Fast Fourier Transform Caterpillar Method

Developing fastest FFT implementation based on precompile tool using data driven approach.
By Alexander Semjonov

Preface

With increasing processing requirements performance is becoming a bottleneck for applications where DSP calculations are used. In most cases, the use of a specialized hardware is not an option, so the problem requires a software solution. To achieve higher performance while keeping implementation cost to a minimum a data driven programming approach may be used. With this programming model different data sets control the flow of the program based on a generic form of the program logic. It is often claimed that a data driven approach is too narrow and less applicable for widely used algorithms. However, a precompile tool or compiler extensions can be used to overcome these concerns. This paper describes the methodology for building two data-driven FFT algorithms on a precompiler tool base and promotes the discovered *caterpillar method* as an alternation for the widely used butterfly method.

Theory

Time domain signals are converted to frequency domain signals by means of the Digital Fourier Transform:

$$X_k = \sum_{n=0}^{N-1} x_n * \omega_N^{kn}, \quad k= 0..N-1, \quad \omega_N = e^{-2\pi i/N}, \quad i = \sqrt[2]{-1};$$

In this scheme both domains contain a signal composed of N complex points. Each of these complex points x is made up of two parts, the real value and the imaginary value. The complex twiddle factor ω represents a “rotating vector” which maps onto the unit circle in the complex plane.
Practically, DSP applications are based on Fast Fourier Transform (FFT) algorithms which have less computation complexity. The first step in computing an FFT is to split the current N-point signal $X=(x_0, x_1 \dots x_{n-1})$ into two $A=(x_0, x_2 \dots x_{n-2})$ and $B=(x_1, x_3 \dots x_{n-1})$ each consisting of N/2 points:

$$A_k = \sum_{n=0}^{N/2-1} x_{2n} * \omega_N^{k(2n)};$$

$$B_k = \sum_{n=0}^{N/2-1} x_{2n+1} * \omega_N^{k(2n+1)} = \sum_{n=0}^{N/2-1} x_{2n+1} * \omega_N^{k} * \omega_N^{k(2n)} = \omega_N^{k} * \sum_{n=0}^{N/2-1} x_{2n+1} * \omega_N^{k(2n)};$$

Consequently, $X_k = A_k + \omega_N^{k} * B_k$;
The DFT of the even N/2 points is combined with the DFT of the odd N/2 points to produce a single N-point DFT. In the next step N/2-point DFTs A_k and B_k are calculated using N/4-point DFTs in the same way. Such deduction is also applicable for Inverse FFT, which is out of scope for the paper.
In the narrow sense, a standard multiply–accumulate operation (MAC) is `a = a + b * c`, where “a” is an accumulator. In the wide sense, MAC can be presented as `c = a + w * b`, that is an elementary operation of an FFT. The number of points for an FFT has to be a power of 2. A direct DFT implementation takes N^2 operations for N points. For an FFT the total computation is proportional to $N*\log_2(N)$.

Modern C++ Implementation

The direct FFT implementation according to the formulas above is not so difficult. The pseudo code for an FFT recursive algorithm can be found in [1]. To prove the concept the algorithm is converted to modern C++ code which is provided below:

```
#include <complex>
#include <vector>
#include <algorithm>
#include <iostream>
#include <math.h>

#define M_PI 3.14159265358979323846
using namespace std;
typedef complex<double> w_type;

static vector<w_type> fft(const vector<w_type> &In)
{
  int i = 0, wi = 0;
  int n = In.size();
  vector<w_type> A(n / 2), B(n / 2), Out(n);
  if (n == 1) {
      return vector<w_type>(1, In[0]);
  }
  i = 0;
  copy_if( In.begin(), In.end(), A.begin(), [&i] (w_type e) {
           return !(i++ % 2);
  } );
  copy_if( In.begin(), In.end(), B.begin(), [&i] (w_type e) {
           return (i++ % 2);
  } );

  vector<w_type> At = fft(A);
  vector<w_type> Bt = fft(B);

  transform(At.begin(), At.end(), Bt.begin(), Out.begin(), [&wi, &n]
  (w_type& a, w_type& b) {
       return  a + b * exp(w_type(0, 2 * M_PI * wi++ / n));
  });
  transform(At.begin(), At.end(), Bt.begin(), Out.begin() + n / 2, [&wi, &n]
   (w_type& a, w_type& b) {
      return  a + b * exp(w_type(0, 2 * M_PI * wi++ / n));
  });
  return Out;
}
void main(int argc, char* argv[])
{
    int ln = (int)floor( log(argc - 1.0) / log(2.0) );
    vector<w_type> In(1 << ln);
    std::transform(argv + 1, argv + argc, In.begin(),[&](const char* arg) {
        return w_type(atof(arg), 0);
```

```
    });
    vector<w_type> Out = fft(In);
    for (vector<w_type>::iterator itr = Out.begin(); itr != Out.end(); itr++) {
        cout  << *itr << endl;
    }
}
```

