UCSD 237C: Project 3 DFT

Steven Daniels

sdaniels@ucsd.edu

Student ID# A53328625

November 15, 2023

1. Introduction

The Discrete Fourier Transform (DFT) is a common operation in signal processing which generates a discrete frequency domain representation of the discrete input signal. This report presents various hardware designs for the DFT. The root mean square error of CORDIC output for "R" and "Theta" is used to confirm its accuracy against a set of expected values. The purpose of this report is to show the trade-offs of varying different hardware parameters when determining the best design for a 256-point DFT and a 1024-point DFT hardware implementation. Six different designs were explored:

1. DFT baseline.
2. DFT that uses lookup table for sin() and cos().
3. DFT with Separate Arrays for Inputs and Outputs
4. DFT with Loop Unroll Factor of 16
5. DFT with Block Array Partitioning Factor of 16
6. DFT with Cyclic Array Partitioning Factor of 16
7. DFT with Loop unrolling Factor 16 and Cyclic Array Partitioning Factor 16
8. DFT with Dataflow
9. DFT using HLS streaming interface
10. Best Architecture: pick your "best" 256 architecture and synthesize that as a 1024 DFT.

Each section of the report seeks to answer a different question about the different hardware implementations of the DFT.

2. DFT Designs

2.1. Calculating Throughput

Throughput for a DFT is calculated by measuring the number of DFT operations per second and converting that number to the number of cycles per second. Design without pipeline optimizations were calculated as follows:

$$Throughput(kHz) = 1000000/(ClockPeriod(ns) * \#\ LatencyCycles)$$

2.2. Baseline

2.2.1. Throughput Max: 40 Hz

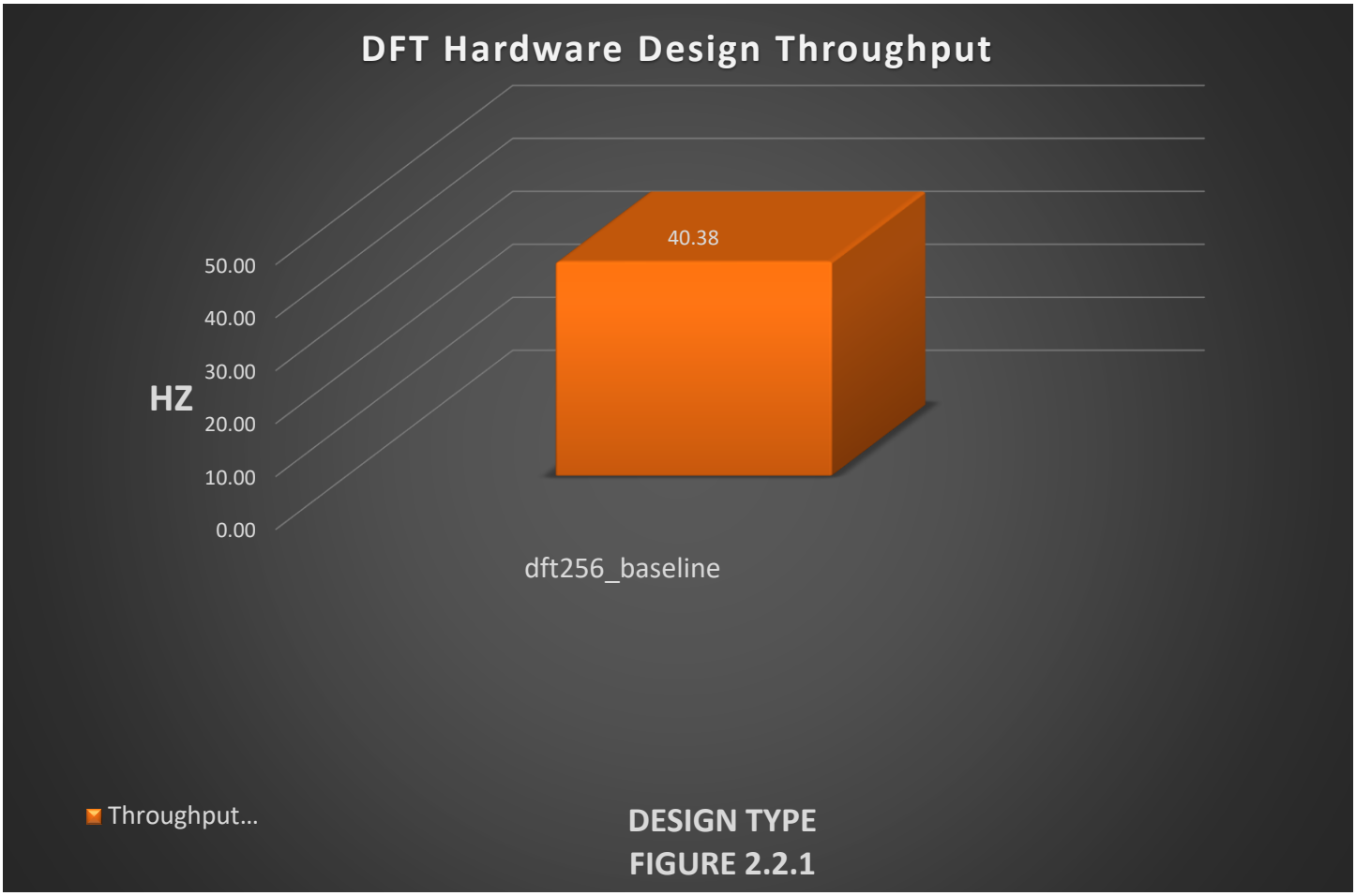


FIGURE 2.2.1

2.2.2. Implementation

```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    int i, j;
    DTYPE w;
    DTYPE c, s;
    // Temporary arrays to hold the intermediate frequency domain results
    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
    // Calculate each frequency domain sample iteratively
    for (i = 0; i < SIZE; i += 1) {
    temp_real[i] = 0;
    temp_imag[i] = 0;
    // (2 * pi * i)/N
    w = (2.0 * 3.141592653589 / SIZE) * (DTYPE)i;
    // Calculate the jth frequency sample sequentially
    for (j = 0; j < SIZE; j += 1) {
        // Utilize HLS tool to calculate sine and cosine values
        c = cos(j * w);
        s = -sin(j * w);
        // Multiply the current phasor with the appropriate input sample and keep
        // running sum
        temp_real[i] += (real_sample[j] * c - imag_sample[j] * s);
        temp_imag[i] += (real_sample[j] * s + imag_sample[j] * c);
    }
    }
    // Perform an inplace DFT, i.e., copy result into the input arrays
    for (i = 0; i < SIZE; i += 1)
```

```
    {
        real_sample[i] = temp_real[i];
        imag_sample[i] = temp_imag[i];
    }
}
```

2.2.3. Optimizations

None

2.2.4. Resources

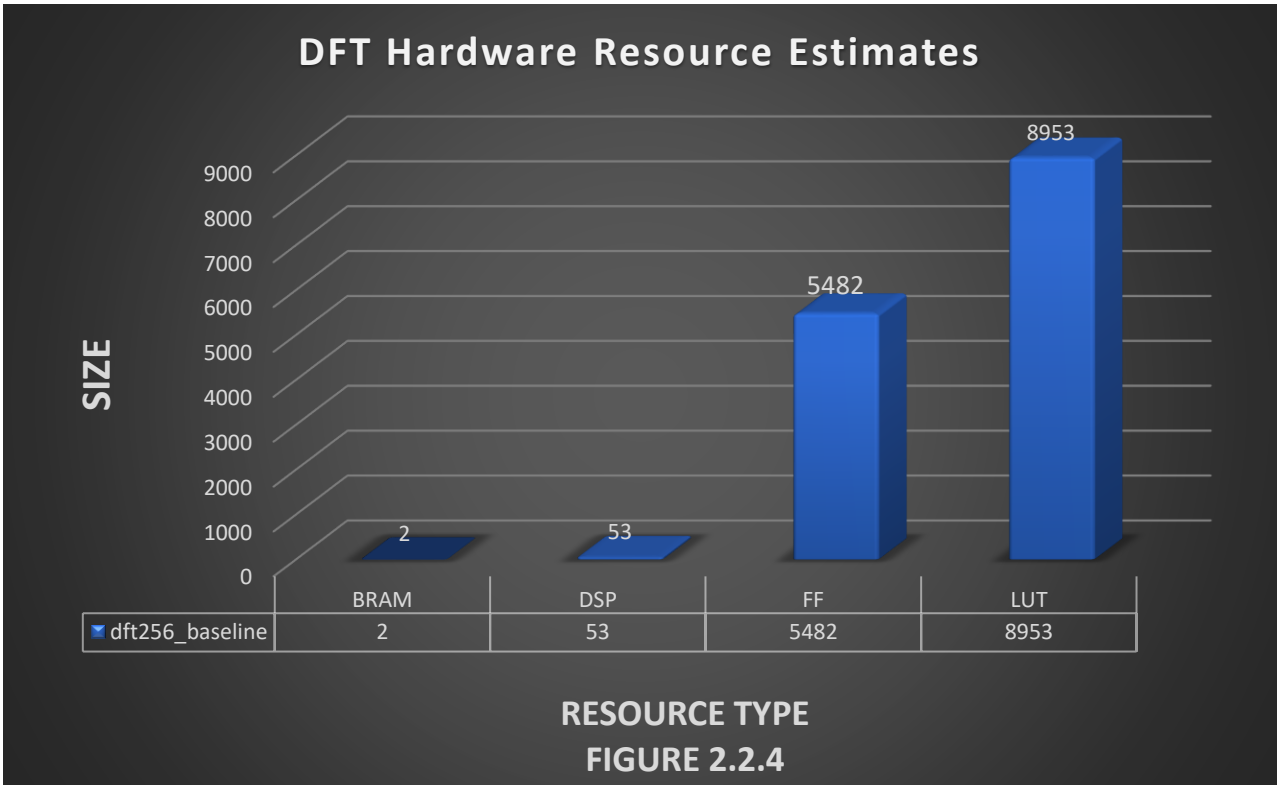


FIGURE 2.2.4

2.2.5. Analysis

This section presents a baseline implementation of the DFT algorithm using floating variables as the data type and no optimizations. This implementation returns 256 samples.

2.3. DFT with LUT for sin() and cos()

2.3.1. Throughput Max: 131 Hz

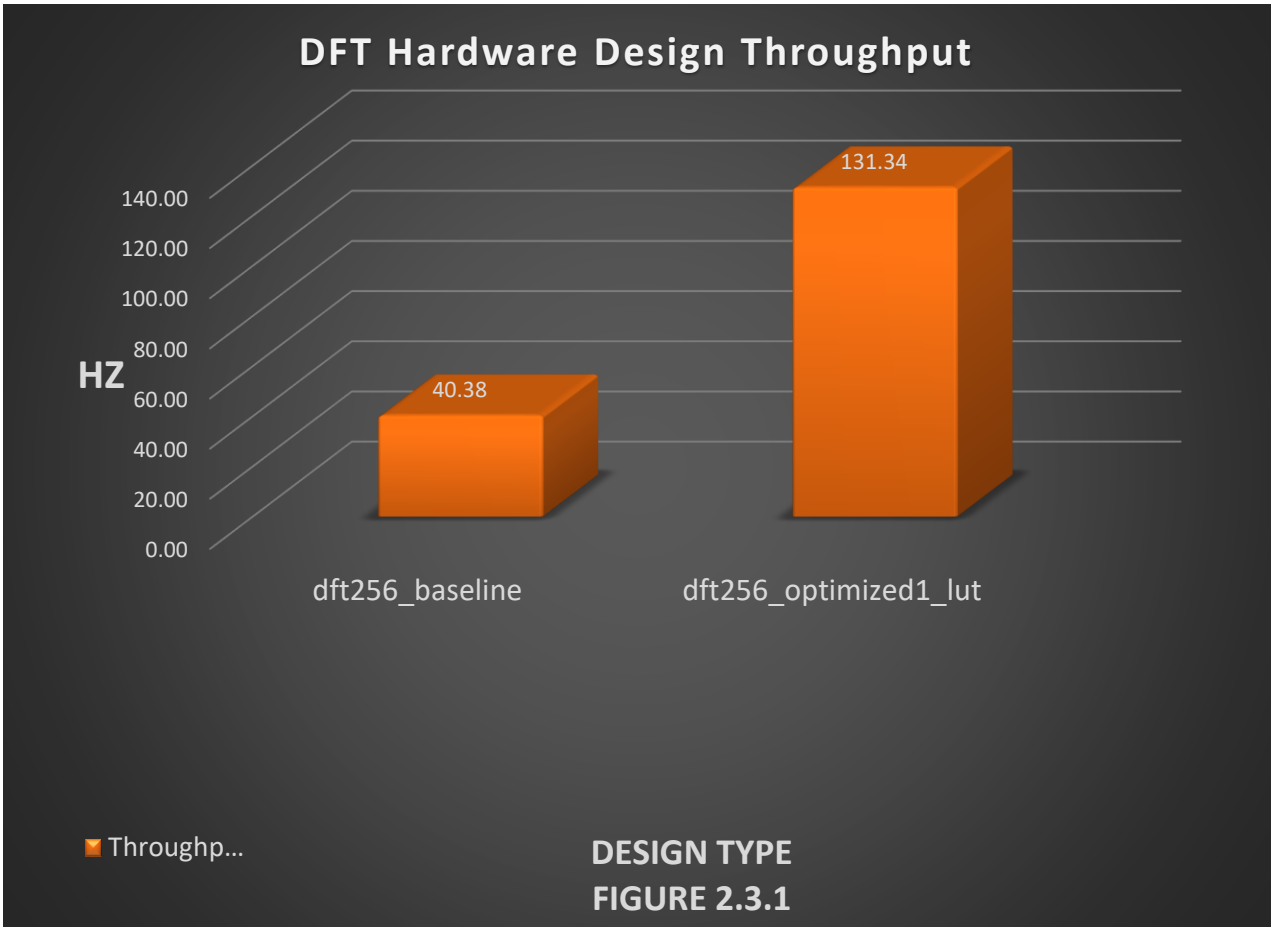


FIGURE 2.3.1

2.3.2. Implementation

```
void dft(data_t real_sample[SIZE], data_t imag_sample[SIZE])
{
    int i, j;
    ap_uint<8> index;
    data_t w;
    data_t c, s;

    data_t temp_real[SIZE];
    data_t temp_imag[SIZE];

    for (i = 0; i < SIZE; i += 1)
    {
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample[j] * c - imag_sample[j] * s);
            temp_imag[i] += (real_sample[j] * s + imag_sample[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
```

```
            real_sample[i] = temp_real[i];
            imag_sample[i] = temp_imag[i];
        }
}
```

