UCSD 237C: Project 4 FFT

Steven Daniels

sdaniels@ucsd.edu

Student ID# A53328625

December 2, 2023

1. Introduction

This report details the optimizations performed on a hardware implementation of the Fast Fourier Transform in HLS. One optimal design was explored:

1. FFT using dataflow for each stage.

2. FFT Design

2.1. Throughput

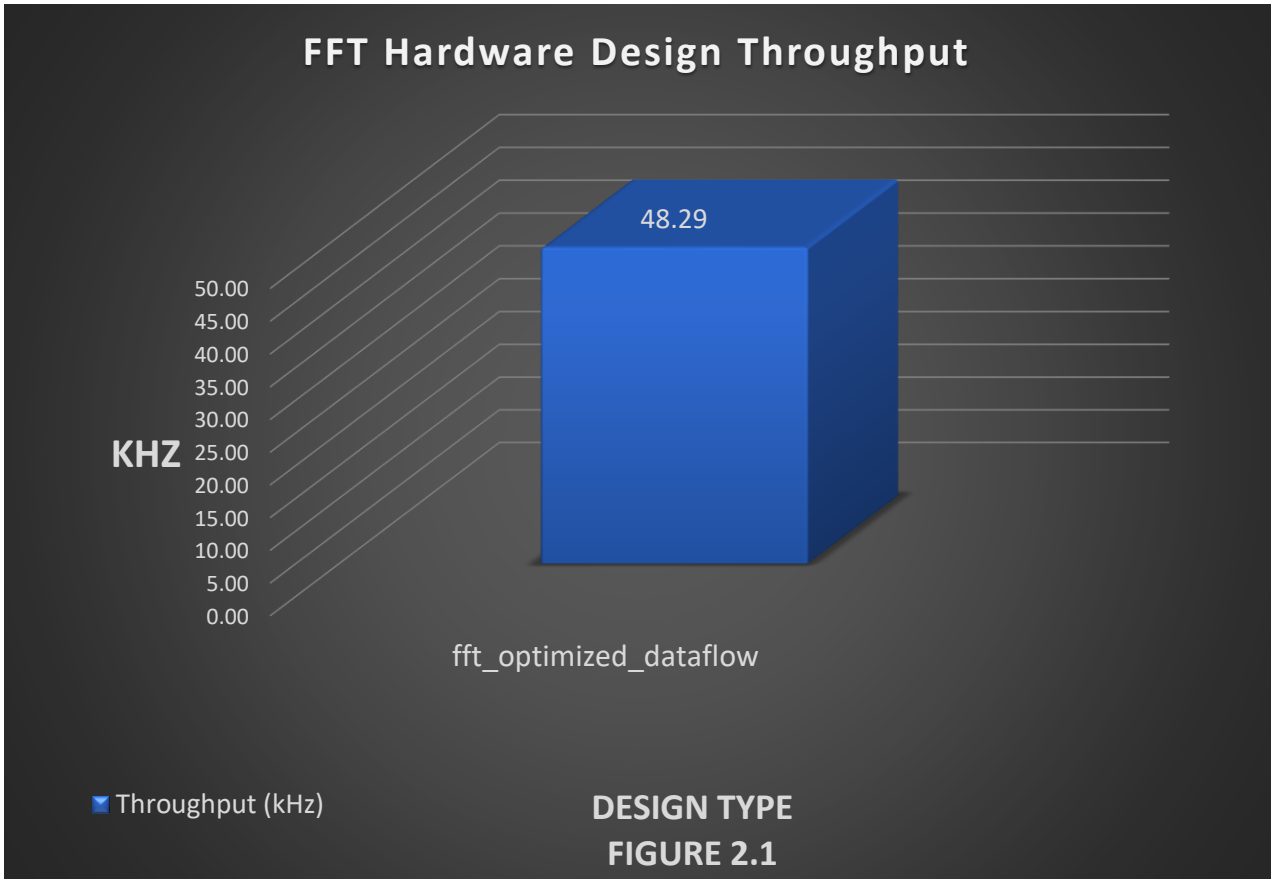

FIGURE 2.1

2.2. Implementation

```
void fft(DTYPE IN_R[SIZE], DTYPE IN_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
#pragma HLS dataflow
    DTYPE X_R[SIZE], X_I[SIZE];

    bit_reverse(IN_R,IN_I,X_R, X_I);

    //Call fft
    DTYPE Stage1_R[SIZE], Stage1_I[SIZE];
    DTYPE Stage2_R[SIZE], Stage2_I[SIZE];
    DTYPE Stage3_R[SIZE], Stage3_I[SIZE];
    DTYPE Stage4_R[SIZE], Stage4_I[SIZE];
    DTYPE Stage5_R[SIZE], Stage5_I[SIZE];
    DTYPE Stage6_R[SIZE], Stage6_I[SIZE];
    DTYPE Stage7_R[SIZE], Stage7_I[SIZE];
    DTYPE Stage8_R[SIZE], Stage8_I[SIZE];
    DTYPE Stage9_R[SIZE], Stage9_I[SIZE];

    fft_stage_first(X_R, X_I, Stage1_R, Stage1_I);
    fft_stages(Stage1_R, Stage1_I, 2, Stage2_R, Stage2_I);
    fft_stages(Stage2_R, Stage2_I, 3, Stage3_R, Stage3_I);
    fft_stages(Stage3_R, Stage3_I, 4, Stage4_R, Stage4_I);
    fft_stages(Stage4_R, Stage4_I, 5, Stage5_R, Stage5_I);
    fft_stages(Stage5_R, Stage5_I, 6, Stage6_R, Stage6_I);
    fft_stages(Stage6_R, Stage6_I, 7, Stage7_R, Stage7_I);
    fft_stages(Stage7_R, Stage7_I, 8, Stage8_R, Stage8_I);
    fft_stages(Stage8_R, Stage8_I, 9, Stage9_R, Stage9_I);
    fft_stage_last(Stage9_R, Stage9_I, OUT_R, OUT_I);

}


void bit_reverse(DTYPE X_R[SIZE], DTYPE X_I[SIZE],DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
    unsigned int reversed= 0;
    unsigned int i;
    DTYPE temp;
    DTYPE temp_rev_R[SIZE];
    DTYPE temp_rev_I[SIZE];
    int temp_rev_idx[SIZE];

    for (i = 0; i < SIZE; i++)
    {
#pragma HLS UNROLL
        temp_rev_idx[i] = reverse_bits(i); // Find the bit reversed index

        // Swap the real values
        OUT_R[i]      = X_R[temp_rev_idx[i]];
        temp_rev_R[i] = X_R[i];

        // Swap the imaginary values
        OUT_I[i] = X_I[temp_rev_idx[i]];
        temp_rev_I[i] = X_I[i];
    }


    for (i = 0; i < SIZE; i++)
    {
#pragma HLS UNROLL
        // Swap the real values
