Projekt JA
Fast Fourier Transform

Hubert Przegendza
AEI, inf, sem V, grupa 3-4

Analiza zadania

Fast Fourier Transform (dalej nazywane FFT) jest algorytmem rekurencyjnym, za pomocą którego można policzyć natężenia składowych częstotliwości danego dźwięku. Algorytm ten od dawna jest używany do analizy dźwięku albo estetycznej wizualizacji muzyki. W tym projekcie postanowiłem spróbować go zaimplementować w języku C++ oraz w języku ASM z wykorzystaniem funkcji wektorowych i porównać ich efektywność.

Matematyka
Dyskretna Transformata Fouriera

$$X_k = \sum_{n=0}^{N-1} x_n * e^{\frac{-i\,2\pi kn}{N}} \quad ,$$

gdzie $k = 0, \ldots, N-1$, a $e^{\frac{-i\,2\pi n}{N}}$ jest N-tym „pierwiastkiem z jedynki” (ang. primitive root).

Niestety złożoność obliczeniowa tej definicji jest $O(N^{2)})$ – N sum N elementów. Dzisiaj można znaleźć wiele różnych implementacji algorytmów wykorzystujących pewne zależności i powtarzalności do zmniejszenia złożoności do aż O(NlogN).

Przebieg wybranego przeze mnie algorytmu
Po otrzymaniu źródła dźwięku, kod wczytuje próbki dźwięku wielkości 2^{14} = 16384, z częstotliwością 44100HZ (domyślna wartość biblioteki).
Na każdej próbce wykonuję rekurencyjną funkcję **fft**, która w swoim ciele dzieli próbkę X na dwie mniejsze X_{even} i X_{odd} składające się z parzystych i nieparzystych elementów, i na każdej z nich powtarza funkcję **fft**.

Kiedy wielkość tablicy przekazanej do funkcji **fft** będzie 2, to nie będą dalej wywoływane rekurencyjnie kolejne funkcje.

Następnie w FFT na przekazanej tablicy X o wielkości N wykonuje się:
dla $k = 0, \ldots, N-1$, $X[k] = X_{even}[k] + X_{odd}[k] * e^{-2*\pi*k/N}$,
gdzie po wyjściu z tej funkcji, X będzie X_{even} lub X_{odd} dla rekurencyjnie nadrzędnej funkcji.

Po zakończeniu się funkcji **fft** wykonanej jako pierwszej, w tablicy w której była przechowywana próbka dźwięku, zawiera teraz wynik algorytmu FFT, który można wyświetlić.

Implementacja

C++

```
void fft_CPP(float* real, float* imag, int N, ThreadMom& tm)
{
	if (N <= 1) return; //algorythm condition to finish recurrence

	//must copy given array into two smaller ones - even and odd
	float* realEven = new float[N / 2 + N % 2];
	float* imagEven = new float[N / 2 + N % 2];
	float* realOdd = new float[N / 2];
	float* imagOdd = new float[N / 2];
```

```
//copying arrays
int i = 0;
int k = 0;
for (; k < N / 2; )
{
	realEven[k] = real[i];
	imagEven[k] = imag[i];
	i++;
	realOdd[k] = real[i];
	imagOdd[k] = imag[i];
	i++; k++;
}
if (i < N)
{
	realEven[k] = real[i];
	imagEven[k] = imag[i];
}

//ask for a free thread, to do the calculation even or odd part of the sample
Response r1 = tm.askForHelp_float({ realEven, imagEven, N / 2 });
Response r2 = tm.askForHelp_float({ realOdd, imagOdd, N / 2 });

if (r1.finish == nullptr && r2.finish == nullptr)
{ //no free thread - calculate in this one
	fft_CPP(realEven, imagEven, N / 2, tm);
	fft_CPP(realOdd, imagOdd, N / 2, tm);
}
else if (r1.finish != nullptr && r2.finish == nullptr)
{ //only one thread free - wait for it to finish after calculating the other half in
	//this one
	fft_CPP(realOdd, imagOdd, N / 2, tm);
	r1.finish->wait();
	r1.accept->post();
}
else if (r1.finish == nullptr && r2.finish != nullptr)
{ //only one thread free - wait for it to finish after calculating the other half in
	//this one
	fft_CPP(realEven, imagEven, N / 2, tm);
	r2.finish->wait();
	r2.accept->post();
}
else {//two threads free - wait for them to finish
	r1.finish->wait();
	r1.accept->post();
	r2.finish->wait();
	r2.accept->post();
}

for (int k = 0; k < N / 2; k++)
{ //main calculation X[k] = X_even [k] + X_odd[k] * e^(-2k*pi/n)
	float polarReal = sineASM_float(negPi2f * k / N + piHalff);
	float polarImag = sineASM_float(negPi2f * k / N);

	float realTemp = polarReal * realOdd[k] - polarImag * imagOdd[k];
	float imagTemp = polarReal * imagOdd[k] + polarImag * realOdd[k];

	//x[k] = even[k] + t;
	real[k] = realEven[k] + realTemp;
	imag[k] = imagEven[k] + imagTemp;
	//x[k + N / 2] = even[k] - t;
	real[k + N / 2] = realEven[k] - realTemp;
	imag[k + N / 2] = imagEven[k] - imagTemp;
}
delete[] realEven;
```

```
        delete[] imagEven;
        delete[] realOdd;
        delete[] imagOdd;
}
```