Listing 1. C++ implementation of FFT Recursive Algorithm

In the `main` procedure the number of input program arguments is truncated to the nearest power of 2 that defines both the transform size and the size of the input vector. The standard C++ complex template library is used for these computations. STL vectors of complex values are used for processing. Real parts of complex values of input vector are composed of input program arguments. The imaginary parts are filled by zeros that represents the time domain signal.
The first level of recursion splits the n-point signal `In` into 2 signals `A` and `B` each consisting of n/2 points. The next level breaks the data into 4 signals of n/4 points. The recursion stops when only 1-point signals are left. Transformed `At` and `Bt` vectors are used to compose of the output vector `Out`.
Finally, the program prints real and imaginary parts of the frequency domain signal from the output vector.

Implementation on Different Styles

The C++ implementation has too high a level of abstraction, so it may be not applicable for low-cost platforms. This is a reason to provide a C implementation. The problem, however, lies in achieving the right trade-off between portability and efficiency. Software engineers need to apply different programming styles depending on the requirements for the application. Platform optimizations for code efficiency can be applicable for embedded applications. The higher the level of programming style that is applied the fewer programming efforts are required. In general code efficiency has an inverse relation to this.
The MAC is the elementary FFT operation which can be presented as `c = a + w * b` in complex notation. The following example demonstrates applicability of different programming styles to implement complex MAC operation as `w_mac` function.

Level \ Prototype	`void w_mac( w_type* cc, w_type a, w_type w, w_type b)`
High level	`*cc = a + b * exp(w_type(0, 2 * M_PI * w / n))`
Classic	`cc->r = a.r + w.r * b.r - w.i * b.i;` `cc->i = a.i + w.r * b.i + w.i * b.r;`

Embedded (prepared for platform MAC optimization)	`register double reg;`
	`reg = mac(a.r, w.r, b.r);`
	`cc->r = mac(reg, -w.i, b.i);`
	`reg = mac(a.i, w.r, b.i );`
	`cc->i = mac(reg, w.i * b.r );`

Table 1. Example of MAC Function

The logarithm of FFT size to base 2 is the number of FFT recursions. Several styles of the `n2ln` useful routine for such calculations are provided below.

Level \ Prototype	`int n2ln( int n )`
High level	`return (int)floor( log(n) / log(2.0) );`
Classic	`int ln = 0;`

<table>
<tr><td>

- required 15 cycles for the worst case

</td><td>

```
while (n >>= 1) ln++;
return ln;
```

</td></tr>
<tr><td>

Embedded

- the worst case equals the average case
- 3 assembler shifts, 4 branches and 1 move are required for each result
- can be implemented as a macro to calculate constant values at compiler time

</td><td>

```
return  n/256
            ?n/4048
               ?n/16348
                   ?n/32768?15:14
                   :n/8096?13:12
               :n/1024
                   ?n/2048?11:10
                   :n/512?9:8
            :n/16
               ?n/64
                   ?n/128?7:6
                   :n/32?5:4
               :n/4
                   ?n/8?3:2
                   :n/2?1:0;
```

</td></tr>
</table>

Table 2. Example of Log_2 Function

Classic C Implementation

This C implementation is algorithmically similar to the C++ implementation above.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define M_PI 3.14159265358979323846

typedef struct { double r; double i; } w_type;

int n2ln( int n );
void w_mac( w_type* cc, w_type a, w_type w, w_type b );

static void fft0( w_type InOut[], int n )
{
    int i;
    w_type w, *A, *B;

    if (n == 1) return;
    A = malloc( sizeof(w_type) * n / 2 );
    B = malloc( sizeof(w_type) * n / 2 );
    for (i = 0; i < n / 2; i++) {
        A[i] = InOut[i * 2];
        B[i] = InOut[i * 2 + 1];
    }
    fft0( A, n / 2 );
    fft0( B, n / 2 );
    for (i = 0; i < n; i++) {
        w.r = cos(2 * M_PI * i / n);
        w.i = sin(2 * M_PI * i / n);
        w_mac( &InOut[i], A[i % (n / 2)], w, B[i % (n / 2)] );
```