2.3.3. Optimizations

2.3.3.1. Replaced baseline sin() and cos() calls with a lookup table.

2.3.4. Resources

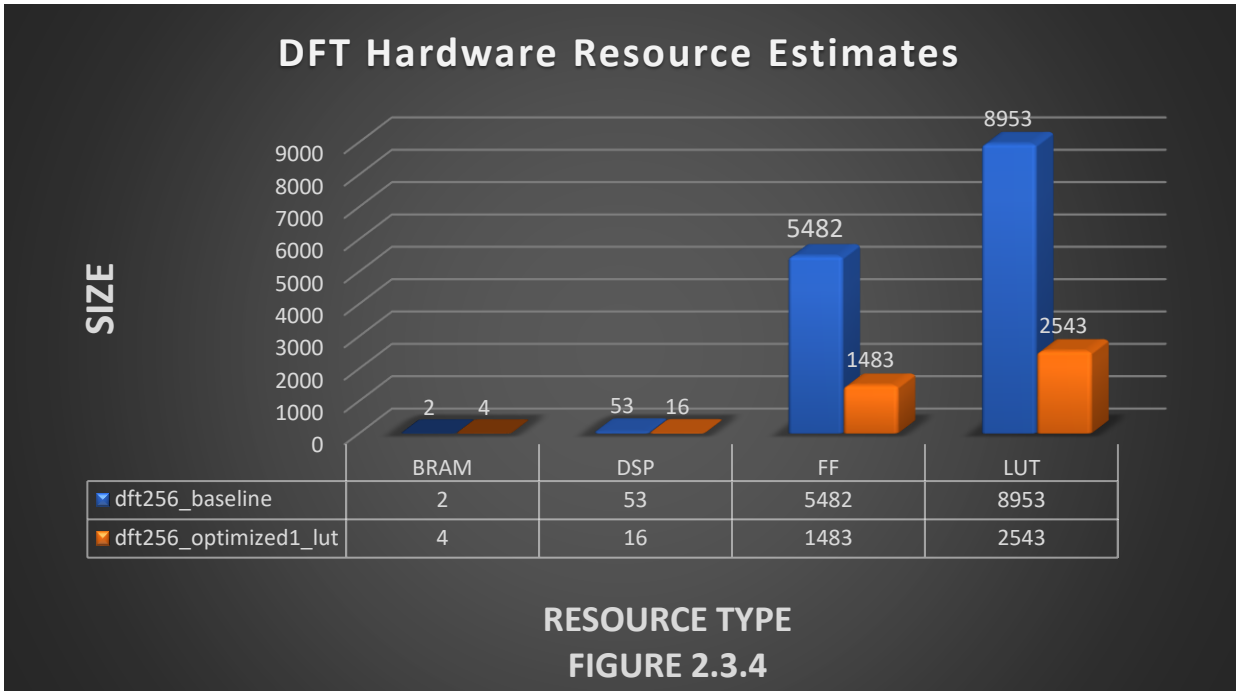


FIGURE 2.3.4

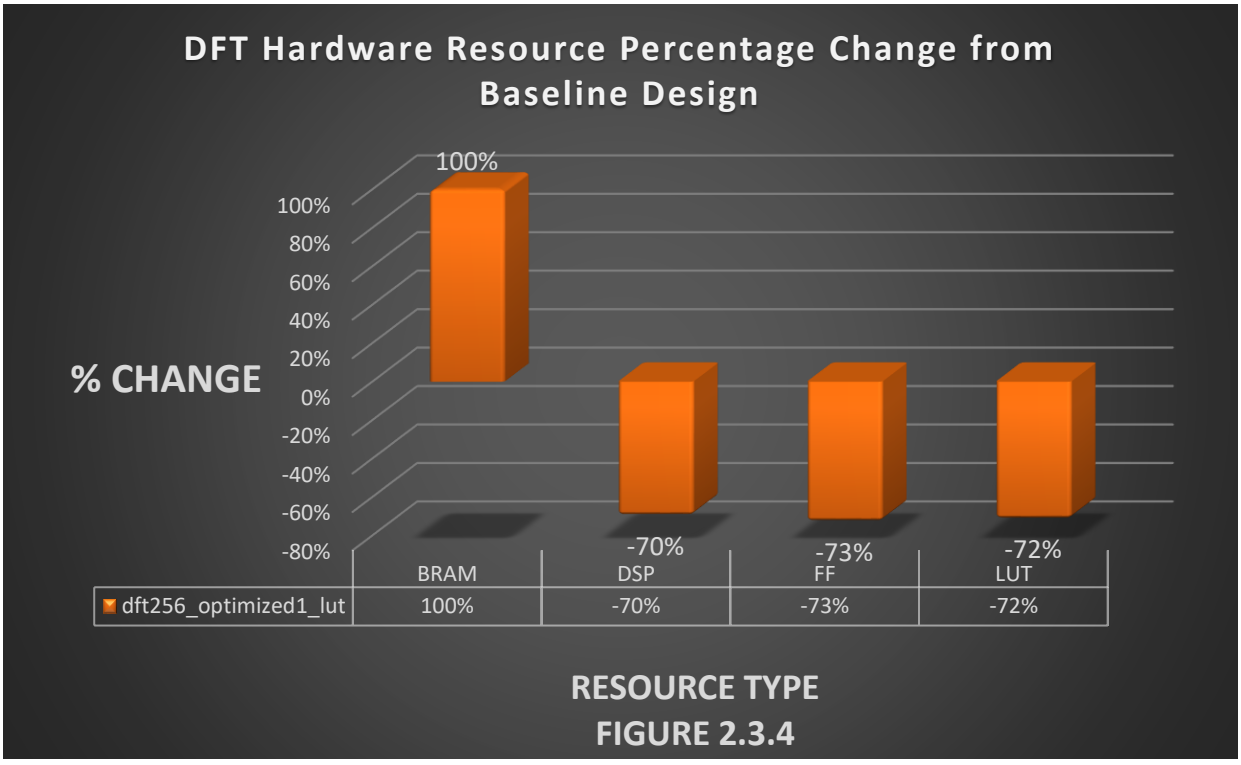


FIGURE 2.3.4

2.3.5. Analysis

This section presents a DFT architecture that utilizes a lookup table and demonstrates the significant resource reduction that can be achieved by computing the sin() and cos() values ahead of time.

2.4. DFT with Separate Arrays for Inputs and Outputs

2.4.1. Throughput Max: 131 Hz

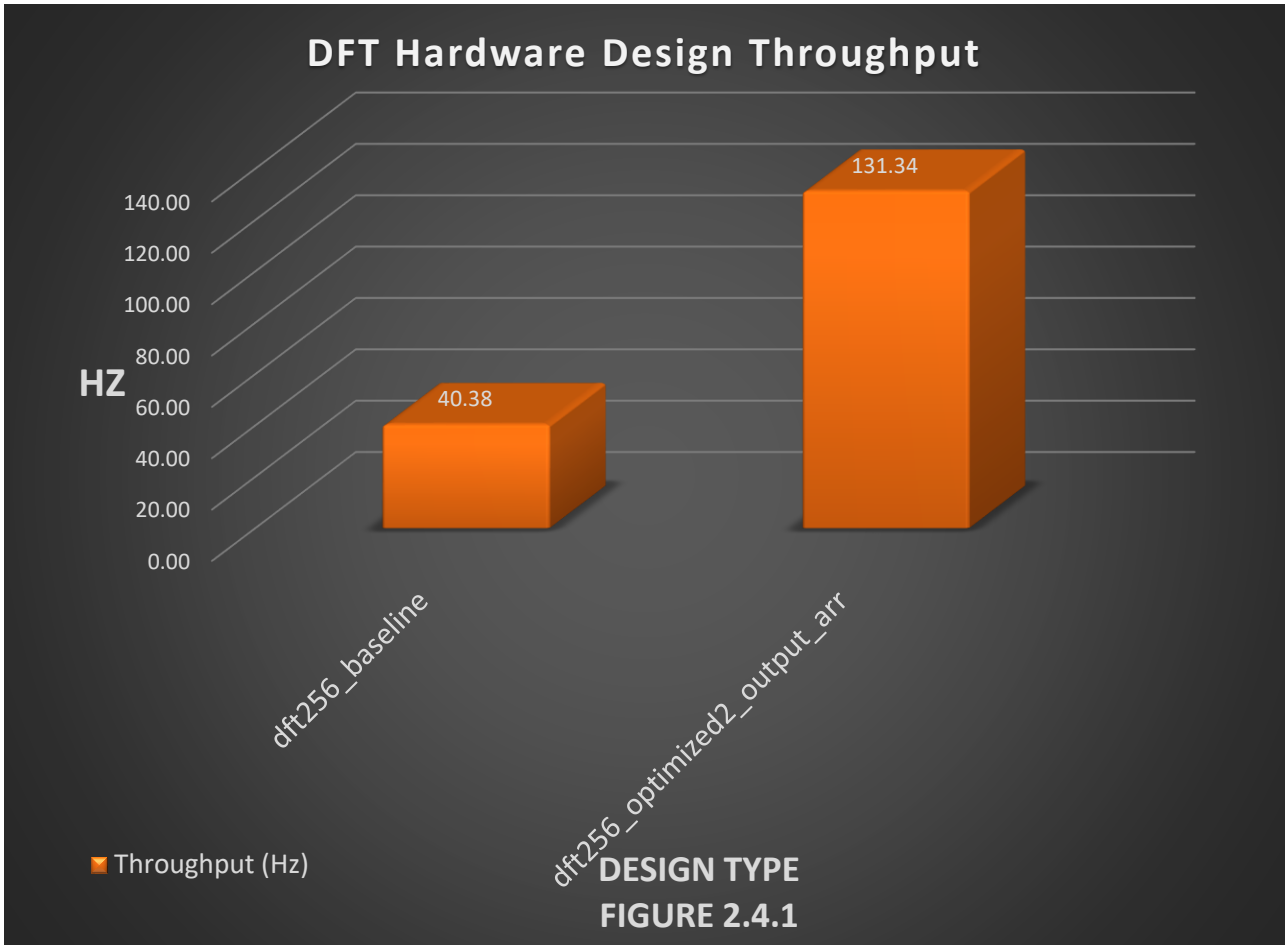


FIGURE 2.4.1

2.4.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];

    for (i = 0; i < SIZE; i += 1)
    {
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];
```

```
            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.4.3. Optimizations

- Separate the input array into two separate arrays. One for input and one for output.