Response – struktura składająca się z 2 semafor. Pierwsza dostarcza sygnał, o skończeniu kalkulacji, a drugiej należy użyć do zasygnalizowania odebrania wyniku kalkulacji.

Próbka dźwięku to tablica liczb złożonych, a chcąc uprościć kod dla łatwiejszego porównania z implementacją w ASM, zmuszony byłem wszystkie operacje robić na dwóch tablicach składających się z części realnych i urojonych oryginalnej tablicy.

ASM

```
;math explanation
;r - r values array
;i - imaginary values array
;rO - odd elements of r array
;rE - even elements of r array
;iO - odd elements of i array
;iE - even elements of i array

;r[k] = rE[k] + rT
;i[k] = iE[k] + iT
;r[k+N/2] = rE[k] - rT
;i[k+N/2] = iE[k] - iT
;pR = sin(-2 * PI * k / N + PI / 2) ;r part of a polar complex number
;pI = sin(-2 * PI * k / N)          ;imaginary part of a polar complex number
;rT = pR * rO[k] - pI * iO[k]       ;r part of temporary complex number
;rT = a - b,
       ;a = pR * rO[k]
       ;b = pI * iO[k]
;iT = pR * iO[k] - pI * rO[k]       ;imaginary part of temporary complex number
;iT = c + d,
       ;c = pR * iO[k]
       ;d = pI * rO[k]

funcASM_float_v proc
;int N,              rcx                 -> rcx
;float* r,    rdx                 -> rdx
;float* i,    r8                  -> r8
;float* rO,   r9                  -> r9
;float* iO,   rsp + 40      -> r10
;float* rE, rsp + 48 -> r11
;float* iE    rsp + 56      -> r12
;float* r + N/2                   -> r13
;float* i + N/2                   -> r14

       ;saving non-volatile registers
       push rbx
       push r12
       push r13
       push r14

       ;prepare pointers
       mov r10, qword ptr[rsp + 40 + 32] ;iO
       mov r11, qword ptr[rsp + 48 + 32] ;rE
       mov r12, qword ptr[rsp + 56 + 32] ;iE
       mov r13, rcx
       shl r13, 1 ;N/2 * 4 = N*2 = N<<1
       mov r14, r13
       add r13, rdx ;r + N/2
```

```
        add r14, r8  ;i + N/2

        xor rbx, rbx

LoopLabel:
        mov rax, rcx ;N -> rcx
        shr rax, 1   ;N/2
        cmp rbx, rax ; while(rbx != N/2) { [...] rbx++ }
        je fin

        ;float pR = sin(-2 * PI * k / N + PI / 2);
        ;float pI = sin(-2 * PI * k / N);
        cvtsi2ss xmm0, rbx ;k
        cvtsi2ss xmm1, rcx ;N
        divss xmm0, xmm1 ;=k/N
        mulss xmm0, negPi2f ;*=-2*PI
        movss xmm4, xmm0
        addss xmm0, piHalff
        ;xmm0 = -2 * PI * k / N + PI / 2
        call sineASM_float ;pR -> xmm0
