Fixed-point FFT Benchmarking

Simone Rossi

April 10, 2017

1 INTRODUCTION

In this laboratory, we studied the effects of fixed-point representation for a well known algorithm in Digital Signal Processing, the **FFT**, a fast algorithm for computing the DFT of a signal.

2 FIXED-POINT OPERATION ROUTINES

Let's analyze how fixed-point numbers are represented and how we can emulate arithmetic operations with them.

2.1 MULTIPLICATION IN Q15

This is the code for implementing a multiplication with two operands in Q15.

```
short FIX_MPY(short x, short y){
  return ((short)(((int)x * (int)y)>>15));
}
```

Simply, it casts both operands in 32 bit integers, performs the multiplication and takes the most significant bits of the result by shifting to the right of 15 positions. Before returning, the result is casted back to `short`.

2.2 Multiplication for Q25 × Q18

Here the situation is slightly different but the procedure is similar: operands are casted in 64 bits (to avoid overflow issues) and the right most 25 bits are taken as result.

```
int FIX_MPY25by18(int x,int y) {
  return ((int)(((long long)x * (long long)y) >> 17));
}
```

2.3 Saturated addition in Q15 and Q25

This is the code for implementing a saturated addition with two operands in Q15.

```
short SAT_ADD16(short x,short y) {
  if ((int)x + (int)y > 32767)
    return(32767);
  else if ((int)x + (int)y < -32767)
    return(-32768);
  else
    return(x + y);
}
```

It extends the operands on 32 bits and it checks whether the result will be an overflow or not. If so, it returns the maximum number representable on 16 bits ($2^{16} - 1 = 32767$), positive or negative according to the case. If not, it simply returns the normal sum.

For the operation in Q25, the procedure is exactly the same with the exception of the return value in case of overflow, which is now $2^{24}-1$

```
int SAT_ADD25(int x,int y) {
  if ((int)x + (int)y > (1<<24)-1)
    return((1<<24)-1);
  else if ((int)x + (int)y < -(1<<24))
    return(-(1<<24));
  else
    return(x + y);
}
```

2.4 4-POINT BUTTERFLY IN Q15

This is the code I modified for computing the simulated results in Q15.

```
bfly[0].r = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].r, +x[N2+n2].r),
   +x[2*N2+n2].r), x[3*N2+n2].r);
bfly[0].i = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].i, +x[N2+n2].i),
   +x[2*N2+n2].i), x[3*N2+n2].i);

bfly[1].r = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].r, +x[N2+n2].i),
   -x[2*N2+n2].r), -x[3*N2+n2].i);
bfly[1].i = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].i, -x[N2+n2].r),
   -x[2*N2+n2].i), +x[3*N2+n2].r);

bfly[2].r = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].r, -x[N2+n2].r),
   +x[2*N2+n2].r), -x[3*N2+n2].r);
bfly[2].i = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].i, -x[N2+n2].i),
   +x[2*N2+n2].i), -x[3*N2+n2].i);

bfly[3].r = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].r, -x[N2+n2].i),
   -x[2*N2+n2].r), +x[3*N2+n2].i);
```

```
bfly[3].i = SAT_ADD16(SAT_ADD16(SAT_ADD16(x[n2].i, +x[N2+n2].r),
   -x[2*N2+n2].i), -x[3*N2+n2].r);

for (k1 = 0; k1 < N1; k1++) {
    twiddle_fixed(&W, N, (double)k1*(double)n2);
    x[n2 + N2*k1].r = SAT_ADD16(FIX_MPY(bfly[k1].r, W.r), -
       FIX_MPY(bfly[k1].i, W.i));
    x[n2 + N2*k1].i = SAT_ADD16(FIX_MPY(bfly[k1].i, W.r), +
       FIX_MPY(bfly[k1].r, W.i));
}
```

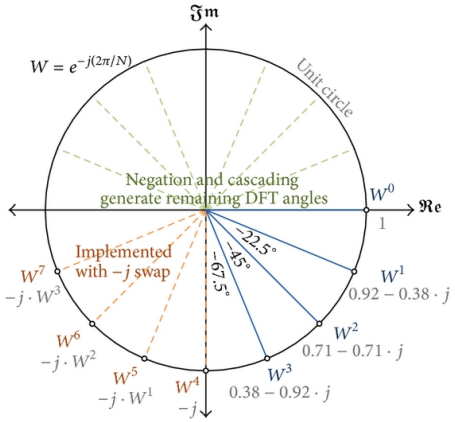
2.5 4-POINT BUTTERFLY IN XILINX DSP FORMAT

