ГОДИШНИК на Минно-геоложкия университет "Св. Иван Рилски", Том 53, Св.III, Механизация, електрификация и автоматизация на мините, 2010
ANNUAL of the University of Mining and Geology "St. Ivan Rilski", Vol. 53, Part III, Mechanization, electrification and automation in mines, 2010

ПЛАТФОРМЕНООРИЕНТИРАНА ВРЕМЕВА ОПТИМИЗАЦИЯ НА БЪРЗОТО ФУРИЕ ПРЕОБРАЗУВАНЕ

Ясен Видолов

Минно-геоложки университет "Св. Иван Рилски", 1700 София, e-mail: nre@abv.bg

РЕЗЮМЕ. Предложена е софтуерна реализация на Бързата Фурие Трансформация, при която чрез използване на особеностите на архитектурата на процесорите от фамилия Intel Pentium и с използване на SSE инструкции е постигнато повишаване на бързодействието с около 30% в сравнение с класическия подход за реализация на тази трансформация.

ARCHITECTURE DEPENDANT REALIZATION OF FAST FOURIER TRANSFORM BY USING SSE INSTRUCTION SET

Yasen Vidolov
University of Mining and Geology "St. Ivan Rilski", 1700 Sofia, e-mail: nre@abv.bg

ABSTRACT. The suggested Fast Fourier Transform code utilizes Intel's Pentium SSE instruction set. The code written in assembler is highly dependent on the architecture. It reduces computation time with 30% than its C-code equivalent.

Въведение

Широкото приложение на Бързата Трансформация на Фурие в различни клонове от съвременната наука е предпоставка за стимулиране търсенето на нови решения за оптимизирането й, като една от целите е минимизиране на времето, необходимо за осъществяване на преобразуването.

Един от пътищата за повишаване на бързодействието е платформеноориентираната оптимизация при която програмната реализация на алгоритъма е съобразена с конкретната изчислителна платформа и ефективно използва ресурсите й. Разбира се това е свързано с компромис по отношение на преносимостта и универсалността на програмния код.

Поради широко наложилата се компютърна архитектура на Intel - Pentium в разработката е предложен код за реализиране на Бързата Трансформация на Фурие с помощта на SSE набор от инструкции.

Анализ на изчислителната специфика и възможност за паралелна обработка на информацията при реализация на БФТ

Дискретната Фурие Трансформация (ДФТ) се реализира чрез N умножения и сумирания за оценка на всеки хармоник, или N^2 умножения и сумирания за целия спектър, където N е обема на извадката от изследвания сигнал, която е подложена на трансформацията. Сложността на алгоритъма по време е $O(N^2)$. При пренареждане на аритметичните операции се постига оптимизиране на алгоритъма до сложност $O(N \log_2 N)$, което води до значително повишаване на бързодействието, като този ефект е особено силно изразен при големи стойности на N. Тази оптимизирана версия е известна като Бърза Фурие Трансформация (БФТ) (C. Sidney Burrus, 2008).

Ключова роля за реализиране на БФТ играе процесът "децимация". Той се състои в разделянето на реда на Фурие на две суми, състоящи се от многочлени, групирани по общ признак - четността.

Дискретната Фурие трансформация може да се представи във вида:

$$F_k = \sum_{n=0}^{N-1} f_n e^{-\frac{j2\pi}{N}kn} \tag{1}$$

където $F = \{F_k\}$ е дискретен вектор от честоти, $f = \{f_n\}$ е дискретен вектор на сигналът за преобразуване (n=0, 1, …, N-1), 'к' е номерът на преобразувания хармоник (k=0, 1, …, N-1).