2.4.4. Resources

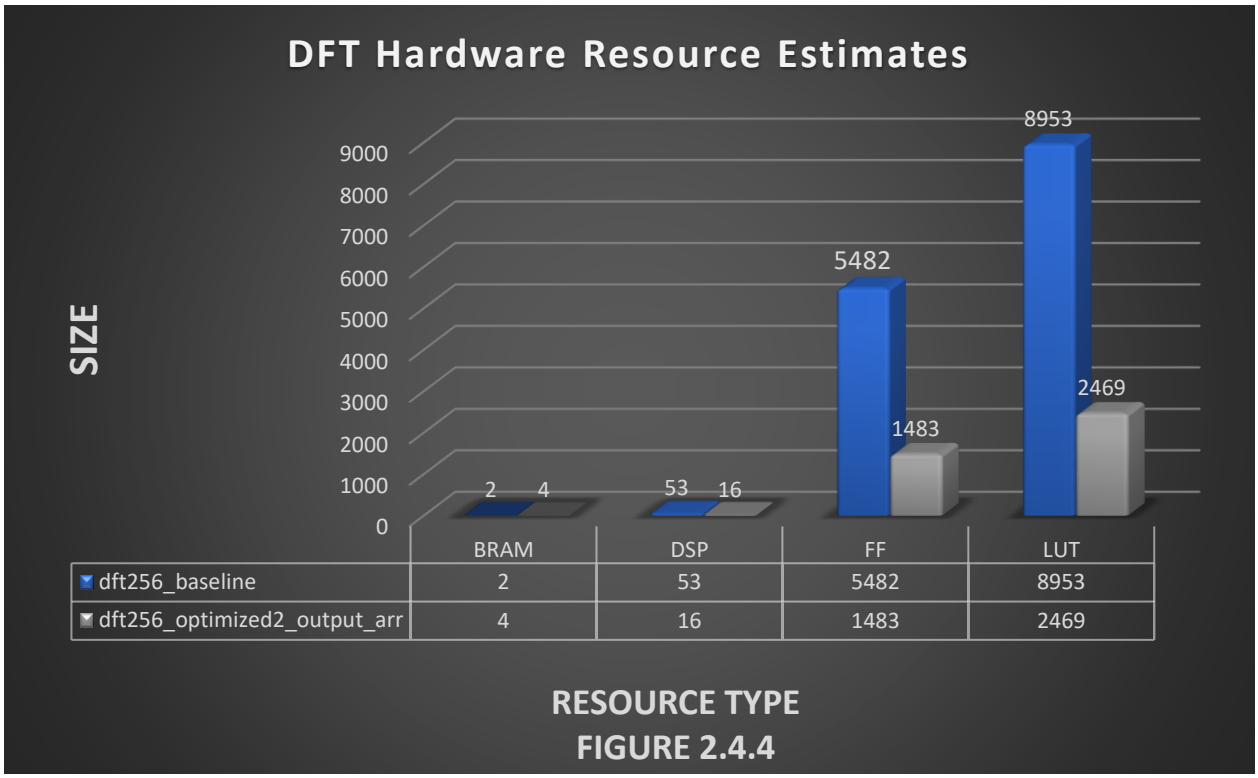


FIGURE 2.4.4

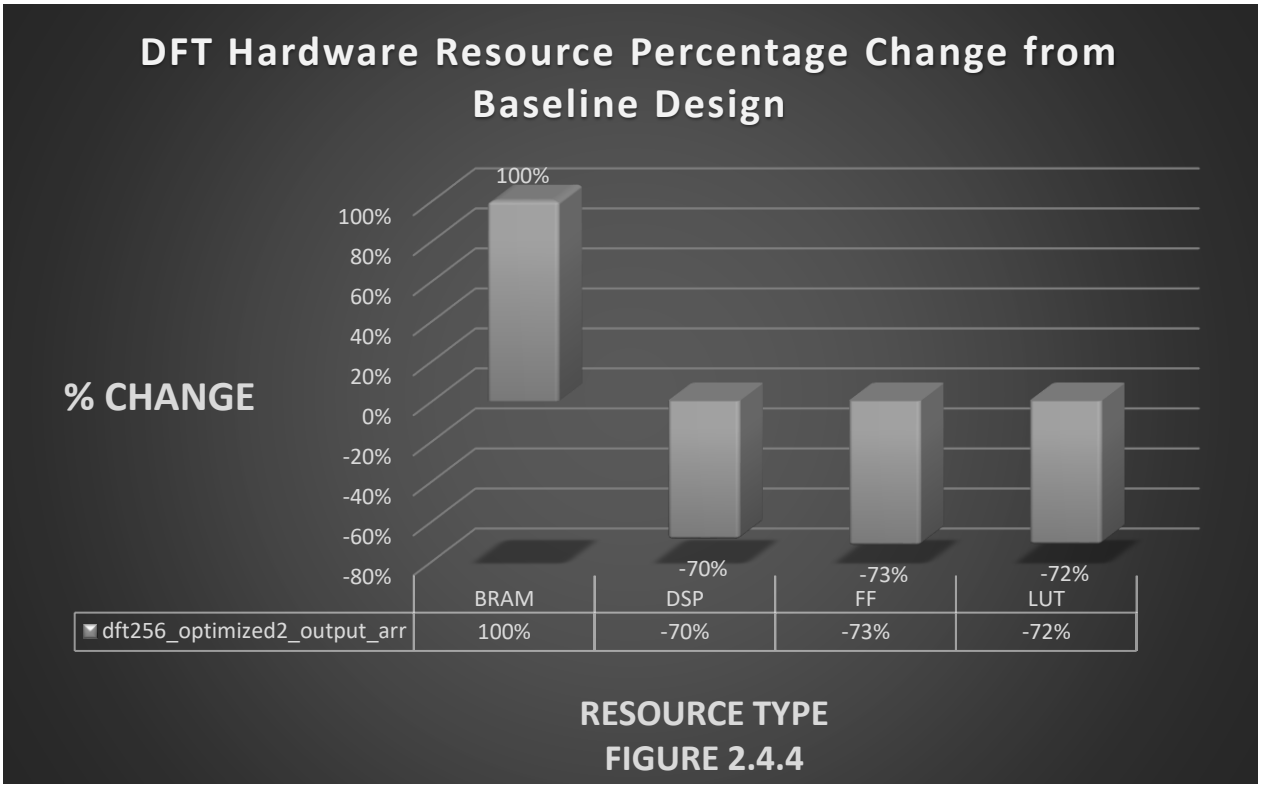


FIGURE 2.4.4

2.5. DFT with Loop Unroll Factor of 16

2.5.1. Throughput Max: 254 Hz

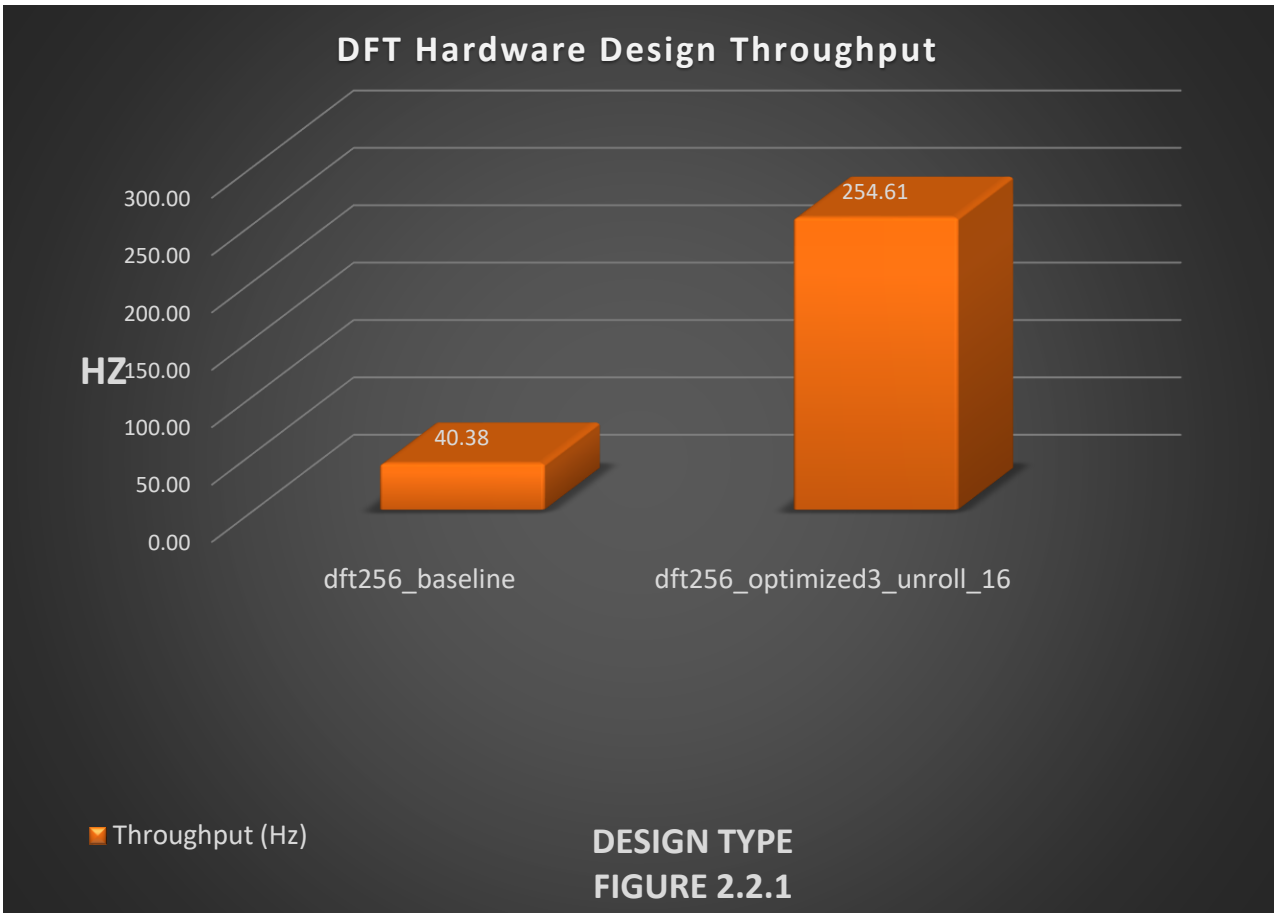


FIGURE 2.2.1

2.5.2. Implementation:

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];

    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=16
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=16
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS unroll
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.5.3. Optimizations:

- Unrolled inner and outer DFT loop by a factor of 16.

