1, o2, o3;

   for(i = 0;i < ns;i+=4)
   {
      o = o1 = o2 = o3 = 0;
      b[wP] = b[wP + N+1] = x[i];
      b[wP-1] = b[wP + N] = x[i+1];
      b[wP-2] = b[wP + N-1] = x[i+2];
      b[wP-3] = b[wP + N-2] = x[i+3];

      for(j = 0;j <= N;j+=4)
```

```
      {
         rP = wP+j;
         o += b[rP] * c[j] + b[rP+1] * c[j+1] + b[rP+2] * c[j+2] +
                     b[rP+3] * c[j+3];
         o1 += b[rP-1] * c[j] + b[rP] * c[j+1] + b[rP+1] * c[j+2] +
                     b[rP+2] * c[j+3];
         o2 += b[rP-2] * c[j] + b[rP-1] * c[j+1] + b[rP] * c[j+2] +
                     b[rP+1] * c[j+3];
         o3 += b[rP-3] * c[j] + b[rP-2] * c[j+1] + b[rP-1] * c[j+2] +
                     b[rP] * c[j+3];
      }

      wP -= 4;
      if(wP < 0)
         wP = N;

      y[i] = o;
      y[i+1] = o1;
      y[i+2] = o2;
      y[i+3] = o3;
   }
```

5. SIMD utasítások

A modern processzorok multimédiás utasítás-kiegészítései egy utasítással egyszerre több adaton végzik el ugyan azt a műveletet. Ezeket az utasításokat egységesen SIMD (**S**ingle **I**nstruction **M**ultiple **D**ata) utasításoknak nevezzük. Ezek az új kiegészítések új regisztereket is igényelnek. Ezek a regiszterek a betöltött adat formátumától és a kiadott utasítástól függően úgy viselkednek, mintha több önálló regiszterre lennének felosztva. Egy 128 bites regiszterbe például egyszerre négy 32 bites adat tölthető be, és ha ennek a regiszternek a tartalmát például hozzáadjuk egy másik, ugyanilyen módon feltöltött regiszterhez, akkor nem két számot adunk össze, hanem egyetlen utasítással négy számpárt, vagyis ebben az esetben a négyszeresére nőtt a számítási teljesítmény.

Az Intel a lebegőpontos számokkal dolgozó utasításkészlet kiegészítését, az SSE-t (**S**treaming **S**IMD **E**xtensions) a Pentium III-mal vezette be 1999-ben. Az SSE azóta számos fejlesztésen, bővítésen esett át. Az SSE4 már lehetővé teszi két darab négyelemű vektor skaláris szorzását egyetlen utasítással [5]. Ennek felhasználásával az előző programkód az alábbiak szerint alakítható át:

```
void Fir(float *x, float *y, int ns, int N)
{
   register int i, j;
   float *rP;
   __m128 o, o1, o2, o3, cm, zero, b0, b1, b2, b3;

   zero = _mm_setzero_ps();

   for(i = 0;i < ns;i+=4)
   {
```

```
      o = o1 = o2 = o3 = zero;

      b[wP] = b[wP + N+1] = x[i];
      b[wP-1] = b[wP + N] = x[i+1];
      b[wP-2] = b[wP + N-1] = x[i+2];
      b[wP-3] = b[wP + N-2] = x[i+3];

      for(j = 0;j <= N;j+=4)
      {
         rP = b+wP+j;

         cm = _mm_load_ps(c+j);

         b0 = _mm_loadu_ps(rP);
         b1 = _mm_loadu_ps(rP-1);
         b2 = _mm_loadu_ps(rP-2);
         b3 = _mm_loadu_ps(rP-3);

         o = _mm_add_ss(_mm_dp_ps(b0,cm,0xf1),o);
         o1 = _mm_add_ss(_mm_dp_ps(b1,cm,0xf1),o1);
         o2 = _mm_add_ss(_mm_dp_ps(b2,cm,0xf1),o2);
         o3 = _mm_add_ss(_mm_dp_ps(b3,cm,0xf1),o3);
      }

      wP -= 4;
      if(wP < 0)
         wP = N;