След полагане $W_N = e^{-\frac{j2\pi}{N}}$, редът на Фурие може да се представи във вида:

$$F_k = \sum_{n=0}^{N-1} f_n W_N^{kn} \tag{2}$$

Ако въведем ограничително изискване за четност на N и използваме следните субституции

$$2r = n, \quad \textit{за четни стойности}$$
$$2r+1 = n, \quad \textit{за нечетни стойности} \tag{3}$$

зависимост (1) може да се раздели на четна и нечетна съставна, в резултат на което придобива вида:

$$F_k = \sum_{r=0}^{\frac{N}{2}-1} f_{2r} W_N^{k(2r)} + \sum_{r=0}^{\frac{N}{2}-1} f_{2r+1} W_N^{k(2r+1)} \tag{4}$$

$$F_k = \sum_{r=0}^{\frac{N}{2}-1} f_{2r} W_N^{k(2r)} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} f_{2r+1} W_N^{k(2r)} \tag{5}$$

Като използваме, че

$$W_N^{2r} = W_N^{-\frac{j2\pi}{N}2r} = W_N^{-\frac{j2\pi}{\frac{N}{2}}r} = W_{\frac{N}{2}}^{r} \tag{6}$$

зависимост (5) може да се запише по следния начин:

$$F_k = \sum_{r=0}^{\frac{N}{2}-1} f_{2r} W_{\frac{N}{2}}^{kr} + W_N^k \sum_{r=0}^{\frac{N}{2}-1} f_{2r+1} W_{\frac{N}{2}}^{kr} \tag{7}$$

Анализът на последната зависимост показва, че двете съставящи я суми са практически Фурие трансформация, приложена върху N/2 точки. Също така $W_m^m = W_N^{-j2\pi} = 1$, понеже

$$W_N^{-j2\pi} = \cos(-2\pi) - j\ \sin(-2\pi) = 1 - j0 \tag{8}$$

съгласно формулата на Ойлер за комплексни числа, от което следва, че

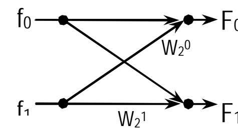
$$W_{\frac{N}{2}}^{m+\frac{N}{2}} = W_{\frac{N}{2}}^{m} \tag{9}$$

При k=N, то $W_N^k = 1$.

Ако N е число, кратно на 2^x можем да извършваме раздробяване по четност на двата полинома в (7) докато не достигнем до две точки при r=(0,1) и N=2. Техните хармоници ще са:

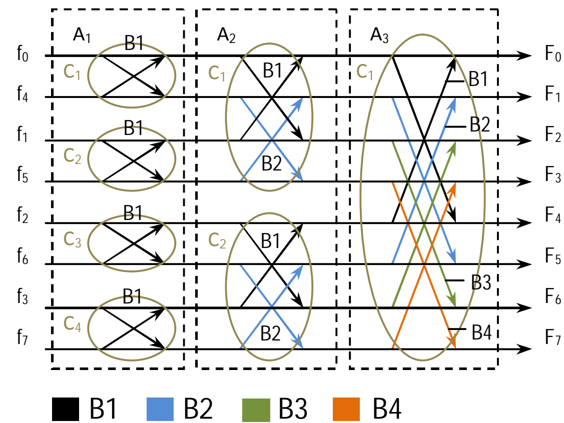
$$F_0 = f_0 W_2^0 + f_1 W_2^0 = f_0 + f_1 W_2^0$$
$$F_1 = f_0 W_2^0 + f_1 W_2^1 = f_0 + f_1 W_2^1 \tag{10}$$

Тези две уравнения могат да се представят чрез граф, наречен "пеперуда" (фиг. 1), където всяка дъга представлява операция „умножение", а всеки връх – „сумиране".



Фиг. 1. **Граф, съответстващ на уравнения** (10)

По този начин децимацията по време се представя чрез граф, изграден от подграфи-пеперуди (фиг.2).



Фиг. 2. **Графично представяне на децимация по време**

От графа, представен на фиг. 2 може да се направи извода, че в изчислителен аспект Бързото Фурие Преобразование се свежда до реализиране на 3 вложени цикъла, условно означени с A, B и C (като B е вложен в A, а C - в B). Тогава описването на всяка пеперуда 'X_{abc}' се изпълнява с наредена тройка индекси – a, b, c, принадлежащи на циклите A, B и C. Всеки граф-пеперуда се състои от 2 подграфа, като поредността на входящите им елементи се различава с 1 за итерация A_1, с 2 за A_2, с 4 за A_3 и с 2^{n-1} за A_n. Бързото Фурие преобразование се извършва на етапи, отговарящи на итерациите на цикъл A (A_1, A_2, A_3). Всеки етап се състои от пресмятане на равен брой граф-пеперуди, което дава възможност за паралелна обработка в рамките на този етап. По този начин бързодействието на БФТ ще се повиши многократно.

