FFT algorithm optimization using Stream SIMD Extension instruction set

Pawel Dawidowski
Wroclaw Uniwersity of technology

This paper explain how to perform Cooley and Tukey fast Fourier transform for sequence length being power of 2 using modern Stream SIMD[1] Extension known as SSE. Basis of FFT algorithm will be shown pointing on which parts can be significantly speed up using SSE focusing on programming side rather than FFT algorithm itself. Also basic SSE instruction will be introduced.

I. FFT

The idea of calculating digital convolution is to approximate integral in (1) to (2) on finite sum over N samples of signal:

$$\int_{-\infty}^{\infty} f(t)\cdot e^{-j\omega t}dt \qquad (1)$$

$$X_k = \sum_{n=0}^{N-1} x_n \cdot e^{\frac{-2\pi j}{N}kn} \quad k = 0,1,...,N-1 \qquad (2)$$

Point is this raw calculation need to perform N multiplication over k which takes also N values so overall complexity of algorithm is N^2. Even for relatively small numbers of samples the amount of calculation can be pretty high. Assuming N is power of 2 Cooley and Tukey has split odd and even elements of sum (2):

$$W^{kn} = e^{\frac{-2\pi j}{N}kn} \qquad (3)$$

$$X_k = \sum_{n\ even}^{N-1} x_n \cdot W^{kn} + \sum_{n\ odd}^{N-1} x_n \cdot W^{kn} \qquad (4)$$

Each of those summations is recognized as being an N/2 point DFT of the respective sequence because:

$$W^2 = e^{\frac{-2\pi j}{N}2} = e^{\frac{-2\pi j}{\frac{N}{2}}} \qquad (5)$$

So if we calculate DFT for odd and even indexed k in (2) we can calculate overall DFT combining (4) together:

$$\overline{X}(k) = \overline{X}_{even}(k) + W^k\overline{X}_{odd}(k) \qquad (6)$$

Using this approach we will need only N · log$_2$N complex multiplication to perform, because we need exactly log$_2$N of 2 times smaller DFT to calculate.
For example if N = 1024:

$$\begin{aligned} &DFT: \quad multiplication\, need = \mathrm{N}^2 = 1048576 \\ &FFT: \quad multiplication\, need = \mathrm{N}\log_2 N = 10240 \end{aligned} \qquad (7)$$

Because of this great time reduction FFT algorithm is widely used in many application which need Fourier transform.

[1] SIMD is abbreviation from Single Instruction Multiple Data. In the term of SSE instruction 4 single precision floating point or 2 double precision floating point operation are performed in parallel on one processor core.

II. Stream SIMD Extension

SSE is a mathematical coprocessor which operates on floating point numbers. It has 8 additional and independed 128-bit registers for calculation which are 4x32-bit single precision floating point value or 2x64 double precision one. The main advantage of using it is performing 4 or 2 operation in parallel, so depending on precision we need it can cut the algorithm execution time 4 or 2 times.

Basic SSE instruction set is shown in table 1, while full instruction set with descriptions and timing is available at Intel web site (Ref. [2]). All instruction which end –PS is doing 4 parallel operations at simultaneously while –SS mean only one operation.

TABLE 1
SSE BASIC INSTRUCTION SET

Instruction	Operation
MOVSS, MOVAPS, MOVUPS MOVSD, MOVAPD,MOVUPD	Copy operation: from memory. to register from register to memory from register to register
ADDPS, ADDSS, ADDPD, ADDSD	Addition
SUBPS, SUBSS, SUBPD, SUBSD	Subtraction
MULPS, MULSS, MULPD, MULSD	Multiplication
DIVPS, DIVSS, DIVPD, DIVSD	Division
SHUFPS	In register data shuffle

There are few things which can be optimized using those instructions. First thing is to rewrite complex multiplication to take advantage of parallel operation. Second is to notice, that in first step W^{kn} = -1 so no complex operation is actually necessary. In second step we have at most 4 W^{kn} values 1, j, -1 ,-j which can threaten as a special case. Since W^{kn} are kth root of one we can calculate it inside of SSE register for next steps. Only value need to load is to load complex number for k = 1 in (3). Next values for next k can be calculated by recursion (8):

$$W^{kn+1} = W^{kn}\cdot W^1 \qquad (8)$$

III. Complex multiplication on SSE

To calculate complex multiplication using SSE we can assume at the beginning that we want to perform 2 multiplications at once. With 4 floating point register each pair of 2 values can be complex real and imaginary part. For start assume we want to calculate:

$$\bar{Z}_A = a_A + jb_A$$
$$\bar{Z}_B = a_B + jb_B \qquad (9)$$
$$\bar{Z}_C = \bar{Z}_A \cdot \bar{Z}_B$$

From algebra we know:

$$\bar{Z}_C = a_A \cdot a_B - b_A \cdot b_B + j\left(a_A \cdot b_B + b_A \cdot a_B\right) \qquad (10)$$

We need 4 multiplications and 2 algebraic additions to calculate complex multiplication.