This is the code I modified for computing the simulated results in the Xilinx DSP format.

```
bfly[0].r = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].r, +x[N2+n2].r),
   +x[2*N2+n2].r), x[3*N2+n2].r);
bfly[0].i = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].i, +x[N2+n2].i),
   +x[2*N2+n2].i), x[3*N2+n2].i);

bfly[1].r = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].r, +x[N2+n2].i),
   -x[2*N2+n2].r), -x[3*N2+n2].i);
bfly[1].i = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].i, -x[N2+n2].r),
   -x[2*N2+n2].i), +x[3*N2+n2].r);

bfly[2].r = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].r, -x[N2+n2].r),
   +x[2*N2+n2].r), -x[3*N2+n2].r);
bfly[2].i = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].i, -x[N2+n2].i),
   +x[2*N2+n2].i), -x[3*N2+n2].i);

bfly[3].r = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].r, -x[N2+n2].i),
   -x[2*N2+n2].r), +x[3*N2+n2].i);
```

```
bfly[3].i = SAT_ADD25(SAT_ADD25(SAT_ADD25(x[n2].i, +x[N2+n2].r),
    -x[2*N2+n2].i), -x[3*N2+n2].r);

for (k1 = 0; k1 < N1; k1++) {
     twiddle_fixed_Q17(&W, N, (double)k1*(double)n2);
     x[n2 + N2*k1].r = SAT_ADD25(FIX_MPY25by18(bfly[k1].r, W.r), -
        FIX_MPY25by18(bfly[k1].i, W.i));
     x[n2 + N2*k1].i = SAT_ADD25(FIX_MPY25by18(bfly[k1].i, W.r), +
        FIX_MPY25by18(bfly[k1].r, W.i));
}
```

3 TWIDDLE FACTOR QUANTIZATION



Generally twiddle factors are chosen such that the lay in the unity circle in the complex domain and it's written as

$$\mathbf{W}_i = [\cos(i2\pi/N), -\sin(i2\pi/N)] \tag{3.1}$$

However this might not always be the case when we deal with quantization and fixed point representation. What is currently done is to take always the `floor` value for the sine and cosine; another possible strategy for computing the twiddle factor could be calculating all four possible combinations of approximations and taking the one which is less distant from the unity circle.

4 DISTORTION TEST

The distortion test is computed as follows.

$$\bar{x}^{in} = \sum_{i=0}^{N-1} \left[\Re\left\{x_i^{in}\right\}^2 + \Im\left\{x_i^{in}\right\}^2 \right] \tag{4.1}$$

$$\bar{\epsilon} = \sum_{i=0}^{N-1} \left[\Re\left\{ x_i^{in} - \frac{x_i^{FP}}{2^{16}-1} \right\}^2 + \Im\left\{ x_i^{in} - \frac{x_i^{FP}}{2^{16}-1} \right\}^2 \right] \tag{4.2}$$

$$SNR = 10\log_{10} \frac{\bar{x}^{in}}{\bar{\epsilon}} \tag{4.3}$$

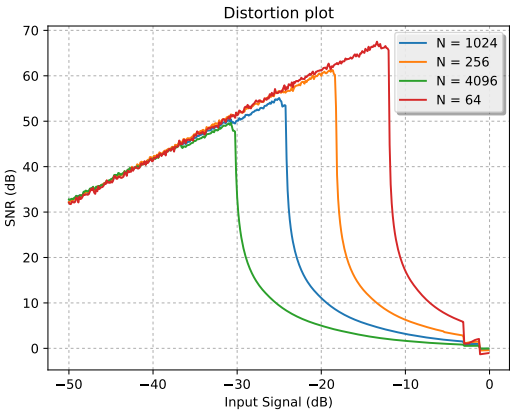
This is the code for doing so.