Особености на Интел SSE-инструкциите

Архитектурата Интел притежава мултимедийно разширение на множеството си от инструкции, като всяка от тях се изпълнява върху набор еднотипни данни (SIMD). По този начин се намалява броя итерации при трансформиране на данните чрез цикъла, в резултат на което времето за изпълнение на повтарящи се операции върху елементите от масива може да бъде редуцирано няколко пъти. Също така SIMD-инструкциите на Интел се изпълняват за по-кратко време спрямо техния пред-шественик – модулът за пресмятане на стойности с плаваща запетая (x87 FPU).

Конкретната разработка е базирана на SSE3 инструкционно множество. SSE3 инструкциите боравят с 8 броя 128-битови SIMD регистри с общо предназначение (xmm0 ÷ xmm7). Всеки от тях има възможността да съдържа 4 стойности с плаваща запетая, образувайки

пакет от данни. SSE3 инструкция, изпълнена върху 2 SIMD-регистъра е аналогична на изпълнение на операцията поотделно за всяка двойка стойности с общ индекс в операндите. Това е изобразено на фиг. 3.

	float1	float2	float3	float4
xmm-регистър1	A1	A2	A3	A4
	*	*	*	*
xmm-регистър2	B1	B2	B3	B4
	=	=	=	=
xmm-регистър3	A1*B1	A2*B2	A3*B3	A4*B4

Фиг. 3. **Изпълнение на операцията** "mulps xmm1, xmm2"

Поддържат се и инструкции за пренареждане съдържанието на пакета от данни в регистър (data shuffling). Те са полезни за реализацията на изчислителната процедура.

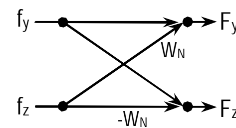
SSE3 инструкциите боравят с 1 или 2 операнда, като единият е допустимо да бъде заменен с адрес от паметта. Поради спецификата на архитектурата и с цел увеличаване на бързодействието на изчислителната операция е желателно адресите от паметта да бъдат кратни на размерността на xmm-регистрите, а именно подравнени на 16 байта.

Реализация на Бързо Фурие Преобразуване чрез SSE-инструкции

Ако входовете на всеки граф пеперуда означим с f_y и f_z, а изходите му с F_y и F, (фиг. 4), то графът може да се опише чрез следната система уравнения:

$$\begin{cases} F_y = f_y + Wf_z \\ F_z = f_y + Wf_z \end{cases} \tag{11}$$

Това означава, че за да се приведе в паралелно изчисление, един граф-пеперуда се нуждае от SIMD-регистър с размерност 2 елемента.



Фиг. 4. **Граф-пеперуда**

Ако боравим с данни с плаваща запетая, притежаващи размерност 4 байта, в един SIMD-регистър ще могат да се пакетират 4 стойности, което ни позволява паралелното изчисление на 2 граф-пеперуди. Това ще доведе до намаляване броя итерации на цикъл 'C' наполовина.

Броят на граф-пеперудите за един етап е винаги кратен на 2, тъй като той е равен на половината от обема на извадката, а тя от своя страна е кратна на 2^x по дефиниция. Това гарантира, че производителността на SIMD-инструкциите ще бъде максимизирана и конвеерната обработка ще е винаги уплътнена. Кодът, реализиращ изчислителната процедура е представен по-долу и обхваща всички итерации от цикъл 'A', с изключение на последната.

Тъй като входните и изходни параметри на преобразованието (f_i, F_i) са комплексни величини, те се представят чрез реална и имагинерна част, записани в два поредни елемента на входния масив 'f', като елементите с четни индекси съдържат реалните компоненти, а нечетните - имагинерните.

