EC527 Project: Accelerating Fast Fourier Transform with FPGA

Lixing Shen

May 4, 2023

Table of Contents

1 Description of the Algorithm

1.1 Base: **Discrete Fourier Transform**

Discrete Fourier Transform (**DFT**), in 1D, is a mathematical operation that takes a vector as input and computes its frequency component as the output. The following equation mathematically shows the operations for getting the frequency components [1]:

$$X_m = \sum_{n=0}^{N-1} x_n \omega^{nm}$$

The operation is computationally expensive, and its complexity is on the order of $\mathbf{O(N^2)}$.

1.2 **Fast Fourier Transform**

The summation in the previous equation can be divided into two summations [1]:

$$X_m = \sum_{n=0}^{N/2-1} x_{2n} \omega^{2nm} + \omega^m \sum_{n=0}^{N/2-1} x_{2n+1} \omega^{2nm}$$

And if N is a power of two, this operation can be performed repeatedly on each sub-summation until we are down to N = 1. This means that, when N is a power of two, the Discrete Fourier Transform problem can be solved using recursion. This is the idea of the **Cooley Tukey FFT algorithm**, which is one of the most widely used approach in real world [1]. It reduces the complexity of Discrete Fourier Transform from the $O(N^2)$ to $\mathbf{O(N \log_2 N)}$. The performance improvement becomes more apparent when N becomes large.

2 Description of the System

All the tests in this exploration are performed on ArtyZ7-20 from Digilent®. The main chip on the board has two major components. One is the **Processing System** (**PS**), and the other is the **Programable Logic** (**PL**).

2.1 Processing System

All serial versions of the FFT algorithm in this exploration, both optimized and unoptimized ones, run on the PS, which is a dual core **32-bit ARM Cortex-A9 processor** capable of running at **650 MHz**. One thing to notice is that only core #0 is used and no multithreading is tested because of project time limitations.

2.2 Programmable Logic

The hardware FFT accelerator in this exploration is emulated on the PL, which is an **Artix-7 series FPGA from Xilinx**. One thing to notice is that not all components of the FFT accelerator are running at the same 650 MHz frequency as the PS.

3 Serial Algorithm and Optimization

3.1 Implementation without Recursion

The recursive version of the algorithm is never tested in this exploration because it takes up too much memory on storing function stacks. The main memory available to the CPU is merely 512MB, of which some part of it is used by the operating system, leaving very little space for the program.

The algorithm I got from my second source [2] already has the recursion substituted with for-loops, which is a huge optimization over the recursive version.

The following is the pseudocode representation of the algorithm in an 8-point FFT operation:

for j = 0 → N-1 step 2 do

    $temp1_j \leftarrow X_j + X_{j+1}$

    $temp1_{j+1} \leftarrow X_j - X_{j+1}$

end

for j = 0 → N-1 step 4 do

    for k = 0 → 1 do

        $temp2_{j+k} \leftarrow temp1_{j+k} + \omega_{2k} \cdot temp1_{j+k+2}$

        $temp2_{j+k+2} \leftarrow temp1_{j+k} - \omega_{2k} \cdot temp1_{j+k+2}$

    end

end

for j = 0 → N/2 - 1 do

    $Y_j \leftarrow temp2_j + \omega_j \cdot temp2_{j+4}$

    $Y_{j+4} \leftarrow temp2_j - \omega_j \cdot temp2_{j+4}$

End

where X is the input array and Y is the output array, and N is the number of points in FFT, which is 8 in this case. “temp1” and “temp2” are two arrays each of size N to store some intermediate results in the algorithm.

This is also the reference code used to validate the outputs from the optimized code and FPGA.

3.2 **Caching for Performance**

Two pairs of pseudocodes in the previous section are marked in red and blue respectively. They both require at least two indexing operations and one multiplication. I think it is a good idea to cache their results so the computations will only be done once but the results will be used twice.