        OUT_R[temp_rev_idx[i]] = temp_rev_R[i];

        // Swap the imaginary values
        OUT_I[temp_rev_idx[i]] = temp_rev_I[i];
    }
}
```

```
/*======================BEGIN: FFT=========================*/
//stage 1
void fft_stage_first(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE])
{
#pragma HLS array_partition variable=W_real type=complete
#pragma HLS array_partition variable=W_imag type=complete

    int stage = 1;
    int DFTpts = 1 << stage; // DFT = 2^stage = points in sub DFT
    int numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
    int step = SIZE >> stage;
    int k = 0;
    DTYPE e = -6.283185307178 / DFTpts;
    DTYPE a = 0.0;
    DTYPE c;
    DTYPE s;
    int twiddle_index = 0;

    // Perform butterflies for j-th stage
    butterfly_loop_first:for (int j = 0; j < numBF; j++)
    {
        // Compute butterflies that use same W**k
        dft_loop_first:for (int i = j; i < SIZE; i += DFTpts) {
#pragma HLS pipeline
                twiddle_index = (k*j)%(DFTpts); // determines which twiddle factor is
selected
                c = W_real[twiddle_index]; // twiddle factor
                s = W_imag[twiddle_index]; // twiddle factor

                int ilower = i + numBF; // index of lower point in butterfly

                DTYPE temp_R = X_R[ilower] * c - X_I[ilower] * s;
                DTYPE temp_I = X_I[ilower] * c + X_R[ilower] * s;

                OUT_R[ilower] = X_R[i] - temp_R;
                OUT_I[ilower] = X_I[i] - temp_I;

                OUT_R[i] = X_R[i] + temp_R;
                OUT_I[i] = X_I[i] + temp_I;
            }
        k += step;
    }
}

//stages
void fft_stages(DTYPE X_R[SIZE], DTYPE X_I[SIZE], int stage, DTYPE OUT_R[SIZE], DTYPE
OUT_I[SIZE]) {
#pragma HLS array_partition variable=W_real type=complete
#pragma HLS array_partition variable=W_imag type=complete
    int DFTpts = 1 << stage; // DFT = 2^stage = points in sub DFT
    int numBF = DFTpts / 2; // Butterfly WIDTHS in sub-DFT
    int step = SIZE >> stage;
    int k = 0;
    DTYPE e = -6.283185307178 / DFTpts;
    DTYPE a = 0.0;
    DTYPE c = 0;
    DTYPE s = 0;
    int twiddle_index = 0;

    // Perform butterflies for j-th stage
    butterfly_loop_stages:for (int j = 0; j < numBF; j++)
    {
        // Compute butterflies that use same W**k