Now let's try to do this using SSE. Since we have 4 floating value in one register we can perform 2 multiplications in parallel. Let's assume we have loaded our complex numbers into XMM0 and XMM1 registers and XMM3 and XMM4 has values to change signs. Each of them contains 2 complex numbers as in the Table 2:

TABLE 2
INITIAL VALUES IN REGISTER

Register name	reg[0]	reg[1]	reg[2]	reg[3]
XMM0	a_{A1}	b_{A1}	a_{A2}	b_{A2}
XMM1	a_{B1}	b_{B1}	a_{B2}	b_{B2}
XMM3	1.0	-1.0	1.0	-1.0
XMM4	-1.0	1.0	-1.0	1.0

The program which realizes complex multiplication is:

```
1.     MOVAPS                %%XMM0,%%XMM2
2.     SHUFPS         $0xA0, %%XMM2,%%XMM2
3.     MULPS                 %%XMM1,%%XMM2
4.     SHUFPS         $0xF5, %%XMM0,%%XMM0
5.     MULPS                 %%XMM3,%%XMM0
6.     MULPS                 %%XMM1,%%XMM0
7.     SHUFPS         $0xB1  %%XMM0,%%XMM0
8.     ADDPS                 %%XMM2,%%XMM0
```

Table 3. shows how register has changed as a result of operation:

TABLE 3
REGISTER VALUES AFTER OPERATION

Line	register	reg[0]	reg[1]	reg[2]	reg[3]
1.	XMM2	a_{A1}	b_{A1}	a_{A2}	b_{A2}
2.	XMM2	a_{A1}	a_{A1}	a_{A2}	a_{A2}
3.	XMM2	$a_{A1}\ a_{B1}$	$a_{A1}\ b_{B1}$	$a_{A2}\ a_{B2}$	$a_{A2}\ b_{B2}$
4.	XMM0	b_{A1}	b_{A1}	b_{A2}	b_{A2}
5.	XMM0	b_{A1}	$-b_{A1}$	b_{A2}	$-b_{A2}$
6.	XMM0	$b_{A1}\ a_{B1}$	$-b_{A1}\ b_{B1}$	$b_{A2}\ a_{B2}$	$-b_{A2}\ b_{B2}$
7.	XMM0	$-b_{A1}\ b_{B1}$	$b_{A1}\ a_{B1}$	$-b_{A2}\ b_{B2}$	$b_{A2}\ a_{B2}$
8.	XMM0	$a_{A1}\ a_{B1} - b_{A1}\ b_{B1}$	$a_{A1}\ b_{B1} + b_{A1}\ a_{B1}$	$a_{A2}\ a_{B2} - b_{A2}\ b_{B2}$	$a_{A2}\ b_{B2} + b_{A2}\ a_{B2}$

As we can see we need only 3 parallel multiplications and one parallel addition to calculate 2 complex multiplications at once. It means we need only 1.5 multiplications not 4 and only 0.5 additions not 2 per one complex multiplication. So optimization has reduced floating point operation from 6 to 2. That's 3 times faster than without use SSE. All copy and shuffle operation are only on internal registers which is done much faster than copying values from/to memory.

IV. Memory operation from and to SSE registers

There are two main instructions to load from memory into XMM register and to store XMM register into memory: MOVUPS and MOVAPS. First one work as a normal MOV operation but it will copy 4x32bit floating point number into XMM register. MOVAPS is faster but with one constraint: all memory addresses has to be 16 bytes aligned, it means four least significant bits in address has to be equal zero. The speed difference in practical application is about 20-30% with MOVAPS in comparison to MOVUPS. Luckily for FFT calculation every step will be 16 bytes aligned since overall data length is power of 2 so only thing to do is enable 16 bytes align in compilation options.

V. Optimizing 1ST and 2ND FFT step

For 1st step W = -1 so there is no need to perform any complex calculation. For second step W = 1, j ,-1, -j. For those special cases we can write even shorter program. Only 3th and next steps will need full complex multiplication.

Multiplication by j is only shuffling real and imaginary part and change the sign properly as shown in (11) so it can be realized on 2 SSE instructions for 2 complex numbers in parallel.

$$\bar{Z} \cdot j = (a + jb)\,j = -b + ja \qquad (11)$$

The program which realize complex multiplication by j using initial values as in Table 2 except XMM1 and XMM4 are skipped:

```
1.     SHUFPS         $0xB1, %%XMM0,%%XMM0
2.     MULPS                 %%XMM4,%%XMM0
```

The point is that even if we want to swap real and imaginary part it could result in multiple memory read/write operation. Here 2 complex numbers are read from and write to memory exactly once so overall floating point operation number is 0.5 multiplications per one complex number.

VI. Summary

Well optimized FFT algorithm can calculate Fourier transform online, even for quite large samples number, while it would be impossible to use raw DFT algorithm to calculate it even on modern processors. Using SSE cuts computation time even more by taking advantage of parallel calculation and coprocessor designed especially for floating point operation. All instructions presented in this paper are available in C++ either by inline assembly code or by intrinsic functions fully described on Microsoft web page (Ref. [3]).

VII. References

[1] Leland B. Jackson "Digital Filters and Signal Processing ", 3th ed. University of Rhode Island. Kluwer Academic Publishers Boston Dordrecht London

[2] http://www.intel80386.com/simd/mmx2-doc.html

[3] http://msdn.microsoft.com/en-us/library/t467de55(VS.71).aspx