3.3 Implicit Caching on Array Index

The pseudocode only does FFT operation once. If it is necessary to perform FFT operation more than once, when the input arrays are stored in a larger array and FFT operation is to be performed on each of these array, then this second optimization can be adopted.

One way of indexing into the sub-array within the longer array is to add an offset to the number used for indexing each time indexing into the input and output array. The benefit of this is not having to change other parts of the code. For example, X_j will become X_{i+j}, where i is the offset.

However, since the actual implementation of this pseudocode is C/C++, X and Y are just pointers to the start of the larger array.

We can apply the idea of "Caching for Performance" again by incrementing X and Y each by N after each round of FFT operation. By doing so, some add operations are eliminated.

3.4 Unroll Loop or Not?

Loop unrolling is a classic performance optimization and usually do bring higher performance on many processors.

However, performance gain is not guaranteed for all. Performance degradation is observed with loop unrolling in this case, as will be shown in the Experiments and Results section.

My guess is that too few instructions are executed in parallel in an ARM Cortex-A9 core, which leads to little performance improvement that cannot even cover the performance cost of swapping in more instructions to its cache.

The results were collected with the first, the third, and the inner one of the second for-loops unrolled by a factor of two.

3.5 Bit Reversal Function

There is a bit-reversal function which is invoked before the FFT algorithm. This function take an array in natural order as input and fills another array in bit-reversed order.

4 Data and Memory in Serial Code

4.1 Memory Information

There is only one DDR3 chip on the development board accessible from the PS. It is also accessible from the PL if **Direct Memory Access** (**DMA**) is implemented on the PL. The bandwidth of the memory is 1050 MBps, which is known from the product reference manual.

If all the bandwidth are used for transferring data, the bandwidth is 131.25M single-precision complex numbers per second.

There are 32KB L1 instruction cache and 32KB L1 data cache within each of the Cortex-A9 cores while the two cores share a 512KB L2 cache. There is no L3 cache.

4.2 Data Structure

Both the input array and the output array are stored sequentially in the DDR3 memory. However, during each round of FFT operation, the data currently being processed is very likely cached. The FFT operation tested is for 8-point single-precision complex numbers. A single-precision complex number takes 64 bits of space, as the real and the imaginary part are both of type single-precision float. Thus, for each FFT operation, the amount of space required by the input points and the output points is $2 \times 8points \times 64b/point = 128B$, which can fit into L1 data cache.

One thing to notice is that **both the input and output are stored as Array of Structures** instead of Structure of Arrays. The purpose of this design is to make the data transfer mechanism for the PL part easier to implement. For fairness, the PS also has to work with Array of Structures.

4.3 Memory Access Pattern

The memory access pattern is interleaved, since there are two arrays, one input, one output. Slowdown is indeed observed when testing the serial version of the algorithm, when the size of the input array and the output array combined was increased from smaller than L1 to larger than L1, and when it was increased from smaller than L2 to larger than L2.