2.5.4. Resources:

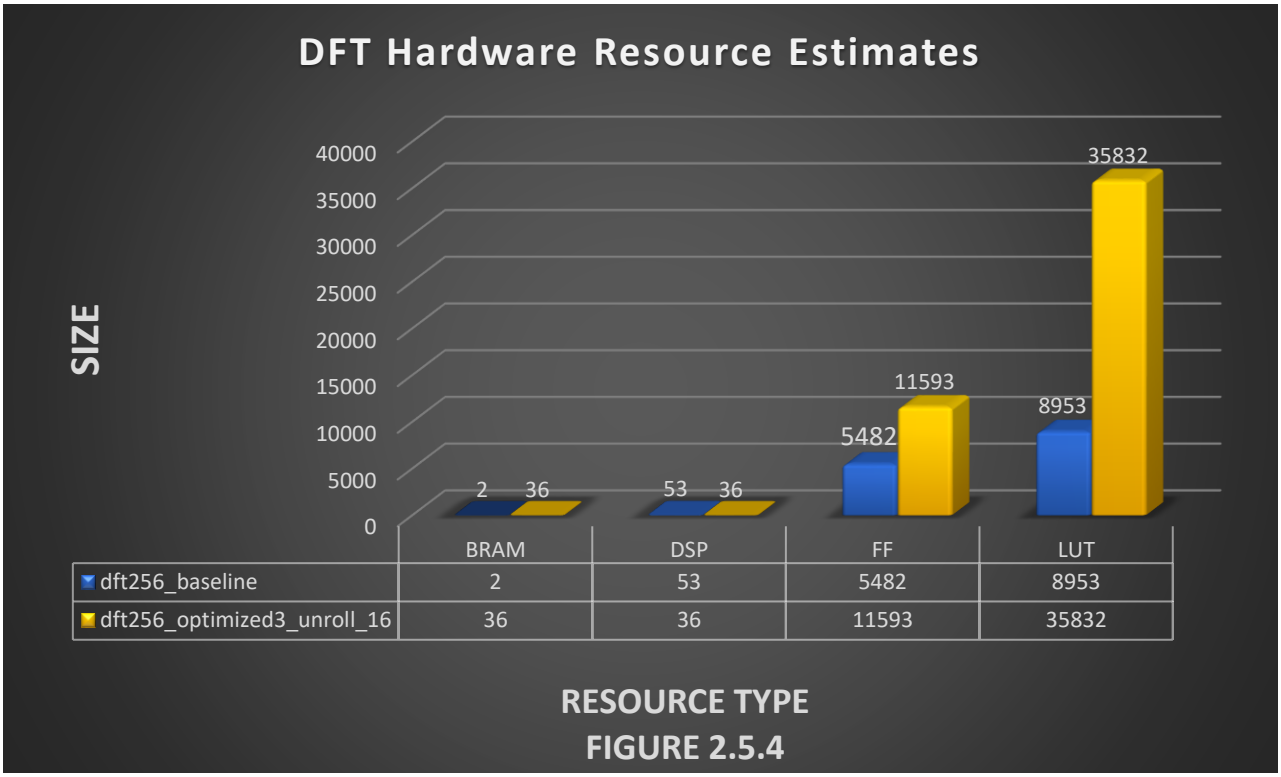


FIGURE 2.5.4

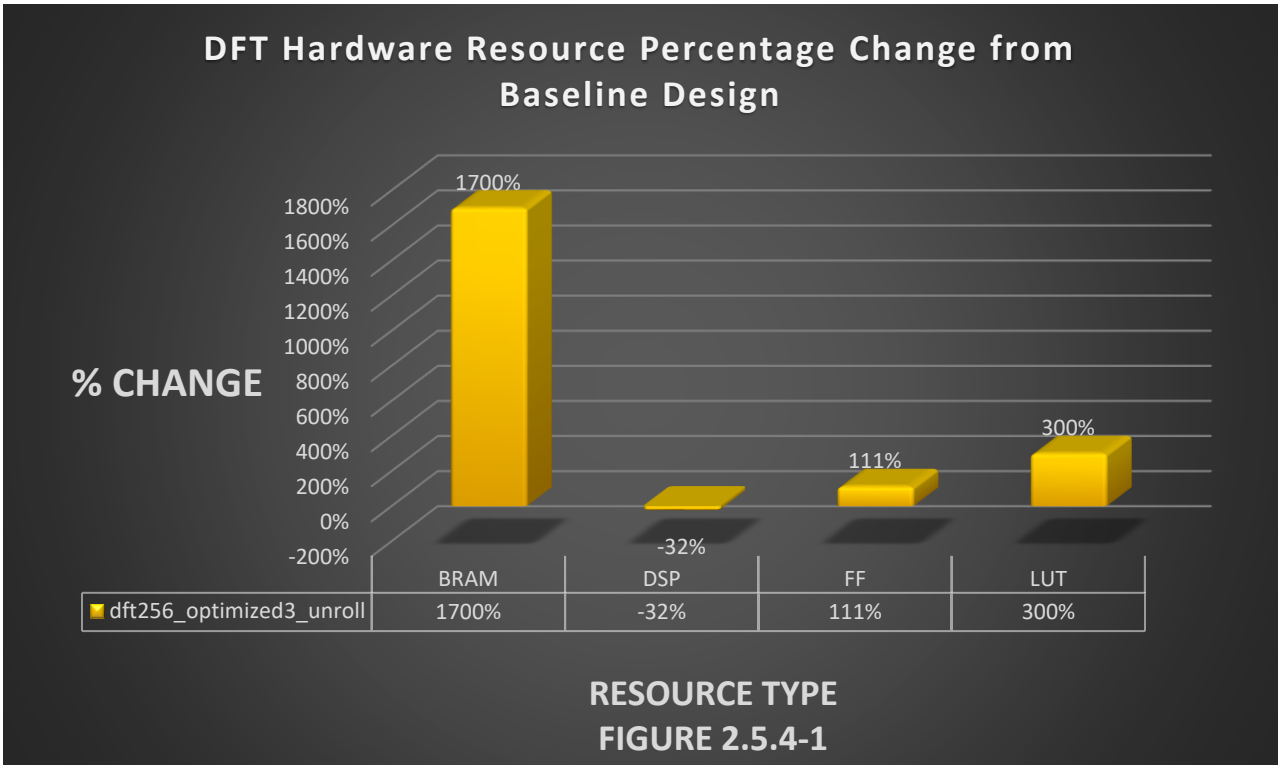


FIGURE 2.5.4-1

2.6. DFT with Block Array Partitioning Factor of 16

2.6.1. Throughput Max: 131 kHz

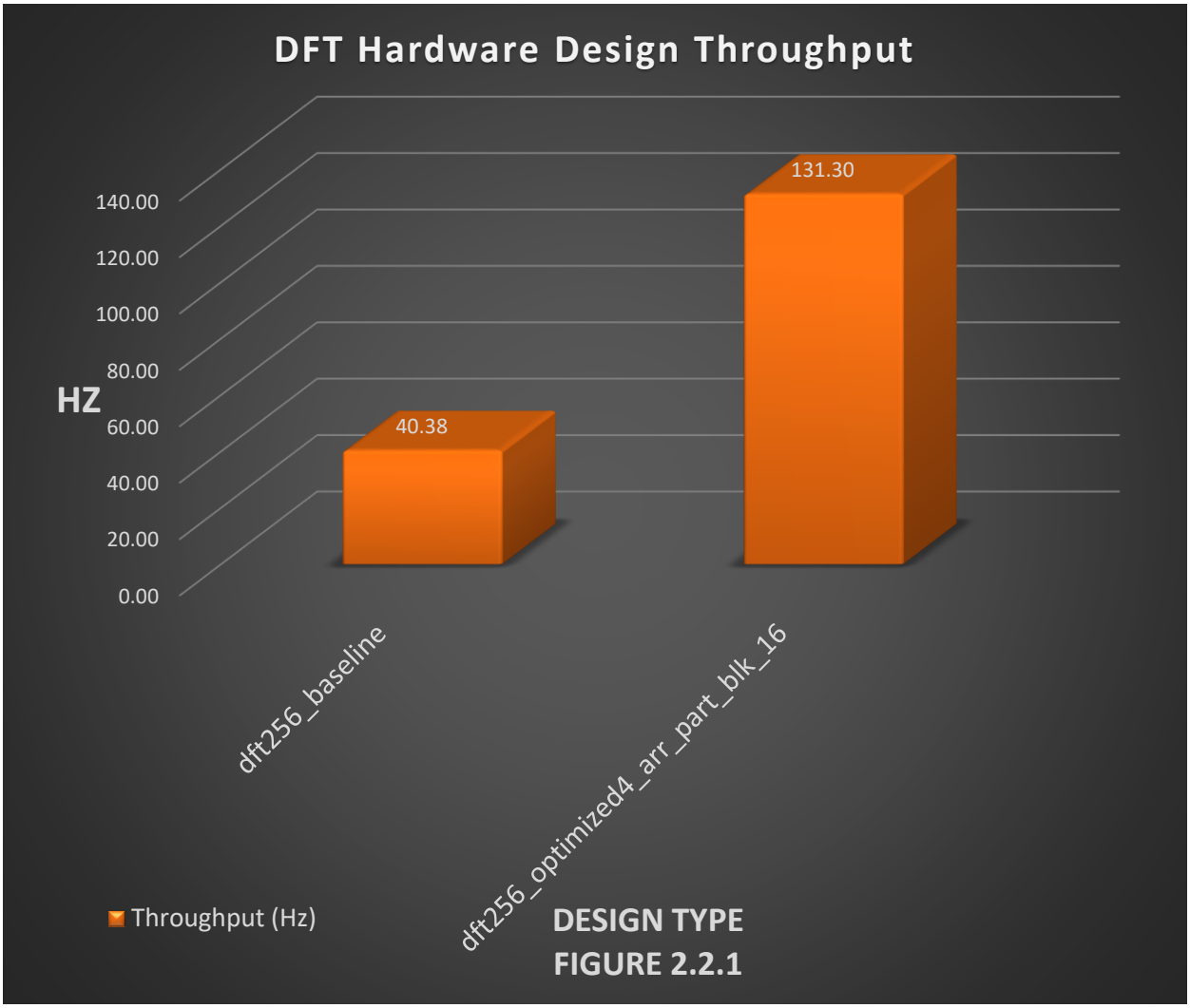


FIGURE 2.2.1

2.6.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
#pragma HLS array_partition variable=real_sample_in type=block factor=16
#pragma HLS array_partition variable=imag_sample_in type=block factor=16
#pragma HLS array_partition variable=real_sample_out type=block factor=16
#pragma HLS array_partition variable=imag_sample_out type=block factor=16
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
#pragma HLS array_partition variable=temp_real type=block factor=16
#pragma HLS array_partition variable=temp_imag type=block factor=16
    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
```

```
#pragma HLS PIPELINE off
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.6.3. Optimizations

- Applied block partitioning to all data arrays using a factor of 16.

2.6.4. Resources

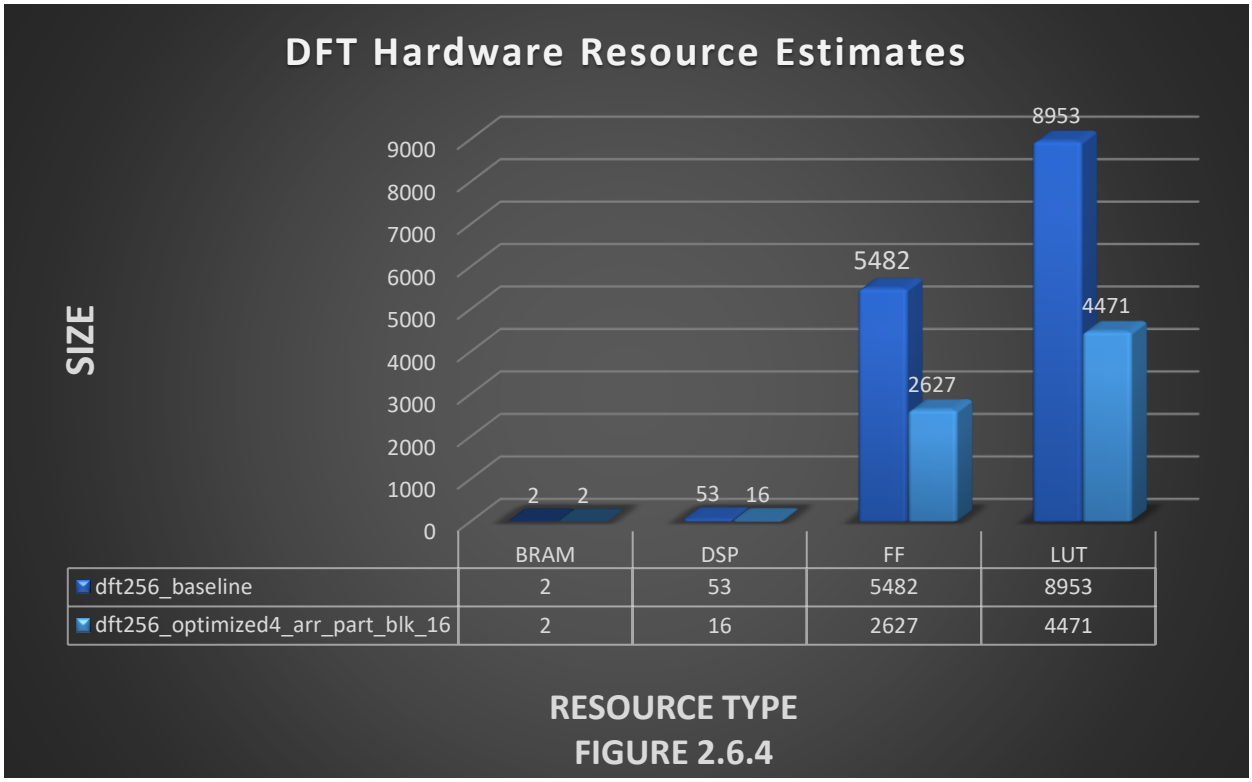


FIGURE 2.6.4

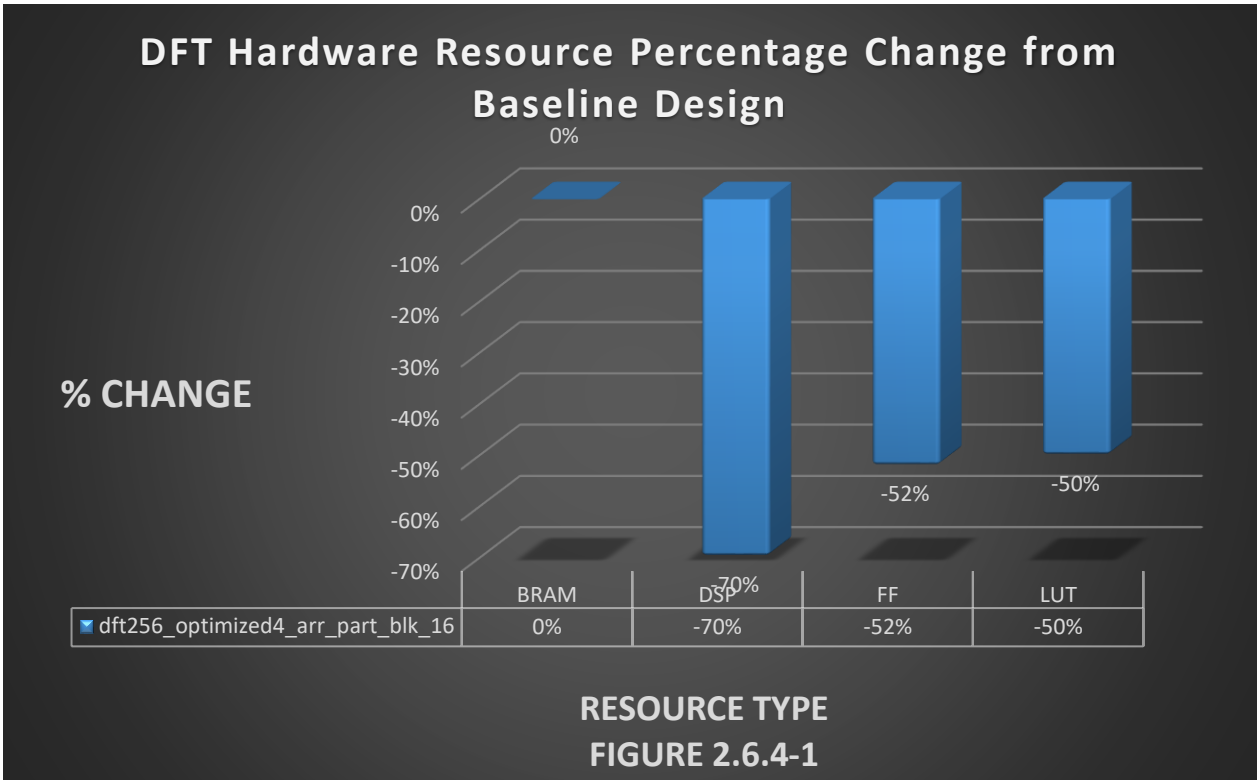

DFT Hardware Resource Percentage Change from Baseline Design
% CHANGE
0%
-10%
-20%
-30%
-40%
-50%
-60%
-70%
-52%
-50%
-70%
BRAM
DSP
FF
LUT
dft256_optimized4_arr_part_blk_16
0%
-70%
-52%
-50%
RESOURCE TYPE

FIGURE 2.6.4-1

2.7. DFT with Cyclic Array Partitioning Factor 16

2.7.1. Throughput Max: **131 Hz**

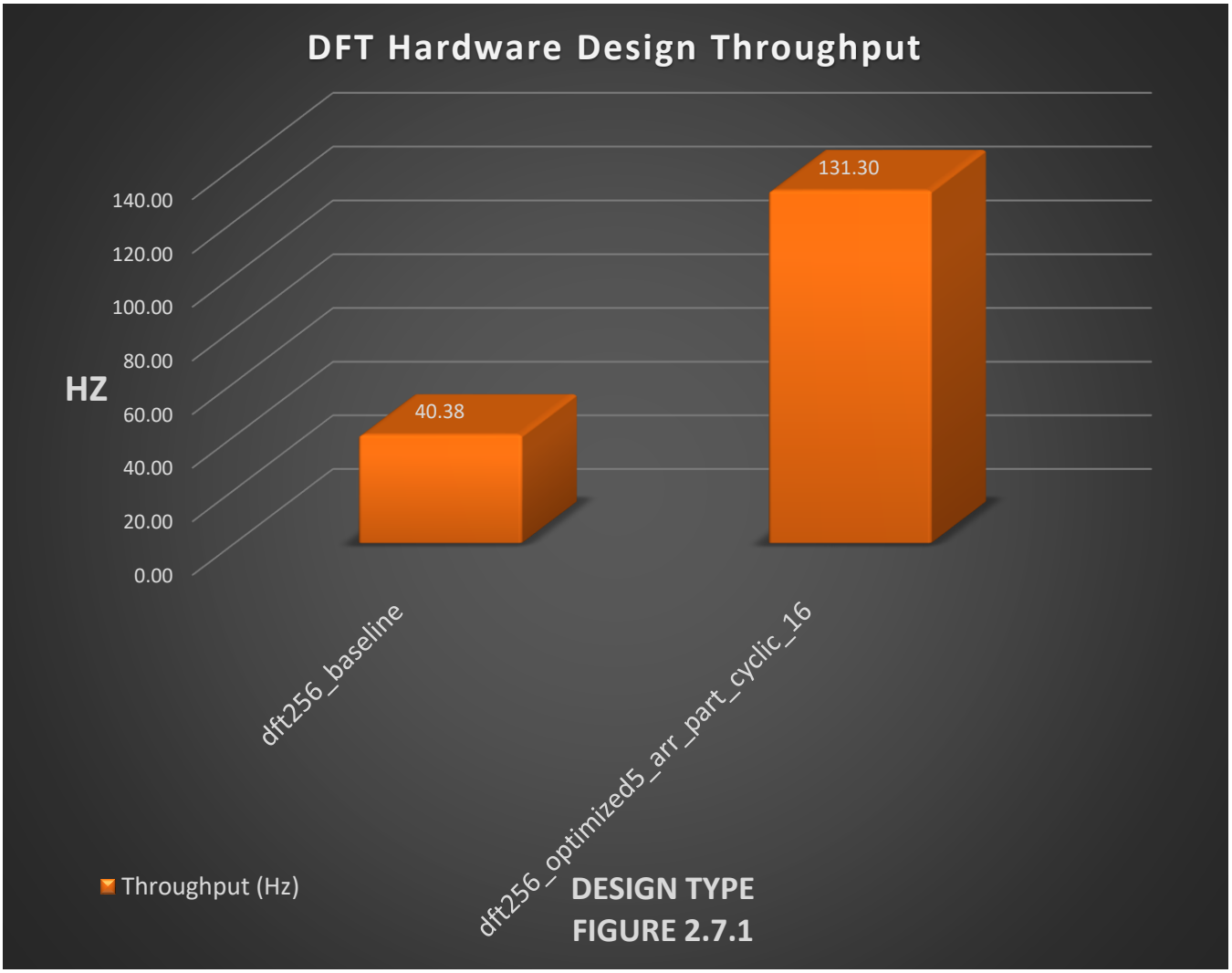


FIGURE 2.7.1

2.7.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
#pragma HLS array_partition variable=real_sample_in type=cyclic factor=16
#pragma HLS array_partition variable=imag_sample_in type=cyclic factor=16
#pragma HLS array_partition variable=real_sample_out type=cyclic factor=16
#pragma HLS array_partition variable=imag_sample_out type=cyclic factor=16
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;
    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
#pragma HLS array_partition variable=temp_real type=cyclic factor=16
#pragma HLS array_partition variable=temp_imag type=cyclic factor=16
    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off
```

```
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.7.3. Optimizations

- Applied cyclic partitioning to all data arrays using a factor of 16.