        movss xmm5, xmm0 ;pR -> xmm5

        movss xmm0,xmm4
        call sineASM_float ; pI
        movss xmm2, xmm5
        movss xmm3, xmm0

        ;xmm2 - pR
        ;xmm3 - pI
        ;float* rO,   r9            -> r9
        ;float* iO,   rsp + 40      -> r10

        movd xmm0, dword ptr[r9] ;rO
        movd xmm1, dword ptr[r10];iO
        movlhps xmm0, xmm1
        ;xmm0 = _,iO,_,rO
        shufps xmm0, xmm0, 00101000b
        ;xmm0 = rO,iO,iO,rO
        shufps xmm3, xmm2, 00000000b
        ;xmm3 - pR,pR,pI,pI

        mulps xmm0,xmm3
        ;xmm0 - a,c,b,d
        movaps xmm1, xmm0
        shufps xmm1, xmm1, 11101110b
        ;xmm1 - a,c,a,c
        shufps xmm0, xmm0, 01000100b
        ;xmm0 - b,d,b,d
        pcmpeqw xmm2,xmm2
        pslld xmm2,25
        psrld xmm2,2
        ;xmm2 - 1,1,1,1
        pcmpeqw xmm3,xmm3
        pslld xmm3,31
        orps xmm3, xmm2
        ;xmm3 - -1,-1,-1,-1
        shufps xmm3,xmm2,00000000b
        ;xmm3 -  1, 1,-1,-1
        mulps xmm1, xmm3
        ;xmm1 - a,c,-a,-c

        shufps xmm3, xmm3, 00111100b
        ;xmm3 - -1, 1, 1,-1
        mulps xmm0, xmm3
```

```
        ;xmm0 - -b, d, b,-d
        ;xmm1 -  a, c,-a,-c
        addps xmm0,xmm1
        ;xmm0 -  (a-b), (c+d), (b-a), (-c-d)
        ;xmm0 -  rT, iT,-rT,-iT

        ;r11 - float* rE,
        ;r12 - float* iE,

        movd xmm1, dword ptr[r11] ;rE
        movd xmm2, dword ptr[r12] ;iE
        movlhps xmm1, xmm2 ;_,iE,_,rE
        shufps xmm1, xmm1, 00100010b
        ;xmm1 - rE,iE,rE,iE

        addps xmm0, xmm1
        ;xmm0 - r[k],i[k],r[k+N/2],i[k+N/2]

        ;rdx - float* r,
        ;r8  - float* i,
        ;r13 - float* r + N/2
        ;r13 - float* r + N/2
        ;r14 - float* i + N/2

        ;i[k+N/2]
        movd dword ptr[r14], xmm0
        shufps xmm0, xmm0, 11111001b
        ;r[k+N/2]
        movd dword ptr[r13], xmm0
        shufps xmm0, xmm0, 11111001b
        ;i[k]
        movd dword ptr[r8], xmm0
        shufps xmm0, xmm0, 11111001b
        ;r[k]
        movd dword ptr[rdx], xmm0

        inc rbx
        ;move all of the pointers to their next element
        add rdx, 4
        add r8 , 4
        add r9 , 4
        add r10, 4
        add r11, 4
        add r12, 4
        add r13, 4
        add r14, 4

        jmp LoopLabel

fin:
        ;recover non volatile registers
        pop r14
        pop r13
        pop r12
        pop rbx
ret
funcASM_float_v endp
```