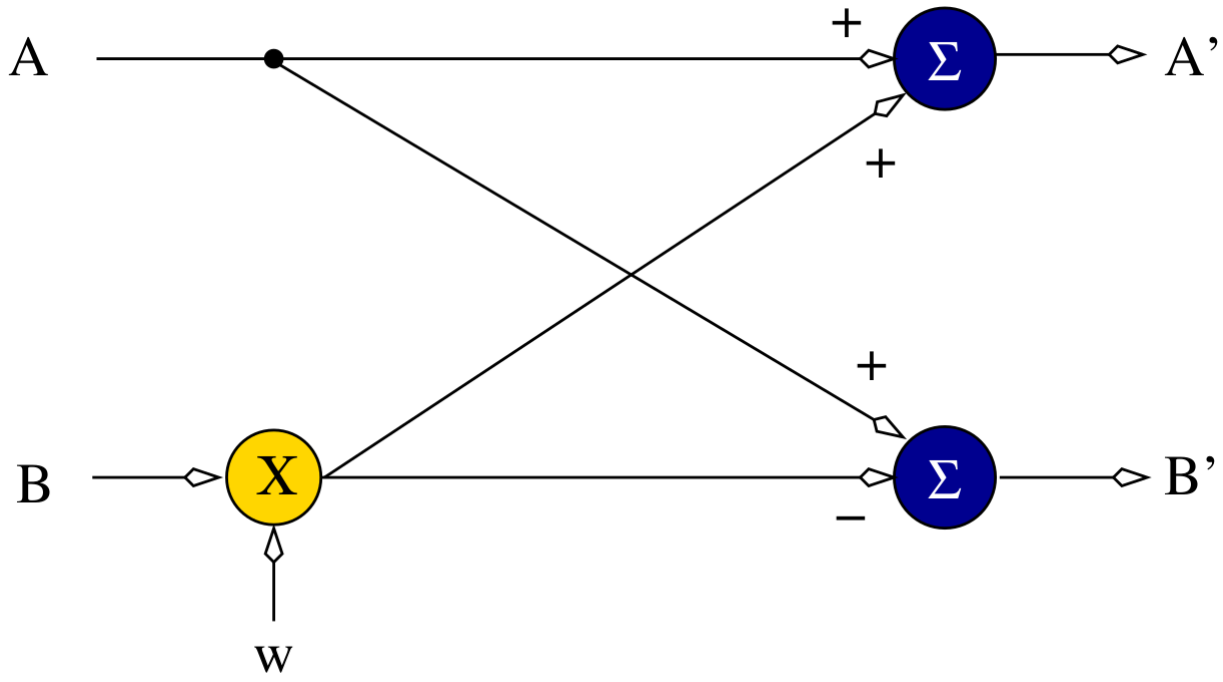
5 Hardware Schematic Explained

There are 4 major components in the FFT accelerator, which are the **Butterfly Unit** (**BFU**), the **Address Generation Unit** (**AGU**), the **In-Accelerator Memory** (**MEM**), and the **Twiddle Factor Read Only Memory** (**Twiddle Factor ROM**) [1].

5.1 FFT Accelerator

5.1.1 BFU

The BFU does the unit operation shown below:



(Fig.1, Mathematic Operations Performed by BFU [1])

A, B, and w are the inputs to the BFU. A' and B' are the outputs of the BFU. All of them are complex numbers each with a real component and an imaginary component.

The mathematical operations are the following:

$$A' = A + Bw$$

$$B' = A - Bw$$

The BFU has a 3-stage design to improve performance.

5.1.2 Twiddle Factor ROM

The Twiddle Factor ROM serves as a look up table for the twiddle factor w that is fed into the BFU. By having the AGU providing the correct index, both the real part and the imaginary part of the corresponding twiddle factor are found and sent to the BFU.

5.1.3 AGU

The AGU is in charge of orchestrating the dataflow throughout each round of FFT operation. There are two complex number outputs from the BFU during each clock cycle. The BFU also requires two complex number inputs during each clock cycle. The AGU generates the corresponding address that is fed into the MEM, which tells the MEM the two addresses of where the two output numbers from the BFU are written into. The two addresses also happen to be where the two input numbers for the BFU were previously stored.