Циклите се инициализират с началния адрес на масива, в който се намират входните данни. Резултатите се записват на същите адреси. Това оптимизира допълнително изчислителните операции, свързани с адресирането, като води до увеличаване на бързодействието на всяка итерация. Също така цикълът е с фиксирана стъпка, съответстваща на броя байтове, отделяни за съхраняване на всяка стойност, представяна във формат с плаваща запетая, умножен двукратно с разликата между индексите на двете входни стойности на графа-пеперуда. Това силно специализира приложението на кода за конкретния тип процесорна платформа, но за сметка на значително увеличаване на скоростта на изпълнение на програмата.

```
          mov       edx,      [n]
          shl       edx,      1                     //n = 2*n
          mov       [n2],     edx
          shl       edx,      1                     //n = 4*n because var size is 4B
          add       edx,      vec                   //n_vec = n + (int)vec;
          mov       [n_vec],  edx
          mov       esi,      8                     //mov esi, [mmax] //mmax is esi
          mov       edi,      8                     //mov edi, [mmax] //istep is edi
loop_A:                                             //-----------------------------------------
          shl       edi,      1                     //istep = 2 * mmax
                                                    //mmax_vec = mmax + (int)vec;
          mov       edx,      esi                   //edx is temporal reg
          add       edx,      vec
          mov       [mmax_vec],edx

          // theta = -8.0f * pi / (float)mmax;
          // wtemp = sin(0.5*theta);
          // wp[0] = -2.0 * wtemp * wtemp;          //wpr = cos(theta) - 1
          // wp[1] = sin(theta);   //wpi
          movss     xmm0,     [pi8]                 //xmm1 = -8*pi
          CVTSI2SS  xmm1,     esi                   //convert mmax to float and save it in xmm1
          divss     xmm0,     xmm1                  //xmm0 = -8*pi/max (theta)   !!! possible division by 0!!!Add exception here
          pshufd    xmm0,     xmm0, 0h              //xmm0 (theta, theta, theta, theta) copy theta to all 4 floats
          movlps    xmm2,     [mul_theta]           //xmm2 (0.5, 1.0), x, x
```

```
            mulps    xmm0,    xmm2                    //xmm0 (0.5*theta, theta), x, x
            call     sin_ps                           //4 float sin functions (in rad): xmm0 = sin(xmm0)
                                                      //xmm0 (sin(0.5*theta), sin(theta), x, x)
            movlps   xmm1,    [mul_theta]             //xmm1 (0.5, 1, x, x)
            movss    xmm1,    xmm0                    //xmm1 (sin(0.5*theta), 1, x, x)
            mulps    xmm0,    xmm1                    //xmm0 (sin(0.5*theta)^2, sin(theta), x, x)
            mulss    xmm0,    [two]                   //xmm0 (-2*sin(0.5*theta)^2, sin(theta), x, x)
            movlps   [wp],    xmm0                    //move xmm0 to (wp[0], wp[1])

            //initialize (wr, wi) with (1, 0)
            //xmm6 contains current (wr, wi) values
            movlps   xmm6,    [cw]                              //xmm6 (1,0,x,x)
            mov      eax,     [vec]                             //counter 'm' is eax
loop_B:     mov      ebx,     eax                               //counter 'i' is ebx
                     mov      ecx,     ebx                      //ecx is i2
                     sub      ecx,     edi
loop_C:              add      ecx,     edi                      //add istep
                              add      ecx,     edi             //add istep

                              //create vector (wr,-wi, wr, wi)
                              movaps   xmm0,    xmm6            //xmm0 (wr, wi, x, x)
                              pshufd   xmm0,    xmm0,44h        //xmm0 (wr, wi, wr, wi) 44h = Lo %01 00 01 00 Hi (order: 00 01 10 11)
                              mulps    xmm0,    [signs]         //xmm0 (wr,-wi, wr, wi)