```
    }
    free( A );
    free( B );
}
void main( int argc, char * argv[] )
{
    int i;
    int ln = n2ln(argc - 1);
    w_type* InOut = malloc( (1 << ln) * sizeof(w_type) );
    for (i = 0; i < (1 << ln); i++) {
        InOut[i].r = atof(argv[i+1]);
        InOut[i].i = 0;
    }
    fft0( InOut, 1 << ln );
    for(i = 0; i < (1 << ln); i++) {
        printf("%.4f %.4f\n", InOut[i].r, InOut[i].i);
    }
}
```

Listing 2. Classic C Implementation

Several algorithmic modifications are introduced to the classic C example:

- Complex number computations are done manually
- Due to explicit memory allocation the actual input buffer size is passed to the recursive procedure
- Input and output vectors of C++ implementation are jointed together and shared as one `InOut` buffer of complex values.

Data-Driven Modifications

The example below is made from Classic C Implementation for demonstration of an approach to build a precompiler tool which can create an embedded style program.

```
#include <stdlib.h>
#include <stdio.h>
#include <math.h>

#define M_PI 3.14159265358979323846

#define LN_FFT 4    /* number of samples is 1 << LN_FFT */

typedef struct { double r; double i; } w_type;

int n2ln( int n );
void w_mac( w_type* cc, w_type a, w_type w, w_type b );

static struct tMac {
    w_type *c, *a, *b, w;
} Mac[LN_FFT * (1 << LN_FFT)];
static int nMac;
static w_type DataTrace[LN_FFT + 1][1 << LN_FFT];
```

```
static int BusyDataTrace[LN_FFT + 1];

static void calculate_macs()
{
    int i;
    for (i = 0; i < nMac; i++) {
        w_mac(Mac[i].c, *Mac[i].a, Mac[i].w, *Mac[i].b);
    }
}

static void record_mac( w_type** cc, w_type* a, w_type w, w_type *b, int n )
{
    int ln = n2ln(n);
    int i = BusyDataTrace[ln]++;

    *cc = &DataTrace[ln][i];
    Mac[nMac].c = &DataTrace[ln][i];
    Mac[nMac].w = w;
    Mac[nMac].a = a;
    Mac[nMac].b = b;
    nMac++;
}

static void fft0( w_type* InOut[], int n )
{
    int i;
    w_type w, **A, **B;

    if (n == 1) return;
    A = malloc( sizeof(w_type*) * n / 2 );
    B = malloc( sizeof(w_type*) * n / 2 );
    for (i = 0; i < n / 2; i++) {
        A[i] = InOut[i * 2];
        B[i] = InOut[i * 2 + 1];
    }
    fft0( &A[0], n / 2 );
    fft0( &B[0], n / 2 );
    for (i = 0; i < n; i++) {
        w.r = cos(2 * M_PI * i / n);
        w.i = sin(2 * M_PI * i / n);
        record_mac( &InOut[i], A[i % (n / 2)], w, B[i % (n / 2)], n );
    }
    free(A);
    free(B);
}

void main( int argc, char* argv[] )
{
    int i;
    w_type** InOut = malloc( sizeof(w_type*) * (1 << LN_FFT) );
    for (i = 0; i < (1 << LN_FFT); i++) {
        InOut[i] = &DataTrace[0][i];
        DataTrace[0][i].r = atof( argv[i % (argc - 1) + 1] );
        DataTrace[0][i].i = 0;
```