2.7.4. Resources

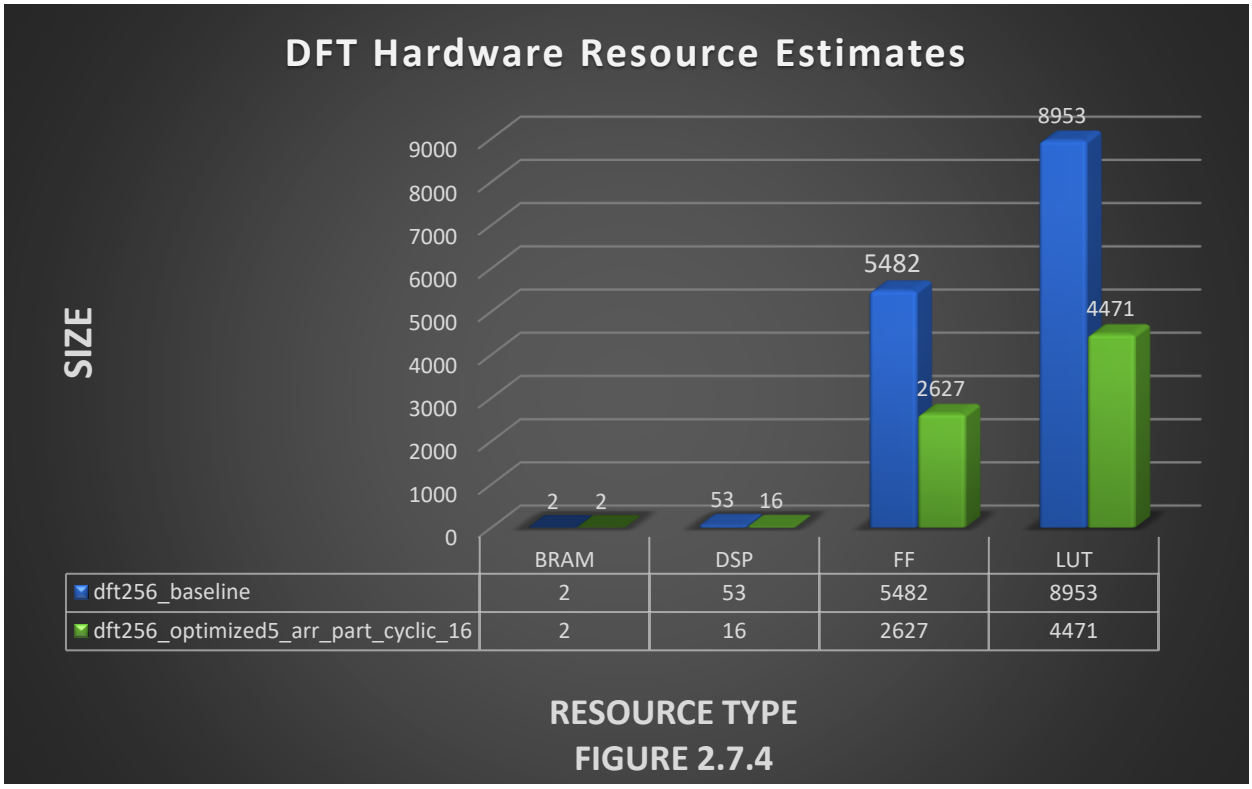


FIGURE 2.7.4

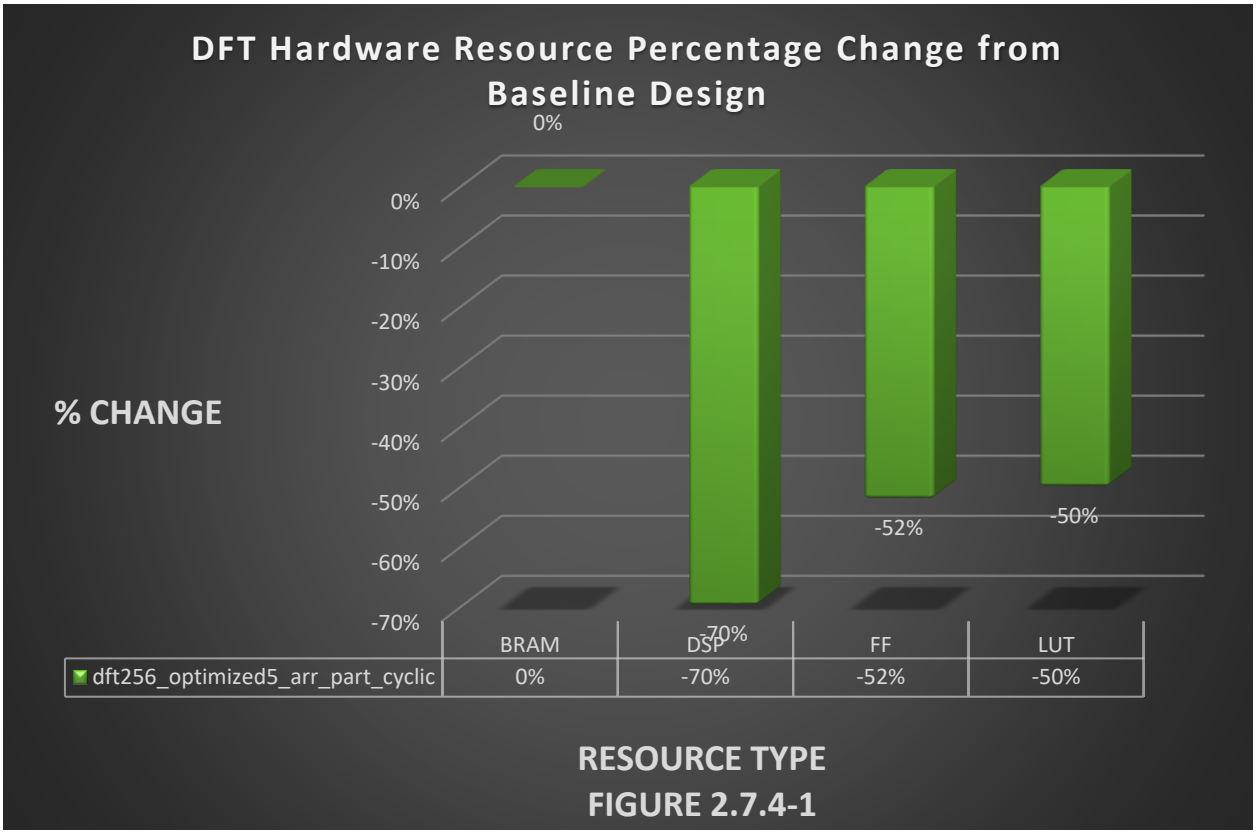

DFT Hardware Resource Percentage Change from Baseline Design
% CHANGE
0%
-10%
-20%
-30%
-40%
-50%
-60%
-70%
0%
-70%
-52%
-50%
BRAM
DSP
FF
LUT
dft256_optimized5_arr_part_cyclic
0%
-70%
-52%
-50%
RESOURCE TYPE

FIGURE 2.7.4-1

2.8. DFT with Loop unrolling Factor 16 and Cyclic Array Partitioning Factor 16

2.8.1. Throughput Max: 259 Hz

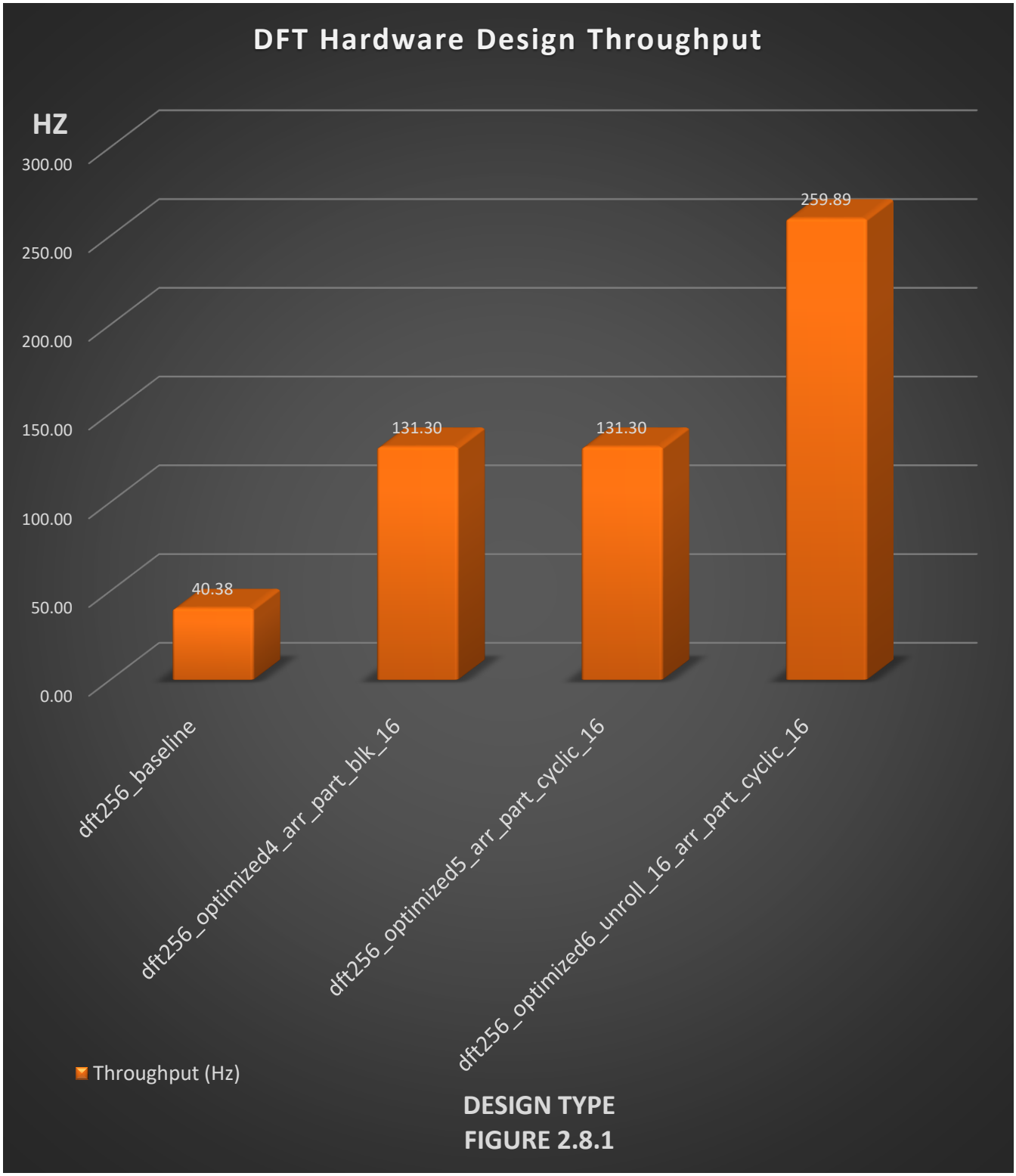


FIGURE 2.8.1

2.8.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
#pragma HLS array_partition variable=real_sample_in type=cyclic factor=16
#pragma HLS array_partition variable=imag_sample_in type=cyclic factor=16
```

```
#pragma HLS array_partition variable=real_sample_out type=cyclic factor=16
#pragma HLS array_partition variable=imag_sample_out type=cyclic factor=16
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
#pragma HLS array_partition variable=temp_real type=cyclic factor=16
#pragma HLS array_partition variable=temp_imag type=cyclic factor=16
    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=16
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=16
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.8.3. Optimizations

- Applied cyclic array partition to all data arrays.
- Applied loop unrolling to inner and outer loops.