                              //start butterfly calculation
                              movlps   xmm1,    [ebx]           //xmm1 (v1[i], c1[i], x, x)
                              movlps   xmm2,    [ebx + esi]     //xmm2 (v1[j], c1[j], x, x)

                              movhps   xmm1,    [ecx]           //xmm1 (v1[i], c1[i],    v2[i], c2[i])
                              movhps   xmm2,    [ecx + esi]     //xmm2 (v1[j], c1[j],    v2[j], c2[j])

                              pshufd   xmm3,    xmm2,14h        //xmm3 (v1[j], c1[j], c1[j], v1[j]) 14h = %00 01 01 00
                              pshufd   xmm4,    xmm2,0BEh       //xmm4 (v2[j], c2[j], c2[j], v2[j]) 14h = %10 11 11 10

                              //calculate (tempr, tempi)
                              mulps    xmm3,    xmm0            //xmm3 (v1[j]*wr, c1[j]*wi, c1[j]*wr, v1[j]*wi)
                              mulps    xmm4,    xmm0            //xmm4 (v2[j]*wr, c2[j]*wi, c2[j]*wr, v2[j]*wi)
                              haddps   xmm3,    xmm4            //xmm3(v1[j]*wr+c1[j]*wi, c1[j]*wr+v1[j]*wi, v2[j]*wr+c2[j]*wi,
                                                                //  c2[j]*wr+v2[j]*wi) , (tempr1, tempi1, tempr2, tempi2)
                              //here xxm4 is free
                              //calculate new values of v[i], c[i]
                              //v[i] = v[i] + tempr;
                              //c[i] = c[i] + tempi;
                              movaps   xmm2,    xmm1            //xmm2 (v1[i], c1[i],    v2[i], c2[i])
                              addps    xmm2,    xmm3            //xmm2 (v1[i]+tempr1, c1[i]+tempi1, v2[i]+tempr2, c2[i]+tempi2)

                              movlps   [ebx],   xmm2            //store new calculated values of (v1[i], c1[i])
                              movhps   [ecx],   xmm2            //store new calculated values of (v2[i], c2[i])

                              //calculate new values of v[j], c[j]
                              //v[j] = v[i] - tempr;
                              //C[j] = C[i] - tempi;
                              subps    xmm1,    xmm3            //xmm1(v1[i]-v1[j]*wr+c1[j]*wi, c1[i]-c1[j]*wr+v1[j]*wi,
                                                                //  v2[i]-v2[j]*wr+c2[j]*wi, c2[i]-c2[j]*wr+v2[j]*wi)
                              movlps   [ebx + esi],  xmm1       //store new calculated values of (v1[j], c1[j])
                              movhps   [ecx + esi],  xmm1       //store new calculated values of (v2[j], c2[j])

                     //loop_C end condition
                     add      ebx,     edi                      //step
                     add      ebx,     edi                      //step2
                     cmp      ebx,     [n_vec]                  //end condition
                     jb       loop_C                            //Jump If less

                     //calculate new normal vector for CORDIC algorithm
                     //wr = (wtemp = wr) * wpr - wi * wpi + wr;
                     //wi = wi * wpr + wtemp * wpi + wi;

                     //xmm0 is (wr,-wi, wr, wi)
                     movlps   xmm4,    [wp]                     //xmm4 (wpr, wpi, x, x)
                     pshufd   xmm4,    xmm4,    14h             //xmm4 (wpr, wpi, wpi, wpr) 14h = Lo %00 01 01 00 Hi (order: 00 01 10 11)
                     mulps    xmm4,    xmm0
                     haddps   xmm4,    xmm4
                     addps    xmm6,    xmm4

            //loop_B end condition
            add      eax,     8                                 //step = 2*4B
            cmp      eax,     [mmax_vec]                        //end condition
            jb       loop_B                                     //Jump If less
            //mmax = istep
            mov      esi,     edi                               //mmax is esi //or shl esi, 1
            cmp      esi,     [n2]                              //if(mmax < n/2) loop again
            jb       loop_A
```

