FFT C2C as 2-way R2C Transform

YangKunlun

August 10, 2013

1 Parellel Problem

The Surface Related Multiple Elimination (SRME) is a widely used algorithm in seismic processing, yet it can be simplified into the this convolution based computational model.

$$M_{a,b} = \sum_{i=-\infty}^{\infty} P_{a,i} * P_{i,b} \tag{1}$$

The algorithm performs convolution in frequency domain, and it makes a lot of forward r2c transforms to produce one result. The r2c transform is one of the critical region to the optimization of SRME.

2 SIMD Efficiency

Cost of r2c transform is half of c2c transform due to symmetry. On a 2.4G Hz i3-3110M CPU, the TPS (Transforms Per Second) of FFTW library compiled into x87 instruction set is shown in Figure 1. The numbers are consistent with the theory about computation costs.

However, if AVX instruction is enabled, things go differently as shown in Figure 2. This means AVX instruction in FFTW's r2c transform is not as well exploit as in c2c transfrom, and it gives us the possibility to use c2c transform as a 2-way r2c transform.

The SSE instruction is capable of operating on 4 float numbers a the same time, and AVX is capable of 8, so ideally, if we care only about throughput, we should implement 4-way transform for SSE and 8-way transform for AVX to better utilize the hardware's capability. But before that, let's try 2-way r2c transfrom on top of c2c transform.

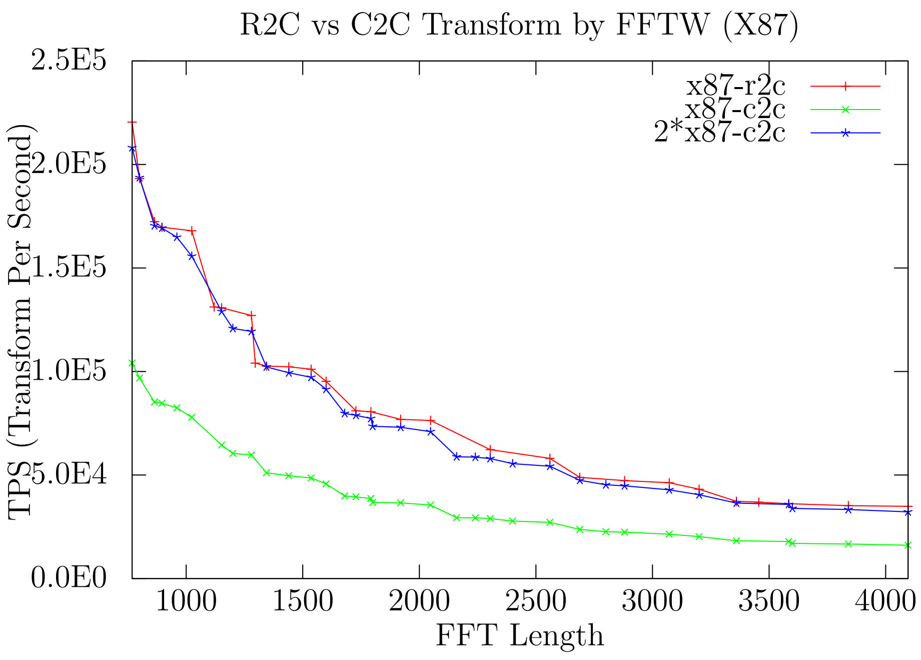

Figure 1: X87 Instruction Set for Commonly Seen FFT Length

3 2-Way R2C Transform

Assume we have two discrete real signal a_j and b_j, where $j \in [0, N)$, and the corresponding DFT are A_j and B_j. Since a_j and b_j are real signal, we know that $A_{N-j} = \bar{A}_j$, and $B_{N-j} = \bar{B}_j$, where $j = [1, N/2)$.

To use the complex algorithm to computer two real transform at one time, we can construct a complex signal $c_j = a_j + Ib_j$, and the corresponding DFT is C_j. We also know that $C_j = A_j + IB_j$. There is a special case for the DC part of the transform that $A_0 = C_0^r$, and $B_0 = C_0^i$, the upper scripts indicate real and image part.

The following equation can be used to solve other frequency component of the two real transform.