```
    }
    fft0( InOut, 1 << LN_FFT );
    calculate_macs();
    for(i = 0; i < (1 << LN_FFT); i++) {
        printf("%.4f %.4f\n", DataTrace[LN_FFT][i].r, DataTrace[LN_FFT][i].i);
    }
    free(InOut);
}
```

Listing 3. The Example of Data-Driven FFT

This FFT recursive algorithm has been rewritten to use a data driven approach for FFT computations. The recursive procedure has been rebased to support buffers of pointers of the complex type instead to using the complex type directly. Real computations are recorded to the `Mac` array to be calculated later. The two-dimensional array `DataTrace` is used to trace all calculations. When all recursions of the FFT are done the user needs to call `calculate_macs`. All recoded complex macs are calculated in the order in which they were recoded. The `calculate_macs` procedure has only one cycle so it is fast as possible. The algorithm has the theoretical FFT complexity $n*\log_2(n)$ and now this relates to the memory usage too. This means that both `Mac` and `DataTrace` arrays have $n*\log_2(n)$ elements. This can also be applicable for modern computation systems when efficiency is the top priority, but low cost systems will need extra modifications.

Table Implementation Based on Generated Code

For cross-platform systems the data-driven approach turns into the static configuration approach. This means that the `Mac` array from the example above is written as C initialized structures to be built and run as new program. There are two degrees of freedom for transposition of `Mac` array elements:

- `Mac` array elements of same recursion level can be safely exchanged
- Pointers of a `Mac` array element can be changed to utilize other `DataTrace` elements. Of course, care is needed for other pointers that refer to the changed places.

This allows the required optimization objectives to be achieved.
The `Mac` array from the previous example utilizes $N*\log_2(N)$ RAM elements from the `DataTrace` arrays. The previous program has been modified to optimize RAM usage to $N+2$ elements and to generate the code below:

```
#include <stdlib.h>
#include <stdio.h>

#define LN_FFT  4  /* power of 2 is number of fft points */
/*   */
#define W_0_02   1.000000000000000 /* angle 0.00 dg */
#define W_1_04   0.000000000000000 /* angle 90.00 dg */
#define W_1_08   0.707106781186548 /* angle 45.00 dg */
#define W_1_16   0.923879532511287 /* angle 22.50 dg */
#define W_3_16   0.382683432365090 /* angle 67.50 dg */

typedef struct { double r; double i; } w_type;
static const struct fft_tbl {
    double wr, wi;
```

```
    unsigned char c, a, b;
} tbl[] = {
{ w_0_02,+w_1_04,16, 0, 8}, {-w_0_02,+w_1_04,17, 0, 8},
{ w_0_02,+w_1_04, 0, 4,12}, {-w_0_02,+w_1_04, 8, 4,12},
{ w_0_02,+w_1_04, 4, 2,10}, {-w_0_02,+w_1_04,12, 2,10},
{ w_0_02,+w_1_04, 2, 6,14}, {-w_0_02,+w_1_04,10, 6,14},
{ w_0_02,+w_1_04, 6, 1, 9}, {-w_0_02,+w_1_04,14, 1, 9},
{ w_0_02,+w_1_04, 1, 5,13}, {-w_0_02,+w_1_04, 9, 5,13},
{ w_0_02,+w_1_04, 5, 3,11}, {-w_0_02,+w_1_04,13, 3,11},
{ w_0_02,+w_1_04, 3, 7,15}, {-w_0_02,+w_1_04,11, 7,15},
{ w_0_02,+w_1_04, 7,16, 0}, {-w_0_02,+w_1_04,15,16, 0},
{ w_1_04,+w_0_02,16,17, 8}, {-w_1_04,-w_0_02, 0,17, 8},
{ w_0_02,+w_1_04,17, 4, 2}, {-w_0_02,+w_1_04, 8, 4, 2},
{ w_1_04,+w_0_02, 4,12,10}, {-w_1_04,-w_0_02, 2,12,10},
{ w_0_02,+w_1_04,12, 6, 1}, {-w_0_02,+w_1_04,10, 6, 1},
{ w_1_04,+w_0_02, 6,14, 9}, {-w_1_04,-w_0_02, 1,14, 9},
{ w_0_02,+w_1_04,14, 5, 3}, {-w_0_02,+w_1_04, 9, 5, 3},
{ w_1_04,+w_0_02, 5,13,11}, {-w_1_04,-w_0_02, 3,13,11},
{ w_0_02,+w_1_04,13, 7,17}, {-w_0_02,+w_1_04,11, 7,17},
{ w_1_08,+w_1_08, 7,16, 4}, {-w_1_08,-w_1_08,17,16, 4},
{ w_1_04,+w_0_02,16,15, 8}, {-w_1_04,-w_0_02, 4,15, 8},
{-w_1_08,+w_1_08,15, 0, 2}, { w_1_08,-w_1_08, 8, 0, 2},
{ w_0_02,+w_1_04, 0,12,14}, {-w_0_02,+w_1_04, 2,12,14},
{ w_1_08,+w_1_08,12, 6, 5}, {-w_1_08,-w_1_08,14, 6, 5},
{ w_1_04,+w_0_02, 6,10, 9}, {-w_1_04,-w_0_02, 5,10, 9},
{-w_1_08,+w_1_08,10, 1, 3}, { w_1_08,-w_1_08, 9, 1, 3},
{ w_0_02,+w_1_04, 1,13, 0}, {-w_0_02,+w_1_04, 3,13, 0},
{ w_1_16,+w_3_16,13, 7,12}, {-w_1_16,-w_3_16, 0, 7,12},
{ w_1_08,+w_1_08, 7,16, 6}, {-w_1_08,-w_1_08,12,16, 6},
{ w_3_16,+w_1_16,16,15,10}, {-w_3_16,-w_1_16, 6,15,10},
{ w_1_04,+w_0_02,15,11, 2}, {-w_1_04,-w_0_02,10,11, 2},
{-w_3_16,+w_1_16,11,17,14}, { w_3_16,-w_1_16, 2,17,14},
{-w_1_08,+w_1_08,17, 4, 5}, { w_1_08,-w_1_08,14, 4, 5},
{-w_1_16,+w_3_16, 4, 8, 9}, { w_1_16,-w_3_16, 5, 8, 9},
};