Instrukcje wektorowe

- `mulps xmm0,xmm3`– mnożenie zapakowanych float, wynik do `xmm0`
- `movaps xmm1, xmm0` – przenieś zapakowane float z `xmm0` do `xmm1`
- `shufps xmm1, xmm1, 11101110b` – „potasuj” zapakowane float względem klucza

- `pcmpeqw xmm2,xmm2` – porównaj zapakowane float i zapisz wynik w `xmm2` (1 – są równe, 0 – nie są)
- `pslld xmm2,25` – przesuń logicznie w lewo zapakowane float
- `orps xmm3, xmm2` – logiczny OR na zapakowanych wartościach float
- `addps xmm0,xmm1` – dodaj zapakowane float z `xmm1` do `xmm0`

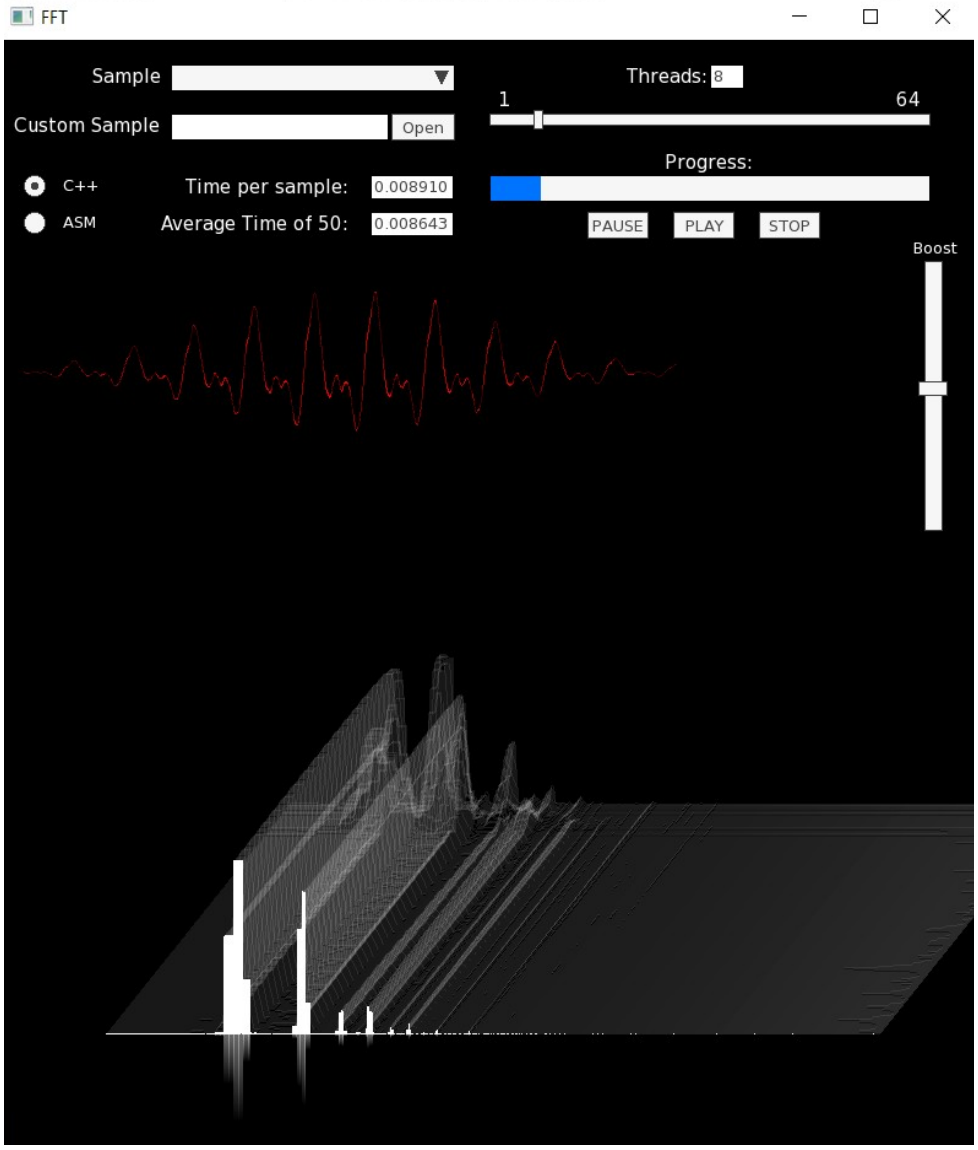
Metoda prośby o dodatkowy wątek (ThreadMom::)

```
Response askForHelp_float(Data_float data)
{
	for (int i = 0; i < amountOfThreads; i++)
	{
		//check if thread[i] is free
		if (freeThreads[i].try_wait())
		{
			//fill container of thread[i]
			threadData_float[i] = data;
			//start thread
			threadGo[i].post();
			//return handles to communicate with the thread
			//(check if finished, accept)
			return { &threadFin[i], &threadAcceptFin[i] };
		}
	}
	//no thread is free
	return Response{ nullptr, nullptr };
}
```

Metoda pojedynczego wątku (ThreadMom::)

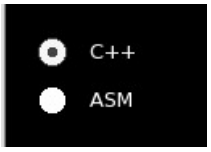
```
void threadMethod(int id)
{
	while (1)
	{
		//wait to be summoned
		threadGo[id].wait();

		//check if the program is not shutting down
		if (dying == true) {
			return;
		}

		//call fft method chosen by the user (C++ vs ASM)
		fft_choice_method(	threadData_float[id].real,
					threadData_float[id].imag,
					threadData_float[id].size,
					*this);

		//signal calculation completion
		threadFin[id].post();

		//wait for accept
		threadAcceptFin[id].wait();

		//signal thread is free for next calculation
		freeThreads[id].post();
	}
}
```