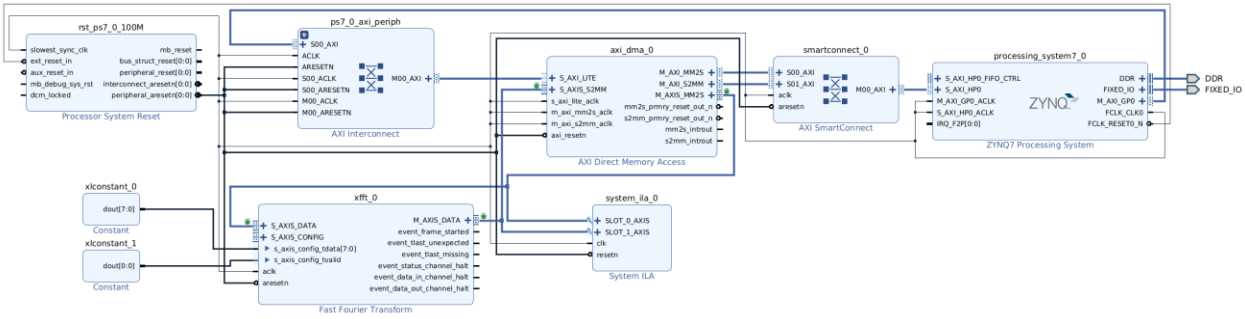
And, as mentioned in the Twiddle Factor ROM section, the AGU is also in charge of generating the index used for finding the correct Twiddle Factor in the Twiddle Factor ROM.

5.1.4 MEM

The MEM is the temporary storage for the intermedia results, which has similar functions as the "$temp_1$" and "$temp_2$" arrays in the serial algorithm. It has two single-precision complex number inputs and two single-precision complex number outputs. It takes two addresses inputs. In

5.2 DMA Design

To migrate data from DDR3 into the MEM before each FFT operation and to transfer the data back, the AXI Direct Memory Access design is chosen.



(Fig. 2, Block Design of the Entire System)

The other design I thought of but not used is to first have the PS transfer data to some Block RAMs in the PL, and have the PS take the processed data back to DDR3 after the FFT operation is done on the PL. Compared to the DMA design, this design requires more involvement of the Cortex-A9 processor, which can lead to performance degradations.

6 Bottleneck Analysis

6.1 Bottleneck Analysis on Serial Algorithm

The **FFT serial algorithm** is heavily compute bound. The **arithmetic intensity** of the algorithm, both the optimized ones and the no-optimization one, is some multiple of $\mathbf{log_2}\ \boldsymbol{N}$.

However, there is another part that is entirely memory bound, which is the **bit-reversal** function that has an **arithmetic intensity** of **0**.

6.2 Bottleneck Analysis on Hardware

The hardware FFT accelerator is also heavily compute bound. If the BFU has a 4-stage design instead of a 3-stage design, higher performance is expected.

The hardware FFT accelerator does not suffer from memory bound though, because the bit-reversal is done by simple wiring at the input, which is maximally parallel.

7 Experiments and Results

Time Taken by Each Optimization Level. Units are all microseconds.					
	Vanilla	+Cache	+ImplicitCache	+Loop Unroll	FPGA
1			4.415385		4.981538
2			4.744616		4.993846
3			4.566154		5.012308
4			4.396923		5.003077
5			4.704616		5.021538
6		4.633846			4.990769
7		4.84			5.224616
8		4.963077			5.003077
9		4.636923			5.04
10		4.566154			4.984615
11	5.123077				5.12
12	5.24				5.390769
13	5.366154				5.123077
14	5.24				5.101539
15	5.443077				5.12
16				5.138462	5.058462
17				4.876923	5.212308
18				5.276923	5.187692
19				4.864615	5.092308
20				5.138462	5.073846
Average	5.282462	4.728	4.565539	5.059077	5.086769
	Worst		Best		

(Table 1, Time Taken by Each Optimization Level)

The level of optimization from the left side of Table 1 to the right side of Table 1 is incremental, except for the FPGA column. For example, the "+Cache" column means results collected from running the "Vanilla" version serial code described in section 3.1 with the caching for performance optimization described in section 3.2. And the "+ImplicitCache" column means results collected from running the serial code with optimizations described in section 3.2 and 3.3 combined.

For each level of serial code optimization, 5 benchmark runs were performed. The program can time both the serial version and the FPGA accelerated version. Thus, there are 5 results for each level of serial code optimization and 20 results for the FPGA accelerated version. Their average time in microseconds are computed and noted at the bottom of the table.