При последната итерация на цикъл 'A' броят на итерациите на цикъл 'C' е единица, което довежда до отстраняването му. За да се запази максималната производителност чрез едновременната работа върху две граф-пеперуди се паралелизират две съседни итерации от цикъл 'B'. Крайният брой итерации в цикъл 'B' е равен на половината от броя входове и също е кратен на 2, което гарантира плътността на конвеерната обработка и паралелизма на последната итерация.

Други механизми за повишаване на бързодействието, намерили приложение при конкретната реализация

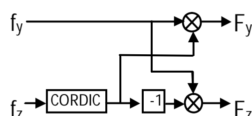
CORDIC-алгоритъм

Пресмятането на всяко тегло 'W' се изразява чрез косинусова трансформация. Тя изисква голяма изчислителна мощ, поради което е ефективно да бъде заменена с CORDIC(COordinate Rotation DIgital Computer)-алгоритъм. Той е опростен и ефикасен метод за пресмятане на хиперболични и тригонометрични функции (Volder Jack E., 1959). Приложим е за намирането на косинус функция при ъгъл, променящ се монотонно с константна стъпка, какъвто е случаят в конкретното приложение.

При умножение на два вектора V_1 и V_2 получаваме трети V_3, изразен чрез системата уравнения

$$\begin{aligned} |V_3| &= |V_1|.|V_2| \\ \arg(V_3) &= \arg(V_1) + \arg(V_2) \end{aligned} \quad (12)$$

От тук следва, че ако единият вектор (V_1) има дължина единица и фазов ъгъл δ, то произведението ще е равно на втория вектор (V_2), ротиран с фазов ъгъл δ (David E. Joyce, 1999). При прилагането на този метод за всяка итерация на цикъл 'A' по веднъж се изчислява аргумента на единичния вектор, след което се прилага CORDIC-алгоритъма за всички итерации на програмните цикли 'B' и 'C'. Участието му в графа-пеперуда е изобразено на фиг.5.



Фиг. 5. Блокова схема на граф-пеперуда с CORDIC-алгоритъм

Паралелно опростено изчисление на тригонометричните функции

Пресмятането на аргумента е базирано на паралелни синус трансформации, реализирани чрез библиотека за опростено пресмятане на тригонометрични функции. Тя допринася за по-бързото изчисляване на тригонометричните операции и понеже се изпълнява $\log_2 N$ пъти оказва съществено влияние на бързодействието при големи размерности за N.

Регистрова оптимизация

Поради ограничения брой регистри с общо предназначение (GPR), голяма част от загубите на времеви ресурс при изчисления се дължат на прехвърлянето на стойности между регистрите и паметта. С цел намаляване на този ефект е предвиден по един регистър за всяка променлива, необходима за реализацията на Бързото Фурие Преобразование. За целта преди извикването на функцията за реализация на трансформацията се запазва съдържанието на използваните регистри в стека, и след нейното изпълнение те се възстановяват. Карта на използваните регистри е представена в таблица 1.

Таблица 1.

x86 рег.	Предназначение
eax	итератор за брояч 'B'
ebx	итератор за брояч 'C', съответстващ на първият вход на графа-пеперуда
ecx	отместване спрямо итератор 'C', съответстващ на втория вход на графа-пеперуда
edx	променлива с общо предназначение
esi	стъпка между входовете на графа-пеперуда за съответната итерация на цикъл 'A'
edi	стъпка между входовете на графа-пеперуда за следващата итерация на цикъл 'A'

Началните и крайни стойности на броячите 'A', 'B' и 'C' са предварително пресметнати за да се избегне последващо сумиране на всеки итератор с началния адрес на масива на входните и изходни стойности на преобразованието. Техните стойности се съхраняват в паметта в променливи, описани в таблица 2.

Таблица 2.

Идентификатор на променлива	Описание
N	брой входни и изходни стойности на трансформацията
n2	брой входни и изходни стойности на трансформацията, умножени по две
n_vec	крайна стойност на итератора за брояч 'C' – броят на входните стойности на трансформацията, сумирани с началния адрес на техния масив
mmax_vec	крайна стойност на итератора за брояч 'B' – сума от стъпката между входовете на графа-пеперуда с началния адрес на вектора на входовете

Всички операции умножение и деление с операнд кратен на 2 са заменени с по-бързия им аналог – операция за побитово преместване наляво/надясно.