2.8.4. Resources

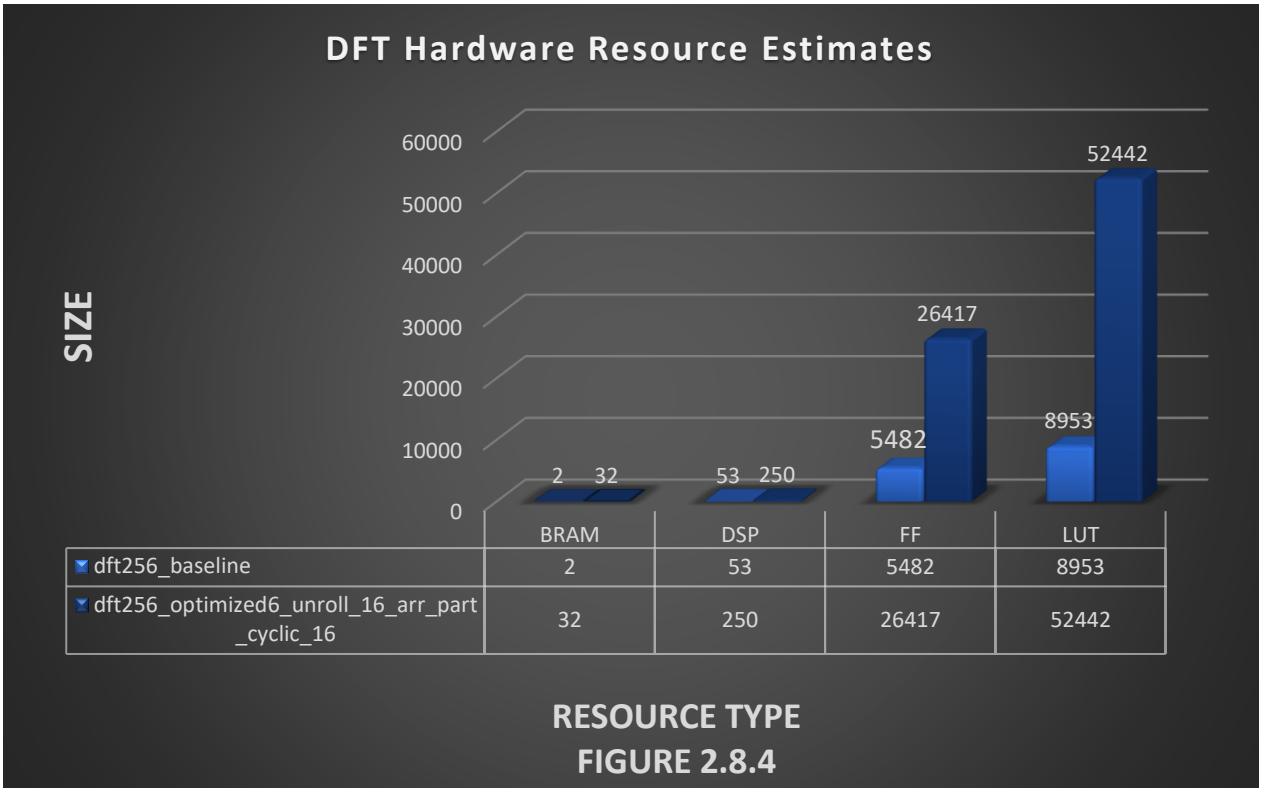


FIGURE 2.8.4

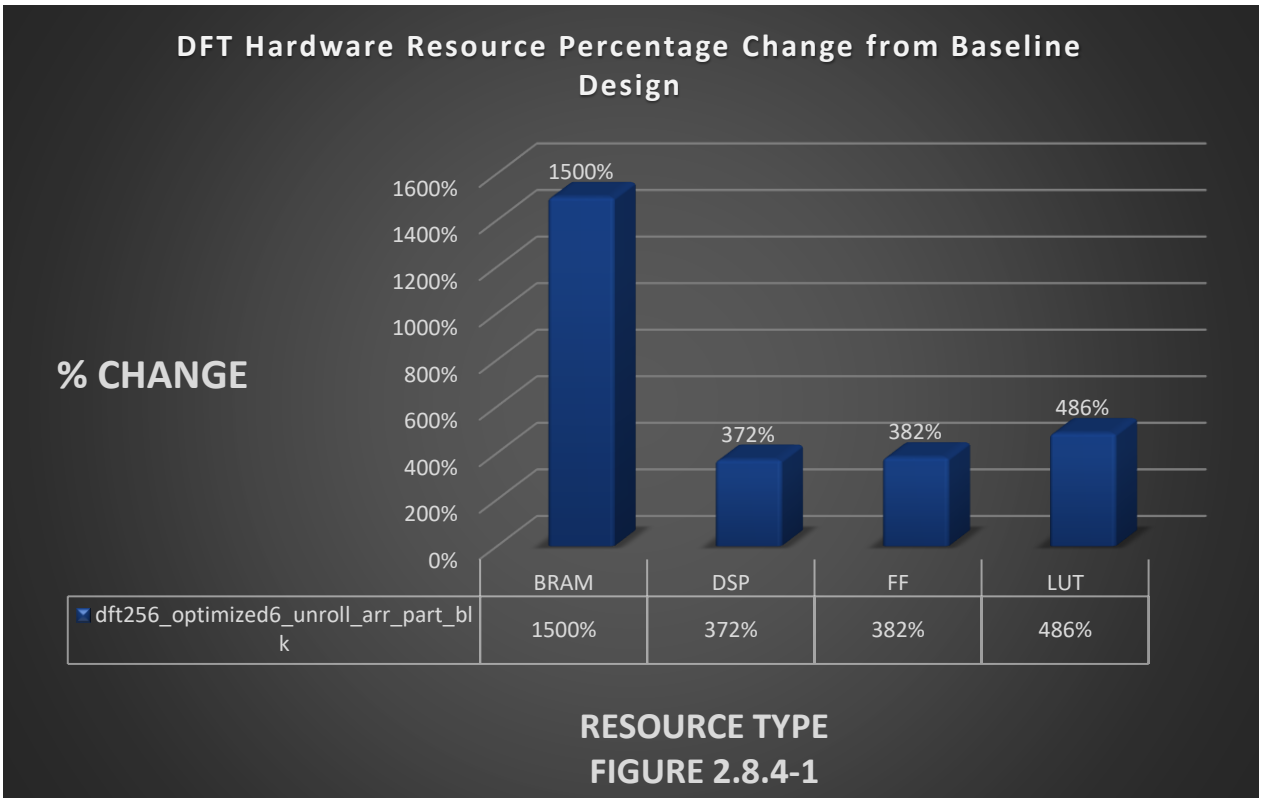


FIGURE 2.8.4-1

2.9. DFT with Dataflow

2.9.1. Throughput Max: 131 Hz

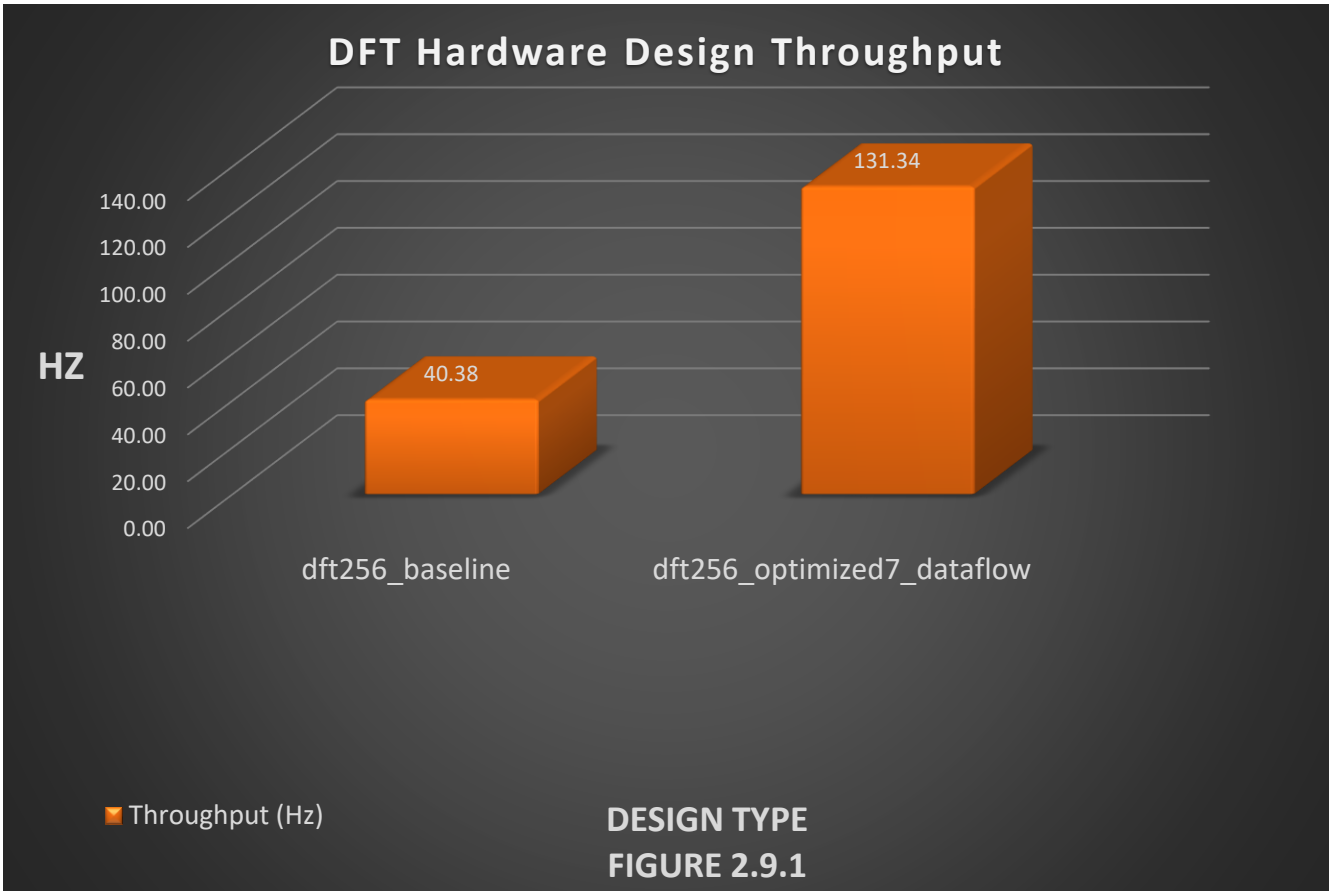


FIGURE 2.9.1

2.9.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
#pragma HLS dataflow

    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];

    DTYPE temp_c[1];
    DTYPE temp_s[1];


    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE on
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off

            index = i*j;
            assign_coefficients(index,temp_c,temp_s);
            mult_accum(real_sample_in[j],
```

```
                        imag_sample_in[j],
                        temp_c,
                        temp_s,
                        temp_real,
                        temp_imag,
                        i);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}