8 Conclusion and Future Improvements

As the data in the previous section shows, the FPGA acceleration can outperform the serial algorithm without the caching and implicit caching optimization. However, with the two optimizations applied on the serial algorithm, the FPGA acceleration falls behind.

Loop unrolling appears to be a deoptimization in this case. Adding it will degrade the performance by 9.76%.

There is currently a limitation with the FPGA acceleration code. **It can only perform the FFT operation once per program run.** Having it perform FFT operations repeatedly by stepping through an array of data point sets will lead to program crash. Thus, the first future improvement will be to enable the step-through-array capability. Then it will be possible to compare the step-through-array performance of the serial algorithm and the FPGA acceleration.

Also for future improvements, multithreading, NEON DSP/FPU Engine, 4-stage BFU design, and having multiple FFT Accelerator instances can be explored to see which one gives the highest performance on the ArtyZ7-20 platform.

9 Appendix

9.1 Code (without loop unrolling deoptimization)

```
/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include <stdbool.h>
#include "xaxidma.h"
#include "platform.h"
#include "xparameters.h"
#include <stdlib.h>
#include <unistd.h>
#include <complex.h>
#include <time.h>
```

```c
#include <xtime_l.h>
#include <math.h>

// Parameters for Data
#define ARR_LEN 8

// Parameters for Algorithm
#define N 8 // Must be power of 2
#define LEVEL 3 // Please set LEVEL to log2(N)
const int bitRevIndex[N] = {0, 4, 2, 6, 1, 5, 3, 7};
const float complex W[N/2] = {1-0*I, 0.7071067811865476-0.7071067811865475*I,
0.0-1*I, -0.7071067811865475-0.7071067811865476*I};

// Global Variables
XAxiDma AxiDma;

// Function headers
void initialize_customWave(float complex arr[ARR_LEN]);
void bitReverse(float complex dataIn[ARR_LEN], float complex dataOut[ARR_LEN]);
void FFT_PS(float complex Data_in[ARR_LEN], float complex Data_out[ARR_LEN]);
int init_DMA();
u32 checkIdle(u32 baseAddress, u32 offset); // This function is defined to check
whether DMA is busy or not.
                                            // offset = 0x4  to check for DMA to
FIFO transaction channel.
                                            // offset = 0x34 to check for FIFO to
MDA transaction channel.

int main()
{
    init_platform();

    printf("---Start of Program---\n");

    if (ARR_LEN % N)
    {
        printf("ARR_LEN must be a multiple of N!\n");
        printf("Terminating program...\n");
        return 0;
    }

    // Timing Variables
    XTime tProcessorStart, tProcessorEnd;
    XTime tFPGAstart, tFPGAend;
```

```
    // Array Variables
    float complex Data_in[ARR_LEN];
    float complex Data_rev[ARR_LEN];
    float complex Data_PSout[ARR_LEN];
    float complex Data_PLout[ARR_LEN];

    // Array Initialization
    printf("Start custom wave initialization.\n");
    initialize_customWave(Data_in);
    printf("Custom wave initialization done.\n");

    // Benchmarks
    printf("Benchmarking CPU (single thread)...\n");
    XTime_GetTime(&tProcessorStart);
    bitReverse(Data_in, Data_rev);
    FFT_PS(Data_rev, Data_PSout);
    XTime_GetTime(&tProcessorEnd);
    printf("CPU (single thread) benchmark finished.\n");

    int status_dma = init_DMA();
    if (status_dma != XST_SUCCESS)
    {
        printf("my_error: Can't initialize DMA\n");