      _mm_store_ss(y+i,o);
      _mm_store_ss(y+i+1,o1);
      _mm_store_ss(y+i+2,o2);
      _mm_store_ss(y+i+3,o3);
   }
}
```

A fenti kód végrehajtásához mindössze 50 163 ezer processzorciklus szükséges (5. ábra).

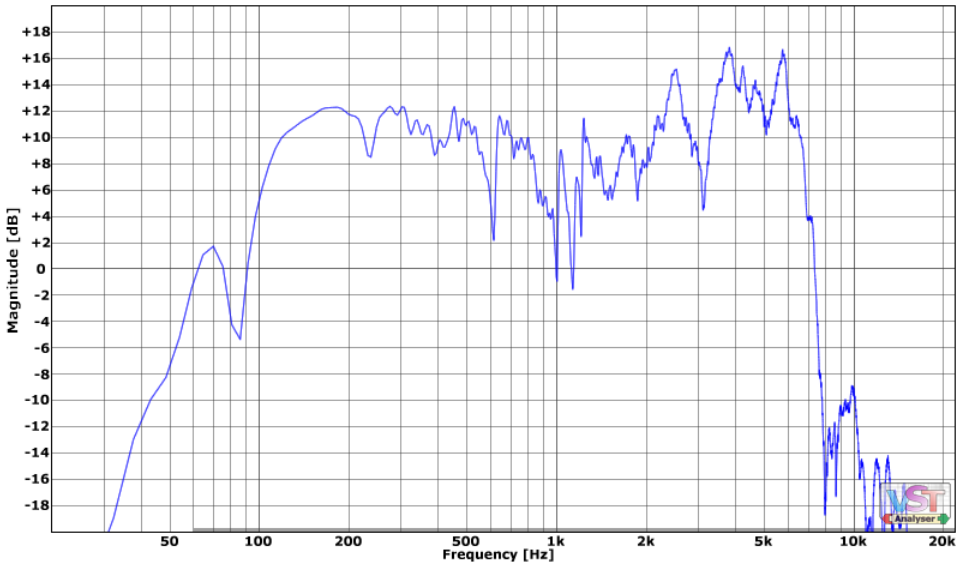
6. Eredmények

A cikkben bemutatott programkódok teszteléséhez egy VST modult készítettünk a Steinberg cég által kiadott VST SDK felhasználásával [6]. A fordításhoz a Microsoft Visual Studio 2008-at használtuk a következő optimalizálási kapcsolókkal: /Ox, /Ot, /GL, valamint letiltottuk, hogy a fordító SSE utasításokat használjon. A teszthardver egy Intel Core i7-2760QM @ 2,4GHz processzorral felszerelt számítógép volt.

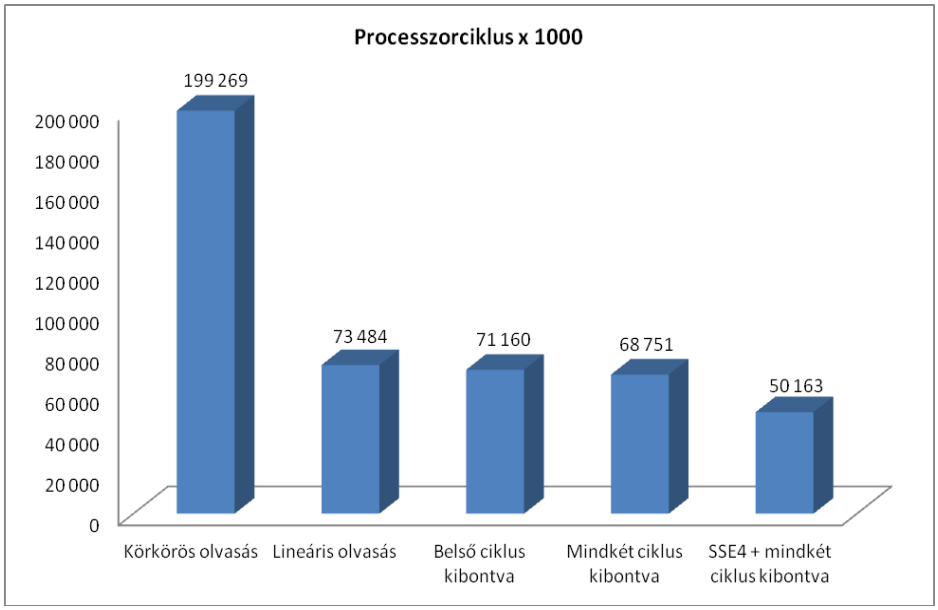
A tesztelést egy 2047 fokszámú FIR szűrővel végeztük. A szűrő együtthatóiként egy Celestion V30 hangszórókkal szerelt gitárhangládán mért impulzusválaszfájl első 2048 értékét használtuk [7]. Ebben az esetben a FIR szűrő frekvenciakarakterisztikája a 4. ábrán látható.

A méréseket a VST Plugin Analiser programmal végeztük [8].

A különféle optimalizálási lépések eredményeként elérhető sebességnövekedést az 5. ábra mutatja.



4. ábra. *A teszteléshez használt FIR szűrő frekvenciakarakterisztikája.*



5. ábra. *Futtatási eredmények 44 100 db véletlen minta feldolgozása esetén.*

7. Összefoglalás

A fentiekben bemutatott optimalizálási lépések jól mutatják, hogy már a programkód megfelelő strukturálásával is jelentős sebességnövekedés érhető el. Emellett további jelentős javulás érhető el a modern processzorok lehetőségeinek kihasználásával. A legutolsó kód futtatásához nagyjából negyedannyi processzorciklusra volt szükség, mint a legelső kód esetében.

8. Köszönetnyilvánítás

A kutató munka a Miskolci Egyetem stratégiai kutatási területén működő Mechatronikai és Logisztikai Kiválósági Központ keretében valósult meg.

9. Irodalom

[1] Zölzer, U.: DAFX-Digital Audio Effects, John Wiley & Sons, 2002.
[2] http://en.wikipedia.org/wiki/Finite_impulse_response
[3] http://en.wikipedia.org/wiki/Circular_buffer
[4] http://en.wikipedia.org/wiki/Loop_unwinding
[5] Streaming SIMD Extensions (SSE), http://msdn.microsoft.com/en-us/library/t467de55(v=vs.90).aspx
[6] http://www.steinberg.net/en/company/developers.html
[7] http://www.redwirez.com/bigbox.jsp
[8] http://www.savioursofsoul.de/Christian/programs/measurement-programs/