void assign_coefficients(ap_uint<8> index, DTYPE c[1], DTYPE s[1])
{
    c[0] = cos_coefficients_table[index];
    s[0] = sin_coefficients_table[index];
}
void mult_accum(DTYPE real_sample_in,
                DTYPE imag_sample_in,
                DTYPE temp_c[1],
                DTYPE temp_s[1],
                DTYPE temp_real[SIZE],
                DTYPE temp_imag[SIZE],
                int i)
{
    temp_real[i] += (real_sample_in * temp_c[0] - imag_sample_in * temp_s[0]);
    temp_imag[i] += (real_sample_in * temp_s[0] + imag_sample_in * temp_c[0]);
}
```

2.9.3. Optimizations

- Applied dataflow pragma to top level function.

2.9.4. Resources

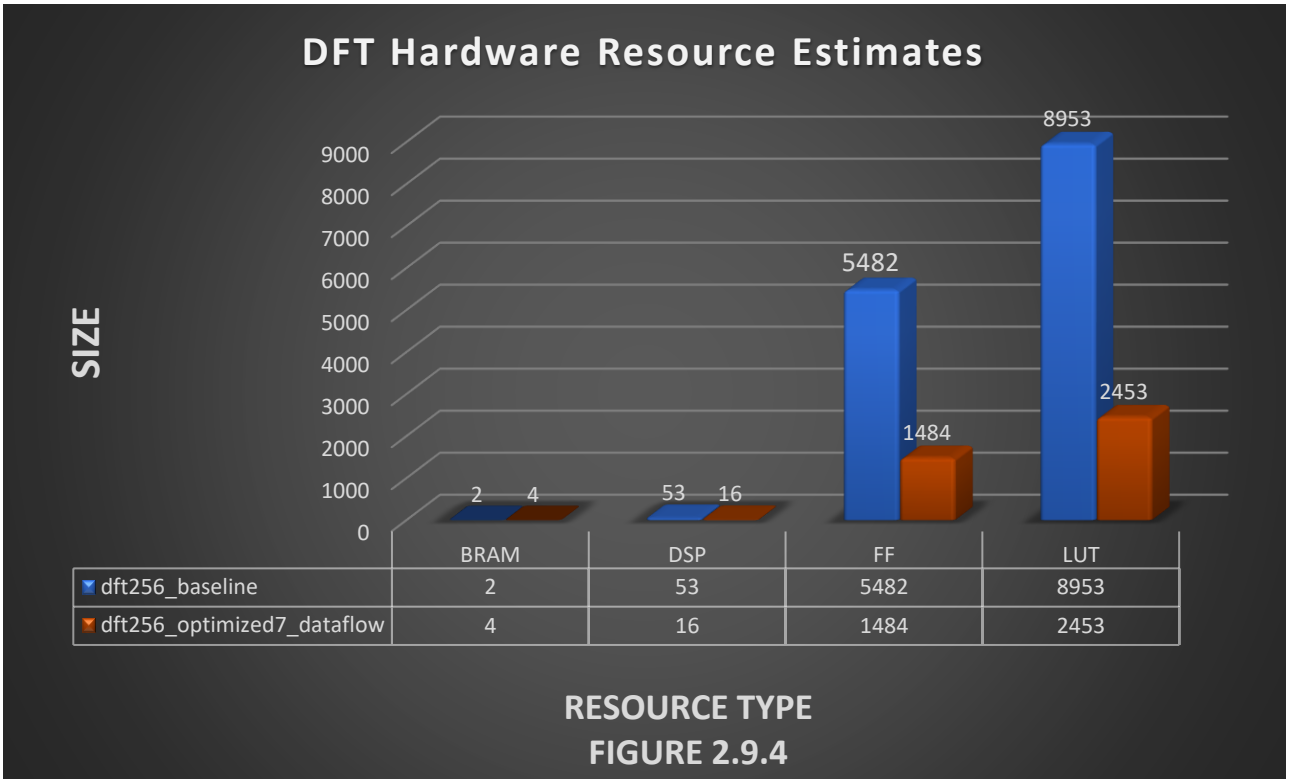


FIGURE 2.9.4

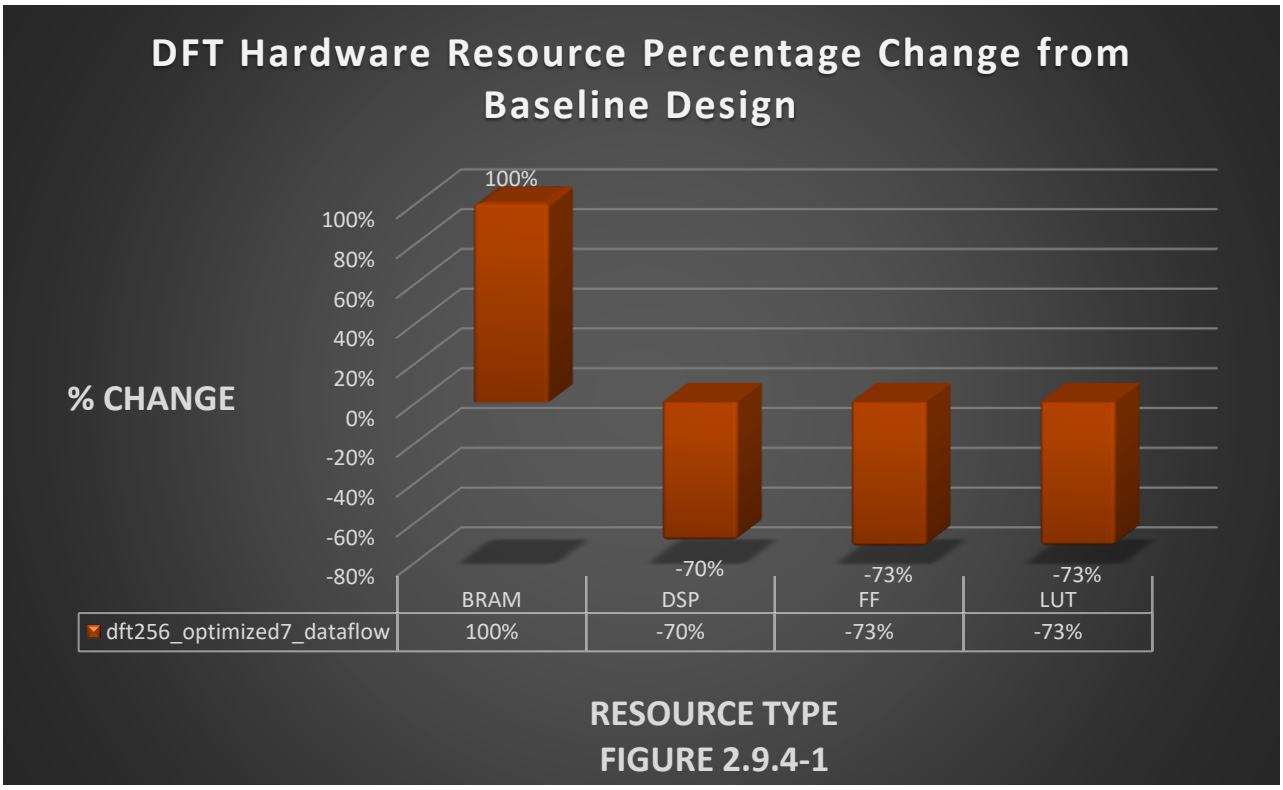


FIGURE 2.9.4-1

2.10. DFT 256 Best Design

2.10.1. Throughput Max: 275 Hz

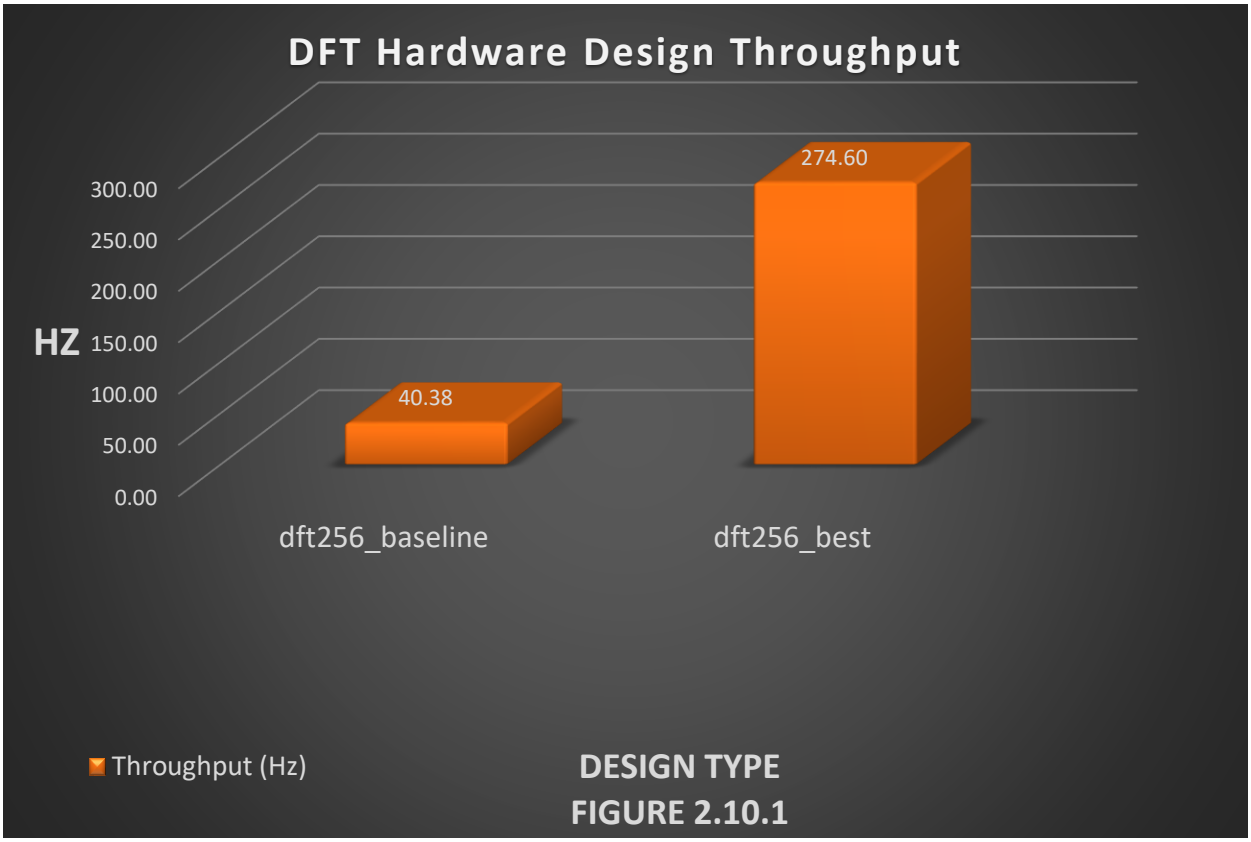


FIGURE 2.10.1

2.10.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
#pragma HLS array_partition variable=real_sample_in type=cyclic factor=8
#pragma HLS array_partition variable=imag_sample_in type=cyclic factor=8
#pragma HLS array_partition variable=real_sample_out type=cyclic factor=8
#pragma HLS array_partition variable=imag_sample_out type=cyclic factor=8
#pragma HLS array_partition variable=cos_coefficients_table type=complete
#pragma HLS array_partition variable=sin_coefficients_table type=complete
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
#pragma HLS array_partition variable=temp_real type=cyclic factor=8
#pragma HLS array_partition variable=temp_imag type=cyclic factor=8
    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=8
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=8
            index = i*j;

            c = cos_coefficients_table[index];
```

```
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS unroll
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.10.3. Optimizations

- Applied cyclic array partitioning with a factor of 8 to the real and imaginary input and output arrays
- Applied cyclic array partitioning with a factor of 8 to the temporary real and imaginary arrays used in the body of the DFT loop.
- Applied complete array partitioning to cosine and sin arrays.
- Applied loop unrolling with a factor of 8 to the inner and outer loops of the DFT.
- Applied complete loop unrolling to the final loop in the top level module.

2.10.4. Resources

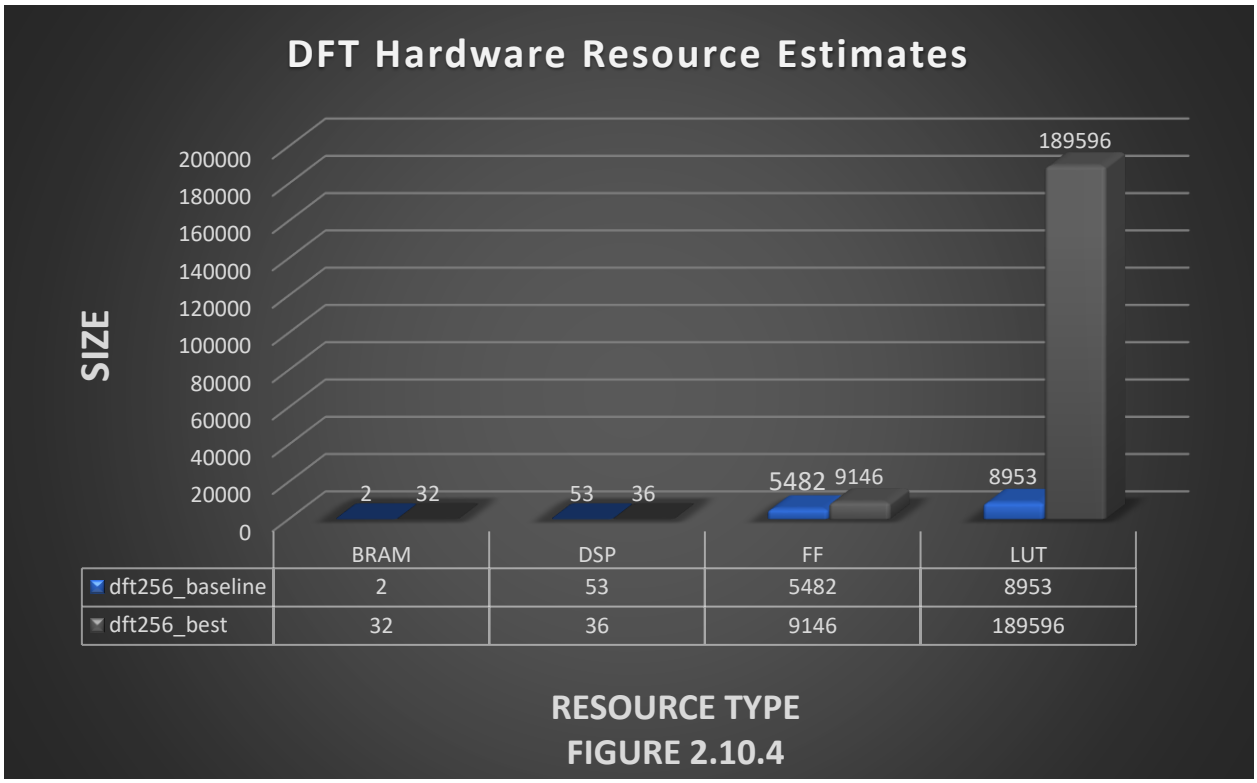


FIGURE 2.10.4

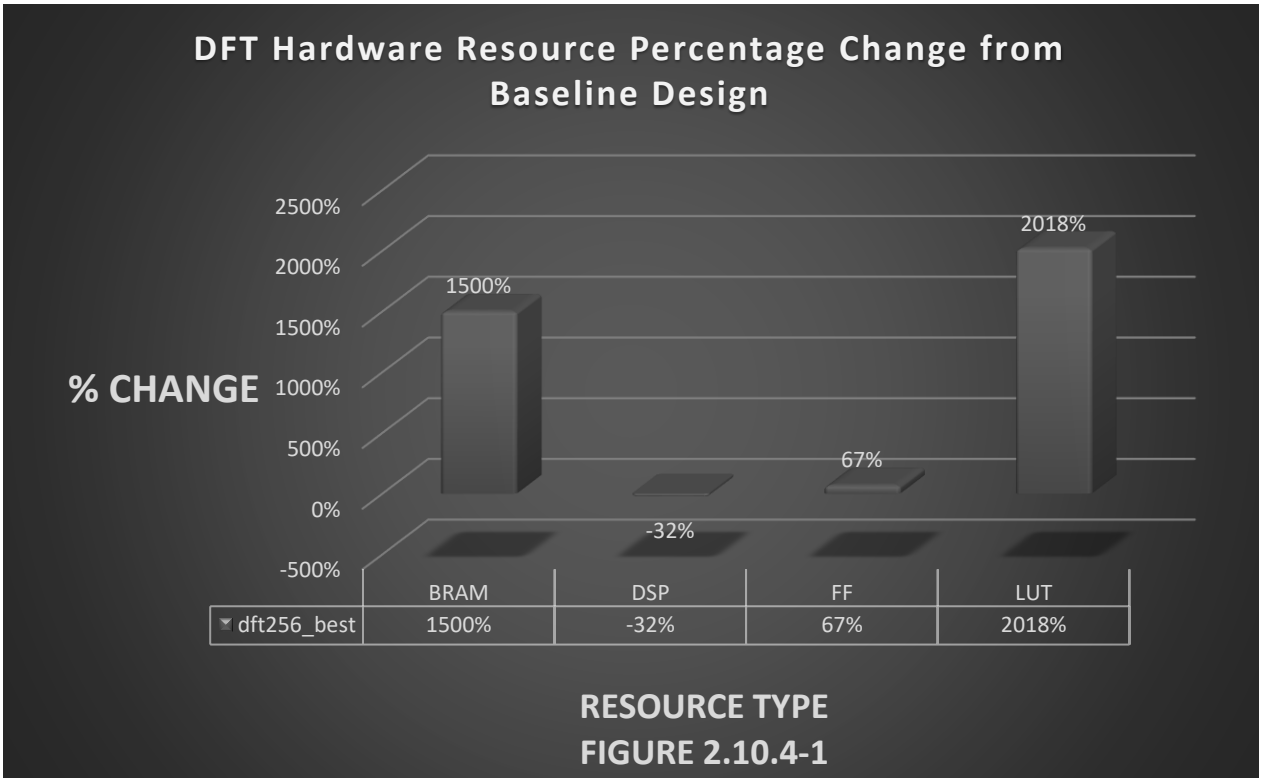


FIGURE 2.10.4-1

2.11. DFT 1024 Best Design

2.11.1. Throughput Max: **17 Hz**

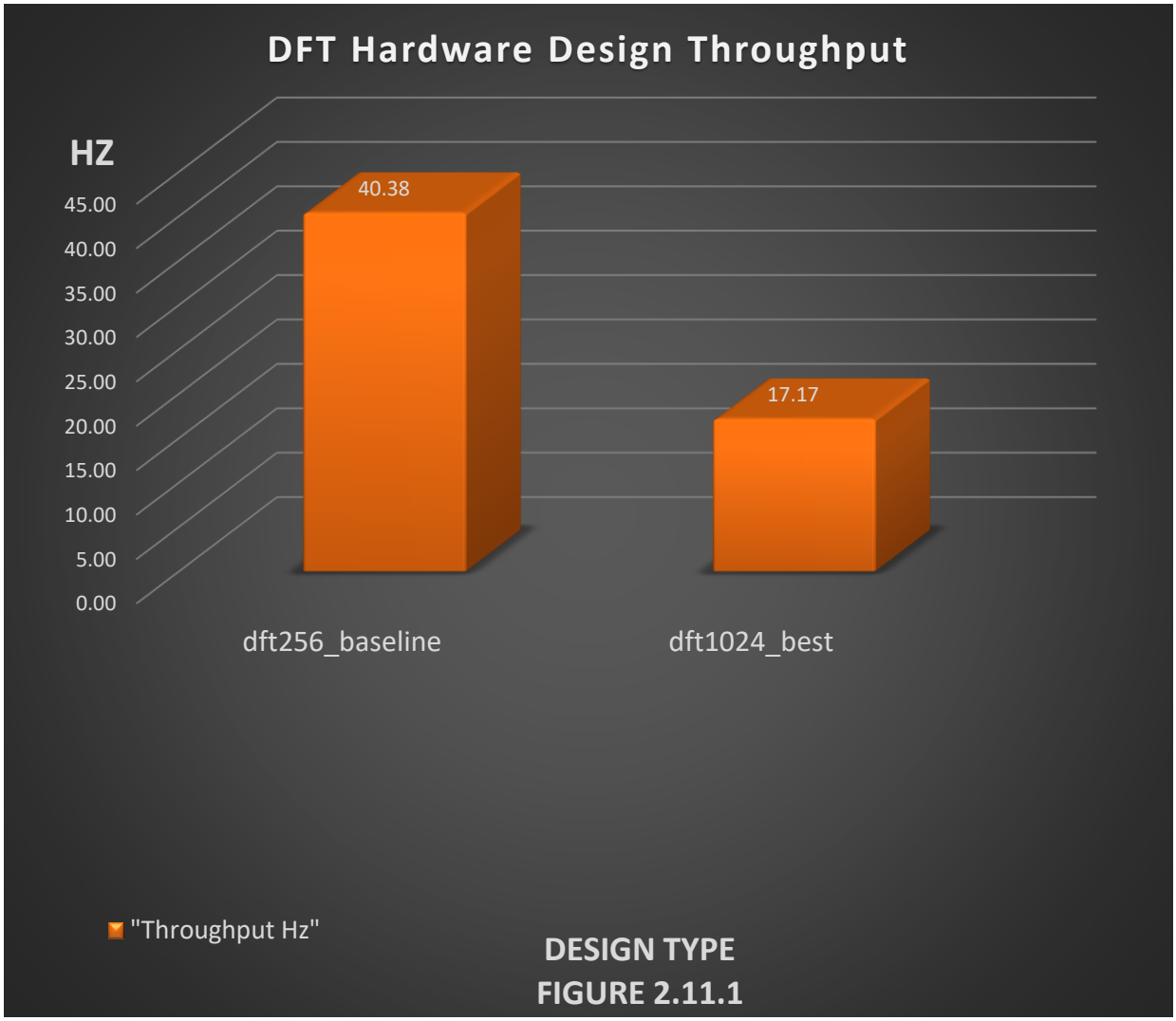


FIGURE 2.11.1

2.11.2. Implementation