        return XST_FAILURE;
    }

    // Flush Cache to Write Back the Data in DDR
    Xil_DCacheFlushRange((UINTPTR)Data_in, (sizeof(float complex)*ARR_LEN));
    Xil_DCacheFlushRange((UINTPTR)Data_PLout, (sizeof(float complex)*ARR_LEN));

    int status, status_transfer;
    printf("Benchmarking FPGA...\n");
    XTime_GetTime(&tFPGAstart);
    // status = 0 and status = 2 both indicate idle
//      printf("DMA status before transfer\n");
//      printf("DMA to Device: %d\n", checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x4));
//      printf("Device to DMA: %d\n", checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x34));
    status_transfer = XAxiDma_SimpleTransfer(
            &AxiDma,
            (UINTPTR)Data_PLout,
            sizeof(float complex)*ARR_LEN,
            XAXIDMA_DEVICE_TO_DMA
            );
    if (status_transfer != XST_SUCCESS)
    {
```

```
        printf("my_error: Write data to PL via DMA failed\n");
        return 0;
    }
    // status = 0 and status = 2 both indicate idle
//      printf("DMA status between transfer\n");
//      printf("DMA to Device: %d\n", checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x4));
//      printf("Device to DMA: %d\n", checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x34));

    status_transfer = XAxiDma_SimpleTransfer(
            &AxiDma,
            (UINTPTR)Data_in,
            sizeof(float complex)*ARR_LEN,
            XAXIDMA_DMA_TO_DEVICE
            );
    if (status_transfer != XST_SUCCESS)
    {
        printf("my_error: Read data from PL via DMA failed\n");
        return 0;
    }
    // status = 0 and status = 2 both indicate idle
//      printf("DMA status after transfer\n");
//      printf("DMA to Device: %d\n", checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x4));
//      printf("Device to DMA: %d\n", checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x34));

    do {
        status = checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x4); // DMA to device
    } while (status != 2);
    do {
        status = checkIdle(XPAR_AXI_DMA_0_BASEADDR, 0x34); // device to DMA
    } while (status != 2);
    XTime_GetTime(&tFPGAend);
    printf("FPGA benchmark finished.\n");

    printf("Comparing results from CPU and FPGA...\n");
    float diff_r, diff_i;
    bool err_flag = false;
    int j;
    for (j = 0; j < ARR_LEN; j++)
    {
//      printf("CPU: %f + %f i    FPGA: %f + %f i\n",
//              creal(Data_PSout[j]),
//              cimag(Data_PSout[j]),
//              creal(Data_PLout[j]),
//              cimag(Data_PLout[j])
//              );
```

```
        diff_i = abs(creal(Data_PSout[j]) - creal(Data_PLout[j]));
        diff_r = abs(cimag(Data_PSout[j]) - cimag(Data_PLout[j]));
        if (diff_r > 0.001 || diff_i > 0.001)
        {
            err_flag = true;
            break;
        }
    }

    if (err_flag)
    {
        printf("Data mismatch found at %d.\n", j);
        printf("Result from CPU : %f + %f i\n", creal(Data_PSout[j]),
cimag(Data_PSout[j]));
        printf("Result from FPGA: %f + %f i\n", creal(Data_PLout[j]),
cimag(Data_PLout[j]));
    }
    else
    {
        printf("Results from CPU and FPGA all match!\n");
    }

    printf("---Timing---\n");
    float time_diff;
    time_diff = (float)1.0 * (tProcessorEnd - tProcessorStart) /
(COUNTS_PER_SECOND/1000000);
    printf("CPU: %f\n", time_diff);