static const unsigned char OutOrder[]={
1,13,7,16,15,11,17,4,3,0,12,6,10,2,14,5,};

static struct { double r; double i; } XY[(1 << LN_FFT) + 2];

void fft_table()
{
    int i;
    register const struct fft_tbl* t;

    for (i = 0, t = tbl; i < sizeof(tbl) / sizeof(tbl[0]); i++, t++) {
        XY[t->c].r = XY[t->a].r + t->wr * XY[t->b].r - t->wi * XY[t->b].i;
        XY[t->c].i = XY[t->a].i + t->wr * XY[t->b].i + t->wi * XY[t->b].r;
    }
}
void main(int argc, char* argv[])
{
    int i;
```

```
        for (i = 0; i < (1 << LN_FFT); i++) {
            XY[i].r = atof( argv[i % (argc - 1) + 1] );
            XY[i].i = 0;
        }
        fft_table();
        for(i = 0; i < (1 << LN_FFT); i++) {
            printf("%.4f %.4f\n", XY[OutOrder[i]].r, XY[OutOrder[i]].i);
        }
    }
```

Listing 4. The FFT Table Implementation

This is an example of an FFT for 16 points. One element of the `tbl` array contains a complex value of the twiddle factor and 3 offsets to provide one complex MAC operation based on 3 RAM elements of XY memory array[1]. This code is fast because it utilizes only one "`for`" cycle.
One FFT complex operation consists of 4 floating point operations, so a data-driven table for floating or fixed operations can be generated to provide optimization capabilities to achieve highest efficiency. Such optimizations in a matrix or algorithmic form are well-known in reference literature [2] but now they can be described on the proposed framework base:

1. Optimizations based on specific FFT particularities[2]:
    - Input time domain signal is presented by N complex points signal with zero imaginary part
    - Output frequency domain signal has duplication scheme. The real part of point N/2+1 equals to the real part of point N/2-1. This is applied until the real part of point N-1 which is the same as the real part of point 1. The imaginary part of point N/2+1 is the negative value of imaginary part of point N/2-1. Samples 0 and N/2 do not have a matching point. Therefore, the points N/2+1 through N-1 are redundant and output frequency domain signal can be presented enough by points 0 through N/2.

    These optimizations can achieve $N*\log_2(N - 1/2 - 1/2) = N*\log_2(N)/2$ performance.
2. Optimizations based on degenerated twiddle factors:
    - Operations with twiddle factors 0,1 and -1 can be reduced to simple addition;
    - Operations with twiddle factor `0.7071/* angle 45.00 dg */` which is the same for real and imaginary part can save one MAC operation
3. Optimizations for input and output representations:
    - The example above uses the `OutOrder` array to restore the standard FFT order of samples of the frequency domain signal. In fact, this order is rarely used for further processing as it. The tool can generate such `OutOrder` array with the required custom order.
    - As a rule, before an FFT the time domain signal can be windowed and/or normalized. This operation can be introduced to the FFT table method to be supported by the tool.

Caterpillar Implementation Based on Generated Code

[1] The paper describes common principles of a data-driven approach and does not consider any constant table optimizations. For most architectures a 5 elements structured array is less effective then 5 separate arrays.
[2] Optimizations for Inverse FFT are different.

An FFT table approach utilizes a table of $N*\log_2(N)$ elements. Another optimization approach is to combine the table elements that are similar in some fields. The following example demonstrates grouping complex MACs against twiddle factors to generate them as code statements:

```
#include <stdlib.h>
#include <stdio.h>

#define LN_FFT  5  /* power of 2 is number of fft points */
/*   */
#define W_0_02   1.000000000000000 /* angle 0.00 dg */
#define W_0_04   0.000000000000000 /* angle 90.00 dg */
#define W_0_08   0.707106781186547 /* angle 45.00 dg */
#define W_0_16   0.923879532511287 /* angle 22.50 dg */
#define W_1_16   0.382683432365090 /* angle 67.50 dg */
#define W_0_32   0.980785280403230 /* angle 11.25 dg */
#define W_1_32   0.831469612302545 /* angle 33.75 dg */
#define W_2_32   0.555570233019602 /* angle 56.25 dg */
#define W_3_32   0.195090322016128 /* angle 78.75 dg */

typedef struct { double r; double i; } w_type;

#define X2Y(a) (a + (1 << (LN_FFT - 1)))
#define XMAC(c, a, wr, wi) \
    c->r = a->r + wr * X2Y(a)->r - wi * X2Y(a)->i; \
    c->i = a->i + wr * X2Y(a)->i + wi * X2Y(a)->r;

static w_type XY[2][(1 << LN_FFT)];
static const w_type* pOut = LN_FFT % 2 ? &XY[1][0] : &XY[0][0];
static const unsigned char OutOrder[]={31,15,23,14,27,13,22,12,29,11,21,
10,26,9,20,8,30,7,19,6,25,5,18,4,28,3,17,2,24,1,16,0,};

void fft_caterpillar()
{
    int i, j, lim;
    register  w_type *pc, *pa; /* pb = a + (1 << (LN_FFT - 1)) */
    for (i = 1; i <= LN_FFT; i++) {
        pc = i % 2 ? &XY[1][0] : &XY[0][0];
        pa = i % 2 ? &XY[0][0] : &XY[1][0];
        lim = 1 << (LN_FFT - i);
        for (j = 0; j < lim; j++) {
            switch (i) {
            case 5:
                XMAC(pc, pa,  W_0_32, -W_3_32); pc++; pa += 1;
                XMAC(pc, pa,  W_1_32, -W_2_32); pc++; pa += 1;
                XMAC(pc, pa,  W_2_32, -W_1_32); pc++; pa += 1;
                XMAC(pc, pa,  W_3_32, -W_0_32); pc++; pa += 1;
                XMAC(pc, pa, -W_3_32, -W_0_32); pc++; pa += 1;
                XMAC(pc, pa, -W_2_32, -W_1_32); pc++; pa += 1;
                XMAC(pc, pa, -W_1_32, -W_2_32); pc++; pa += 1;
                XMAC(pc, pa, -W_0_32, -W_3_32); pc++; pa += 1;
                pa -= 8;
                XMAC(pc, pa, -W_0_32, +W_3_32); pc++; pa += 1;
                XMAC(pc, pa, -W_1_32, +W_2_32); pc++; pa += 1;
                XMAC(pc, pa, -W_2_32, +W_1_32); pc++; pa += 1;
```