$$C_j = A_j + IB_j \tag{2}$$

$$C_{N-j} = A_{N-j} + IB_{N-j} \tag{3}$$

$$j \in [1, N/2) \tag{4}$$

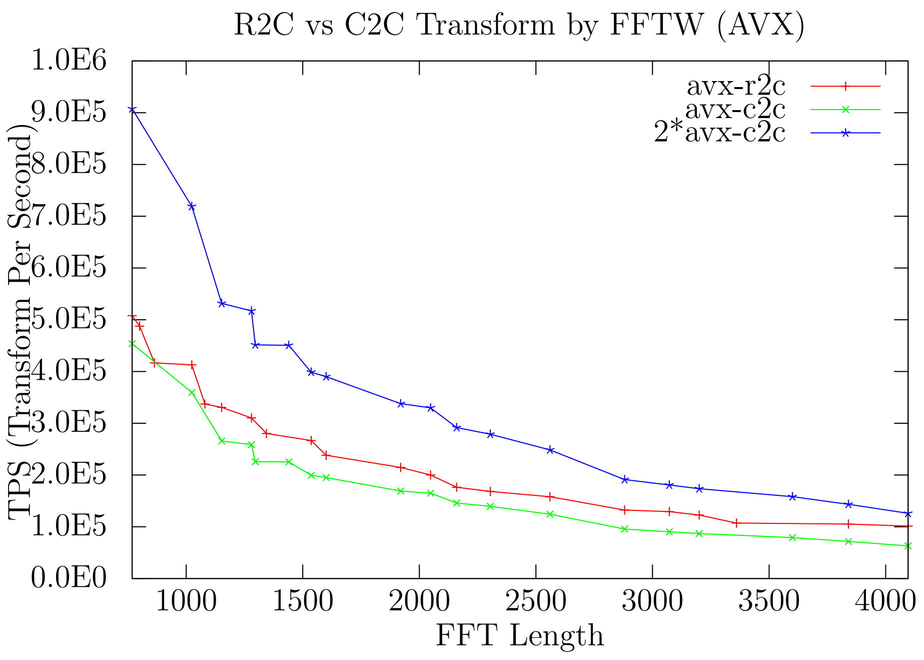

Figure 2: AVX Instruction Set for Commonly Seen FFT Length

The above equations can be further simplified and solve as

$$A_j = \frac{\bar{C}_{N-j} + C_j}{2} \tag{5}$$

$$B_j = \frac{\bar{C}_{N-j} - C_j}{2} I \tag{6}$$

4 Faster convolution

For convolution of series a_j and b_j, we need to calculate A_jB_j.

$$A_jB_j = \frac{\bar{C}^2_{N-j} - C^2_j}{4} I, j \in [1, N/2) \tag{7}$$

The following list is SSE3 intrinsic of two complex multiplications, it is more or less the same as what *icc* will generate for vectorization.

```
inline static __m128 cmul_sse3_i(__m128 a, __m128 b)
{
    __m128 c = _mm_mul_ps(_mm_moveldup_ps(a), b);
    b = _mm_shuffle_ps(b, b, _MM_SHUFFLE(2,3,0,1));
    a = _mm_mul_ps(_mm_movehdup_ps(a), b);
    return _mm_addsub_ps(c, a);
}
```

Based on the above code, convolution Equation 7 that use c2c (2-way r2c) transform can be implement as shown in the following list.

```
void two_way_conv(const complex float *C, int nr, complex float *D)
{
    int nc = nr/2+1;
    int nn = (nc%2)?(nc):(nc-1);
    D[0] += ((float*)C)[0]*((float*)C)[1]; //DC component

    for(int i=1; i<nn; i=i+2) {
        __m128 ci = _mm_loadu_ps((float*)(C+i));
        __m128 cj = _mm_mul_ps(_mm_loadu_ps((float*)(C+nr-1-i)),
                /*conjugate*/   _mm_set_ps(-1.0f, 1.0f, -1.0f, 1.0f));
        cj = _mm_shuffle_ps(cj, cj, _MM_SHUFFLE(1,0,3,2));
        __m128 pl = _mm_add_ps(cj, ci);
        __m128 mi = _mm_sub_ps(cj, ci);
        __m128 xx = cmul_sse3_i(pl, mi); //multiple (a+b)(a-b)
        xx = _mm_shuffle_ps(xx, xx, _MM_SHUFFLE(2,3,0,1));
        xx = _mm_mul_ps(xx, _mm_set_ps(0.25f, -0.25f, 0.25f, -0.25f));
        xx = _mm_add_ps(xx, _mm_loadu_ps((float*)(D+i)));   //+=
        _mm_storeu_ps((float*)(D+i), xx); //multiple by I/4
    }
    if(nn!=nc) {
        complex float A =   (conjf(C[nr-nn])+C[nn])/2.0f;
        complex float B = I*(conjf(C[nr-nn])-C[nn])/2.0f;
        D[nn] += A*B;
    }
}
```

I need to consider how to implement this use AVX for this 2-way r2c transform. But for better performance, 4-way or 8-way should be a better choice, maybe I should try OpenCL.

5 Conclusion

Benchmark of one example using c2c as 2-way r2c transform for fast convolution. I squeeze about 30% more performance out of FFTW, that is not an easy task. However, when the transform length goes to 4096, the boost drops to about 16%. And more or less consistant with trend in Figure 2. The source code that generates the below result and validates the result should come along with this document.

```
start testing speed....
nr=1536, nc=769 step=1536
FFT of 1536x28800 repeat 20 times:
  fftw : 4968.310059 ms
  newf : 3776.236084 ms
```