    time_diff = (float)1.0 * (tFPGAend - tFPGAstart) /
(COUNTS_PER_SECOND/1000000);
    printf("FPGA: %f\n", time_diff);

    printf("---End of Program---\n");
    cleanup_platform();
    return 0;
}

void initialize_customWave(float complex arr[ARR_LEN])
{
    for (int i = 0; i < ARR_LEN; i+=N)
    {
        arr[i  ] = 11+23*I;
        arr[i+1] = 32+10*I;
        arr[i+2] = 91+94*I;
        arr[i+3] = 15+69*I;
        arr[i+4] = 47+96*I;
```

```
        arr[i+5] = 44+12*I;
        arr[i+6] = 96+17*I;
        arr[i+7] = 49+58*I;
    }
}

void bitReverse(float complex dataIn[ARR_LEN], float complex dataOut[ARR_LEN])
{
    int j;
    for (int i = 0; i < ARR_LEN; i+=N)
    {
        for (j = 0; j < N; j++)
        {
            dataOut[i+j] = dataIn[i+bitRevIndex[j]];
        }
    }
}

void FFT_PS(float complex Data_in[ARR_LEN], float complex Data_out[ARR_LEN])
{
    int j, k;
    float complex temp1[N], temp2[N];
    float complex* Data_in_wOffset;
    float complex* Data_out_wOffset;
    float complex tempComplex;
    for (int i = 0; i < ARR_LEN; i+=N)
    {
        Data_in_wOffset = Data_in + i;
        Data_out_wOffset = Data_out + i;
        for (j = 0; j < N; j+=2)
        {
            temp1[j] = Data_in_wOffset[j] + Data_in_wOffset[j+1];
            temp1[j+1] = Data_in_wOffset[j] - Data_in_wOffset[j+1];
        }

        for (j = 0; j < N; j+=4)
        {
            for (k = 0; k < 2; ++k)
            {
                tempComplex = W[2*k]*temp1[j+k+2];
                temp2[j+k] = temp1[j+k] + tempComplex;
                temp2[j+k+2] = temp1[j+k] - tempComplex;
            }
        }
```

```
        for (j = 0; j < N/2; j++)
        {
            tempComplex = W[j]*temp2[j+4];
            Data_out_wOffset[j] = temp2[j] + tempComplex;
            Data_out_wOffset[j+4] = temp2[j] - tempComplex;
        }
    }
}

int init_DMA()
{
    XAxiDma_Config* CfgPtr;
    int status;

    CfgPtr = XAxiDma_LookupConfig(XPAR_AXI_DMA_0_DEVICE_ID);
    if (!CfgPtr)
    {
        printf("my_error: No config found for %d\n", XPAR_AXI_DMA_0_DEVICE_ID);
        return XST_FAILURE;
    }

    status = XAxiDma_CfgInitialize(&AxiDma, CfgPtr);
    if (status != XST_SUCCESS)
    {
        printf("my_error: DMA Initialization Failed. Return Status: %d\n",
status);
        return XST_FAILURE;
    }
    if (XAxiDma_HasSg(&AxiDma)) // Check that DMA isn't in Scatter Gather Mode.
    {
        printf("my_error: Device should not be in Scatter Gather Mode\n");
        return XST_FAILURE;
    }

    return XST_SUCCESS;
}

u32 checkIdle(u32 baseAddress, u32 offset)
{
    u32 status;

    status = (XAxiDma_ReadReg(baseAddress, offset)) & XAXIDMA_IDLE_MASK;

    return status;
}
```

9.2 Works Cited

[1] Slate, George. “Fast Fourier Transform in Hardware: A Tutorial Based on an FPGA Implementation.” *ResearchGate*, May 20, 2014, https://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf

[2] Bhard, et. al. “Lab_11_Part_1: DMA and FFT in Zynq SoC.” *YouTube*, Dec. 7, 2021, https://www.youtube.com/watch?v=NjNVnrorvPU&list=PL579fbjB-a0u7ilbp5173Ulm-RJelsHtR&index=30

[3] Bhard, et. al. “Lab_11_Part_2: DMA and FFT in Zynq SoC.” *YouTube*, Dec. 7, 2021, https://www.youtube.com/watch?v=WUbpPsD_8zw&list=PL579fbjB-a0u7ilbp5173Ulm-RJelsHtR&index=31

[4] Bhard, et. al. “Lab_11_Part_3: DMA and FFT in Zynq SoC.” *YouTube*, Dec. 22, 2021, https://www.youtube.com/watch?v=6OU0ASEIUJg&list=PL579fbjB-a0u7ilbp5173Ulm-RJelsHtR&index=32