        dft_loop_stages:for (int i = j; i < SIZE; i += DFTpts) {
#pragma HLS pipeline
                twiddle_index = (k*j)%(DFTpts); // determines which twiddle factor is
selected
```

```
                c = W_real[twiddle_index]; // twiddle factor
                s = W_imag[twiddle_index]; // twiddle factor

                int ilower = i + numBF; // index of lower point in butterfly

                DTYPE temp_R = X_R[ilower] * c - X_I[ilower] * s;
                DTYPE temp_I = X_I[ilower] * c + X_R[ilower] * s;

                OUT_R[ilower] = X_R[i] - temp_R;
                OUT_I[ilower] = X_I[i] - temp_I;

                OUT_R[i] = X_R[i] + temp_R;
                OUT_I[i] = X_I[i] + temp_I;
            }
        k += step;
    }
}


//last stage
void fft_stage_last(DTYPE X_R[SIZE], DTYPE X_I[SIZE], DTYPE OUT_R[SIZE], DTYPE OUT_I[SIZE]) {
#pragma HLS array_partition variable=W_real type=complete
#pragma HLS array_partition variable=W_imag type=complete
    int stage = 10;
    int DFTpts = 1 << stage;  // same as multiplying 1* 2^stage, DFT = 2^stage = points in
sub DFT
    int numBF = DFTpts / 2;   // Butterfly WIDTHS in sub-DFT
    int step = SIZE >> stage; // same as dividing SIZE/(2^stage)
    int k = 0;
    DTYPE e = -6.283185307178 / DFTpts;
    DTYPE c = 0;
    DTYPE s = 0;
    int twiddle_index = 0;

    // Perform butterflies for j-th stage
    butterfly_loop_last:for (int j = 0; j < numBF; j++)
    {
        // Compute butterflies that use same W**k
        dft_loop_last:for (int i = j; i < SIZE; i += DFTpts) {
#pragma HLS pipeline
                twiddle_index = (k*j)%(DFTpts); // determines which twiddle factor is
selected
                c = W_real[twiddle_index]; // twiddle factor
                s = W_imag[twiddle_index]; // twiddle factor
                int ilower = i + numBF; // index of lower point in butterfly
                DTYPE temp_R = X_R[ilower] * c - X_I[ilower] * s;
                DTYPE temp_I = X_I[ilower] * c + X_R[ilower] * s;
                OUT_R[ilower] = X_R[i] - temp_R;
                OUT_I[ilower] = X_I[i] - temp_I;
                OUT_R[i] = X_R[i] + temp_R;
                OUT_I[i] = X_I[i] + temp_I;
            }
        k += step;
    }
}
/*======================END: FFT=========================*/
```