Особености при SSE-оптимизацията

За по-бързото зареждане на SSE-регистрите със стойности от паметта е необходимо тя да бъде подравнена на 16 байта (големината на SSE регистър). За целта статичните данни в MSVC++ се декларират префиксно с командата „_declspec(align(16))". Динамичните данни се заделят подравнени чрез "_aligned_malloc". Това позволява използването на по-бързата инструкция „movaps" за зареждане на регистър с 4 пакетирани стойности с плаваща запетая от паметта, за сметка на по-бавния й аналог „movups", прилагащ се при неподравнени данни.

Ограничения при използване на предложеното решение

Понеже SIMD-регистрите не подлежат на запис в стека, то прекъсването на функцията от друга, използваща тези регистри е недопустимо. Това не позволява повторното

извикване на функцията преди нейното първоначално завършване (non-reentrant function).

Поради спецификата на асемблерния код и SSE-инструкциите представената функция е непреносима, нейното изпълнение е възможно само на x86 P3 съвместими архитектури.

Резултати от проведени изследвания

Използвана е тестовата платформа AMD Athlon 64 Dual Core 4200+. Работната честота е 2.21 GHz.

С цел сравнителна оценка на бързодействието на предложеното решение, задачата е реализирана и на VC++ под MSVS2008 без явното използване на вградените оператори за работа със SIMD-инструкциите (intrinsic).

Двата кода са компилирани при зададени параметри: максимална скорост /O2, боравене със SIMD-инструкции /Oi, оптимално бързодействие за сметка на размера на програмата /Ot и /arch:SSE2, използване на бърз модел на пресмятане на числа с плаваща запетая /fp:fast.

Резултатите от извършените изследвания на бързодействието при различни обеми на извадката са представени в таблица 3 и графично интерпретирани на фигура 6.

Анализът им показва, че във всички случаи предложената платформено-оптимизирана реализация на БФТ е с по-високо бързодействие в сравнение със стандартния подход за реализиране на това преобразуване. Съществено е, че тази тенденция е по-силно изразена при потежките задачи, боравещи с големи по обем извадки от изследвания процес.

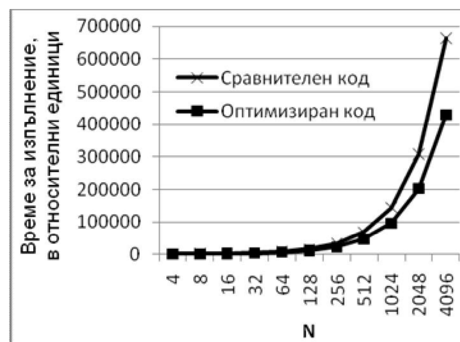
Заключение

Предложена е софтуерна реализация на БФТ, при която чрез използване на особеностите на архитектурата на процесорите от фамилия Intel Pentium и с използване на SSE инструкции е постигнато повишаване на бързодействието с около 30% в сравнение с класическия подход за реализация на тази трансформация.

Препоръчана за публикуване от катедра „Автоматизация на минното производство", МЕМФ

Таблица 3.

Обем на извадката	Скорост на изпълнение на сравнителната програма (в процесорно време)	Скорост на изпълнение на изследваната програма (в процесорно време)	бързодействие, %
4	1043	863	17.2
8	1508	1219	19.2
16	2436	1786	26.6
32	4460	3188	28.5
64	8539	6115	28.4
128	16320	11712	28.2
256	32778	22702	30.7
512	68232	46754	31.5
1024	144331	96465	33.2
2048	308185	202734	34.2
4096	664560	430148	35.3



Фиг. 6. Съпоставка на времената за изпълнение на оптимизирания и сравнителния код

Литература

http://en.wikipedia.org/wiki/CORDIC
http://en.wikipedia.org/wiki/Fast_Fourier_transform
http://www.relisoft.com/science/Physics/fft.html
Joyce David E., 1999, http://www.clarku.edu/~djoyce/complex/mult.html
ssemath, http://gruntthepeon.free.fr/ssemath/
Leiterman James, 2005, 32/64-bit 80x86 assembly language architecture, Wordware Publishing, Inc., p.343
Sidney Burrus C., online book of Fast Fourier Transforms, 2010, http://cnx.org/content/col10550/latest/
Volder Jack E., 1959, The CORDIC Trigonometric Computing Technique, *IRE Transactions on Electronic Computers*, pp.330-334,