Opis GUI programu



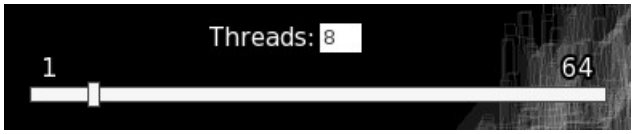
,

gdzie



Wybór biblioteki

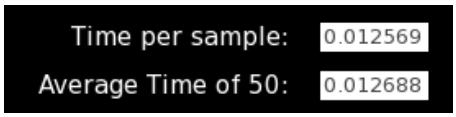
Wybór ścieżki dźwiękowej



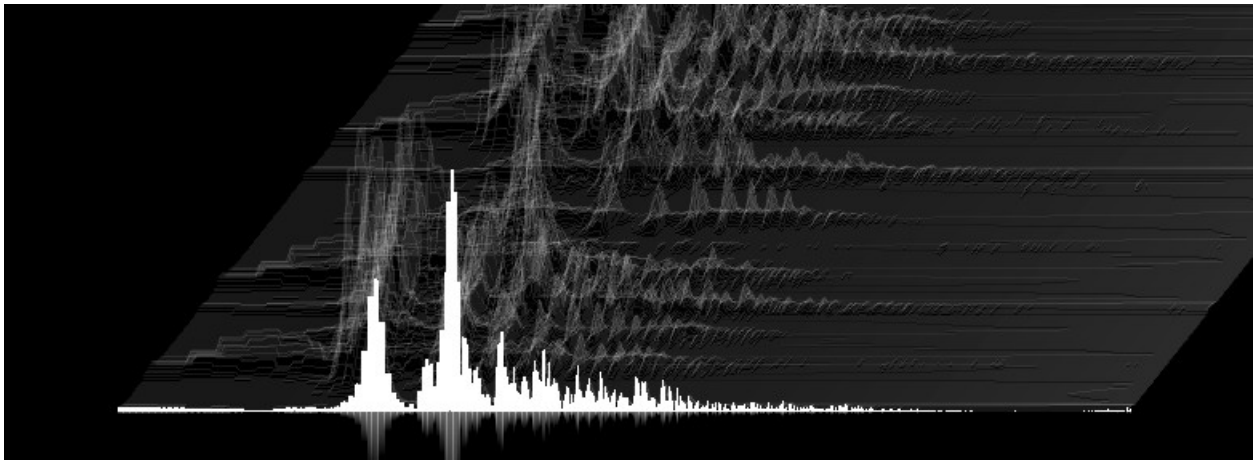
Wybór ilości wątków

Wzmocnienie wizualizacji (przydatne dla zbyt głośnych lub cichych ścieżek)

Wskaźnik postępu oraz przyciski do odpowiednio zatrzymania, wznowienia oraz przerwania odtwarzania



Czas przeliczenia pojedynczej próbki oraz średnia czasu przeliczenia ostatnich 50 próbek (w sekundach)



Wizualizacja dźwięku – każda kolejna kolumna odpowiada kolejnej częstotliwości. Im wyższa kolumna, tym większe natężenie odpowiadającej jej częstotliwości.

W tle historia poprzednich wykresów.

Porównanie czasu

Testy przeprowadzono na maszynie obsługującej 8 logicznych procesorów.

Wielkość pliku nie wpływa na czas wykonania algorytmu, więc porównanie wystarczy wykonać na tylko jednym pliku.

Czas wykonania algorytmu [s]							
	Ilość wątków						
Biblioteka	1	3	8	12	24	32	64
C++	0.0119	0.0113	0.0136	0.0152	0.0175	0.0187	0.0206
ASM	0.0149	0.0125	0.0146	0.0152	0.0180	0.0189	0.0214

Jak widać, optymalizacja kodu w C++ przez kompilator okazała się być skuteczniejsza, niż mój kod wykorzystujący operacje wektorowe napisany w ASM.

Najwyższa prędkość osiągana jest dla 3 wątków, mimo że maszyna obsługuje aż 8 procesorów logicznych.

Wnioski

Operacje wektorowe są szybkie, ale wymagają pakowania danych, co dla fragmentu kodu, który wybrałem kosztuje więcej czasu niż zyskuje.

Napisanie większego fragmentu kodu w .asm pozwoliłoby na większą optymalizację, co mogłoby poprawić wydajność. Można również skorzystać z większych rejestrów (YMM zamiast XMM). Pozwoliłoby to na operowaniu na większej ilości danych naraz, co również powinno skrócić czas wykonania.