2.3. Resources

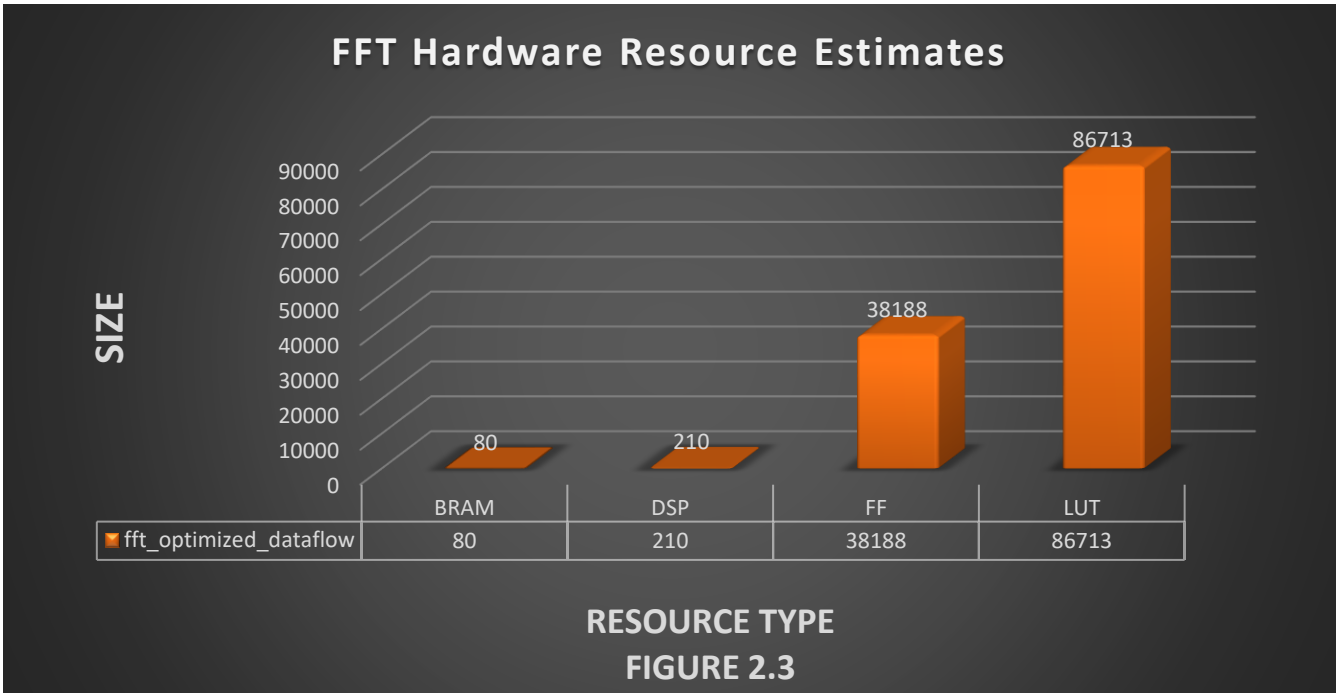

FIGURE 2.3

2.4. Optimizations

- Added dataflow to top level function.
- Separated loop in bit reverse function into 2 loops and completely unrolled both loops.
- Completely partitioned the twiddle factors.

2.5. Analysis

This design provides an optimized version of the Fast Fourier Transform and achieves near target performance of 48Khz of throughput. The dataflow pragma was used to ensure that the FFT stages overlap thereby making use of task pipelining. Additional performance gains were achieved by completely partitioning the precomputed twiddle factors used in the inner loop of the FFT algorithm. The primary tradeoffs of this design were in the usage of hardware resources where a large amount of DSP's and Flip Flops were used.