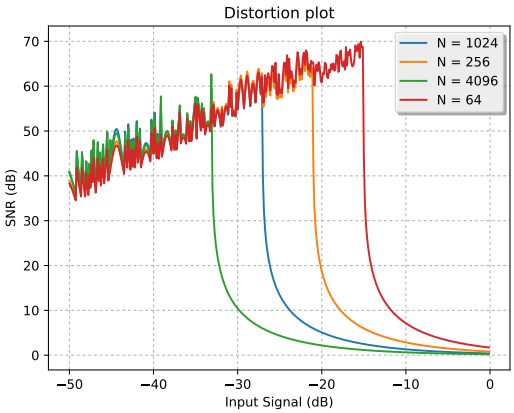
```
mean_error = 0.0;
mean_in = 0.0;

for (i=0;i<N;i++) {
  mean_in += data[i].r*data[i].r + data[i].i*data[i].i;
  mean_error += pow(data[i].r-(double)data32[i].r/32767.0,2) +
     pow(data[i].i-(double)data32[i].i/32767.0,2);
}

SNR = 10*log10(mean_in/mean_error);
```

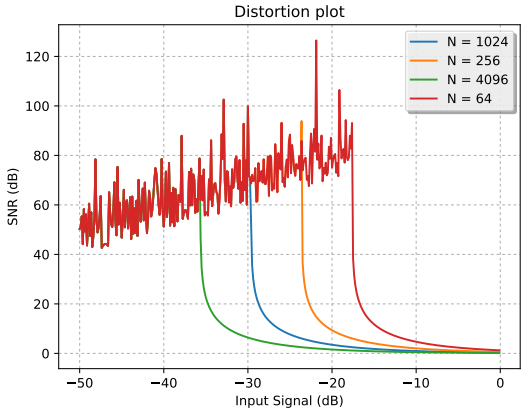
This benchmarking has been performed on three signals. The first test is with a sinusoidal input. The second a random 16-QAM signal as input and finally the third is a white-noise signal.

5 Results

These are the results I got in the two cases. The first one is the traditional Q15 format.

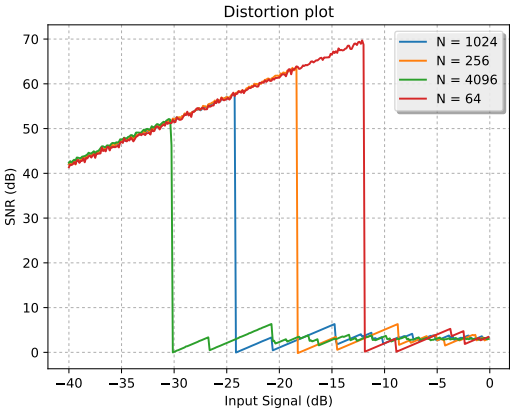


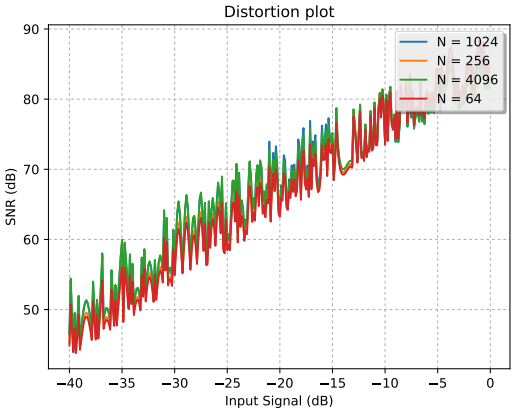


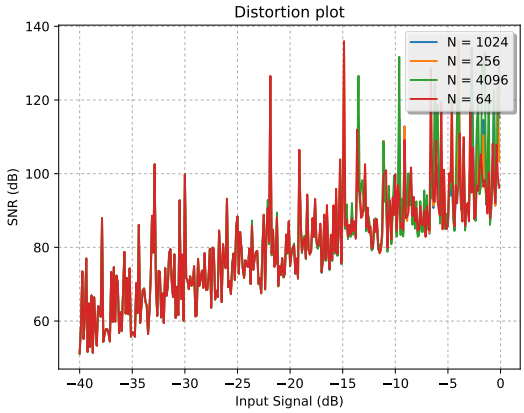


As we can see, in all the three tests, the system suffers performance degradation when the input signal power starts to become higher and higher, possibly due to overflow/underflow in some butterfly stage. As expected, the lower the number of FFT points, the higher is the breakdown point.

The following one, on the other hand, is the one with the Xilinx DSP format.







With sinusoidal input, the behaviour is not so different from the Q15 but the results for the other two tests is interesting. It seems that this particular format for fixed point can avoid destructive overflows.