```
                XMAC(pc, pa, -w_3_32, +w_0_32); pc++; pa += 1;
                XMAC(pc, pa,  w_3_32, +w_0_32); pc++; pa += 1;
                XMAC(pc, pa,  w_2_32, +w_1_32); pc++; pa += 1;
                XMAC(pc, pa,  w_1_32, +w_2_32); pc++; pa += 1;
                XMAC(pc, pa,  w_0_32, +w_3_32); pc++; pa += 1;
            case 4:
                XMAC(pc, pa,  w_0_16, -w_1_16); pc++; pa += 1;
                XMAC(pc, pa,  w_1_16, -w_0_16); pc++; pa += 1;
                XMAC(pc, pa, -w_1_16, -w_0_16); pc++; pa += 1;
                XMAC(pc, pa, -w_0_16, -w_1_16); pc++; pa += 1;
                pa -= 4;
                XMAC(pc, pa, -w_0_16, +w_1_16); pc++; pa += 1;
                XMAC(pc, pa, -w_1_16, +w_0_16); pc++; pa += 1;
                XMAC(pc, pa,  w_1_16, +w_0_16); pc++; pa += 1;
                XMAC(pc, pa,  w_0_16, +w_1_16); pc++; pa += 1;
            case 3:
                XMAC(pc, pa,  w_0_08, -w_0_08); pc++; pa += 1;
                XMAC(pc, pa, -w_0_08, -w_0_08); pc++; pa += 1;
                pa -= 2;
                XMAC(pc, pa, -w_0_08, +w_0_08); pc++; pa += 1;
                XMAC(pc, pa,  w_0_08, +w_0_08); pc++; pa += 1;
            case 2:
                XMAC(pc, pa, -w_0_04, -w_0_02); pc++; pa += 1;
                pa -= 1;
                XMAC(pc, pa,  w_0_04, +w_0_02); pc++; pa += 1;
            case 1:
                XMAC(pc, pa, -w_0_02, +w_0_04); pc++; pa += 1;
                pa -= 1;
            case 0:
                XMAC(pc, pa,  w_0_02, +w_0_04); pc++; pa += 1;
            }
        }
    }
}
void main(int argc, char* argv[])
{
    int i;
    for (i = 0; i < (1 << LN_FFT); i++) {
        XY[0][i].r = atof( argv[i % (argc - 1) + 1] );
        XY[0][i].i = 0;
    }
    fft_caterpillar();
    for(i = 0; i < (1 << LN_FFT); i++) {
        printf("%.4f %.4f\n", pOut[OutOrder[i]].r, pOut[OutOrder[i]].i);
    }
}
```

Listing 5. The FFT Caterpillar Implementation

This is an example of a 32 point FFT. The number of the complex MACs is N. The two dimension array XY is organized on a swap scheme and represents 4 RAM memory banks. Each XMAC accesses 3 memory banks simultaneously: one for reading and two for writing. This program is prepared for a DSP platform. The main advantage of this method is simplification of the address arithmetic. The traditional approach of building effective FFT algorithms is to construct cycles on data arrays with sophisticated address

arithmetic. Graphical representations of this arithmetic looks like butterfly wings. In this example the butterfly transpositions are decomposed, so it is *caterpillar method.* Also the long `switch` operator remains of a caterpillar too.

Further Machine Optimizations

Almost all of the standard optimization described in the table method section can be applied for the caterpillar method. The major advantage of the caterpillar method is the ability to optimize it as a set of abstract MAC instructions. The XMAC statement above implements `c = a + ω * b` complex equation. It can be presented by 4 MACs like this:

```
reg = a.real + w.real * b.real;
c.real = reg - w.imag * b.imag;
reg = a.imag + w.real * b.imag;
c.imag = reg + w.imag * b.real;
```

Practically, the XMAC can be presented by Intel AVX intrinsics as:

```
#define XMAC(c, a, wr, wi) \
   vec1 = _mm_setr_pd(wr, wi); \
   vec2 = *( __m128d*)&X2Y(a)->r;  \
   vec1 = _mm_hadd_pd(_mm_mul_pd(vec1, _mm_mul_pd(vec2, neg)), \
          _mm_mul_pd(vec1, _mm_shuffle_pd(vec2, vec2, 1))); \
   *( __m128d*)c = _mm_add_pd(*( __m128d*)a, vec1);
```

The provided example of the caterpillar method operates on two 64-bits double words simultaneously. For AVX3 four nearest XMACs can be united together into one single statement to operate on eight 64-bits double words similar way. It can be more for 32-bit float or fixed point implementation.

Conclusion

A well-elaborated tool can generate fast data-driven algorithm implementations for specific customer requirements. These algorithm implementations can be prepared for compiler optimizations or they can be based on assembler code. In general, large constant tables in ROM are acceptable for low cost platforms, so highest efficiency can be achieved. Alternatively, the FFT caterpillar method can be adjusted for most DSP architectures. This approach can be extended for a wide range of filtering and transformation algorithms.

References

[1] Notes on Recursive FFT (Fast Fourier Transform) algorithm, Fall 2010, COSC 511, Susan Haynes
http://www.emunix.emich.edu/~haynes/Papers/FFT/NotesOnRecursiveFFT.pdf

[2] Notes on the FFT, C. S. Burrus, Department of Electrical and Computer Engineering
Rice University, Houston, TX 77251-1892, http://www.jjj.de/fft/fftnote.txt

[3] Some `fft_caterpillar` examples, https://github.com/r35382/fft_caterpillar