```
void dft(DTYPE real_sample_in[SIZE], DTYPE imag_sample_in[SIZE],DTYPE
real_sample_out[SIZE], DTYPE imag_sample_out[SIZE])
{
#pragma HLS array_partition variable=real_sample_in type=cyclic factor=8
#pragma HLS array_partition variable=imag_sample_in type=cyclic factor=8
#pragma HLS array_partition variable=real_sample_out type=cyclic factor=8
#pragma HLS array_partition variable=imag_sample_out type=cyclic factor=8
#pragma HLS array_partition variable=cos_coefficients_table type=complete
#pragma HLS array_partition variable=sin_coefficients_table type=complete
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
#pragma HLS array_partition variable=temp_real type=cyclic factor=8
#pragma HLS array_partition variable=temp_imag type=cyclic factor=8
    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=8
```

```
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off
#pragma HLS unroll factor=8
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];

            temp_real[i] += (real_sample_in[j] * c - imag_sample_in[j] * s);
            temp_imag[i] += (real_sample_in[j] * s + imag_sample_in[j] * c);
        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS unroll
        real_sample_out[i] = temp_real[i];
        imag_sample_out[i] = temp_imag[i];
    }
}
```

2.11.3. Optimizations

- Applied cyclic array partitioning with a factor of 8 to the real and imaginary input and output arrays
- Applied cyclic array partitioning with a factor of 8 to the temporary real and imaginary arrays used in the body of the DFT loop.
- Applied complete array partitioning to cosine and sin arrays.
- Applied loop unrolling with a factor of 8 to the inner and outer loops of the DFT.
- Applied complete loop unrolling to the final loop in the top level module.

2.11.4. Resources

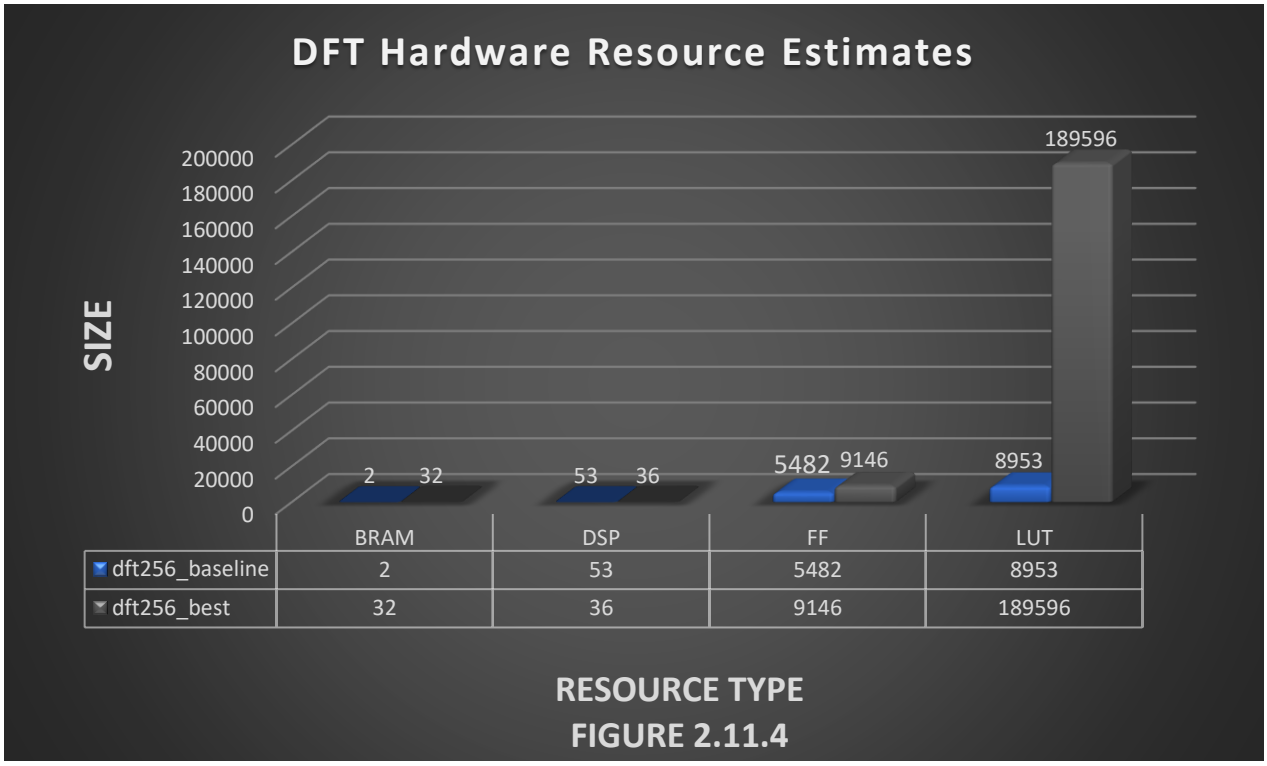


FIGURE 2.11.4

3. DFT Questions

3.1. Question 1 – Comparison with CORDIC

3.1.1. Updating the baseline design to use a CORDIC would require adding an additional function to calculate the sin and cosine components for each sample. The CORDIC function to calculate sin and cosine functions would replace the HLS cos() and sin() calls used in the body of the DFT loop. Using a CORDIC core to replace the cos() and sin() function calls would likely result in using less resources. The performance of the DFT module should improve because the algorithm would be tuned to perform the exact number of cycles needed to produce an accurate output.

3.2. Question 2 – Replace sin() cos() with LUT

3.2.1. Replacing the sin and cosine functions with a lookup table results in a moderate change in throughput and a significant change in resource utilization as seen in Figure 2.3.4. Changing the size of the DFT will result in an increase in the size of your lookup table since the lookup table on use 256 values.

3.3. Question 3 – Separate Output Array

3.3.1. Modifying the DFT function interface to use separate arrays for the input and out results in a major decrease in BRAM usage, but also a large increase in all other resources as seen in Figure 2.4.4. This change allows us to add optimizations to both arrays using the array partition pragma if needed. The performance of this design is not significantly different from the baseline. The major tradeoff with this architecture is the decrease in usage of BRAMS which has a slower access time.

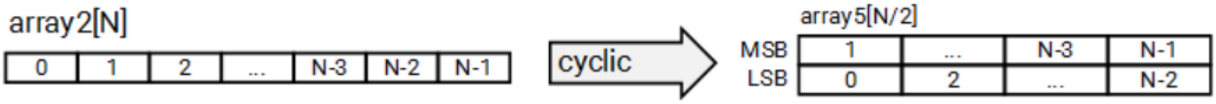
3.4. Question 4 – Loop Optimizations

3.4.1. Loop unrolling by a small factor of 16 provides a significant increase in throughput over the baseline implementation, but also results in major increase in resource utilization as expected. Specifically block RAM by requiring more than 15x the amount of block RAM that the baseline requires as shown in Figure2.5.4-1. Array partitioning by comparison provides a moderate increase in performance while also decreasing the total number of resources used as seen in Figure 2.6.4-1 and Figure 2.7.4-1. Array partition and loop unrolling together produces an architecture that provides a significant increase in performance over the baseline as seen in Figure 2.8.1. However, performing loop unrolling without array partitioning or array partitioning without loop unrolling does not provides approximately the same increase in performance.

3.4.2. The performance of an architecture slightly increases/decreases as the unroll factor and array partition factor increases/decreases.

3.4.3. The best architecture to use would be one that includes array partitioning and loop unrolling. It should use cyclic array partitioning for the inputs, complete array partitioning for the coefficients, complete array partitioning for the outputs, loop unrolling by a factor for the DFT loops, and complete loop unrolling for the final loop.

3.4.3.1. For example, if the real and imaginary input sample values were stored in “array2” like in the figure below the HLS will create smaller arrays by interleaving elements from the original array. The array is partitioned cyclically by putting one element into each new array before coming back to the first array to repeat the cycle until the array is fully partitioned. This allows more data to be accessed in a single clock which enables an N factor number of inputs to be used during the DFT calculation within a clock cycle.



3.4.3.2. The general trend for the performance of the cyclic array portioning shows that increasing the cyclic factor does not greatly affect the throughput as seen in Figure 3.4.3.2-1, and increasing the cyclic factor results in an increase in resource usage as seen in Figure 3.4.3.2-2. Loop unrolling results in an increase in performance as the loop unrolling factor increases as seen in Figure 3.4.3.2-3, and the increase in the loop unrolling factor also results in a major increase the resource usage as seen in Figure 3.4.3.2-4. The general trend for both loop unrolling and array partition shows that a minimal to moderate improvement in performance can be gained at the expense of high resource usage. I would choose loop unrolling due to the greater performance gains. DFT operations are meant to calculate as many samples as possible with a certain amount of time. Therefore, high performance is extremely important to this architecture.

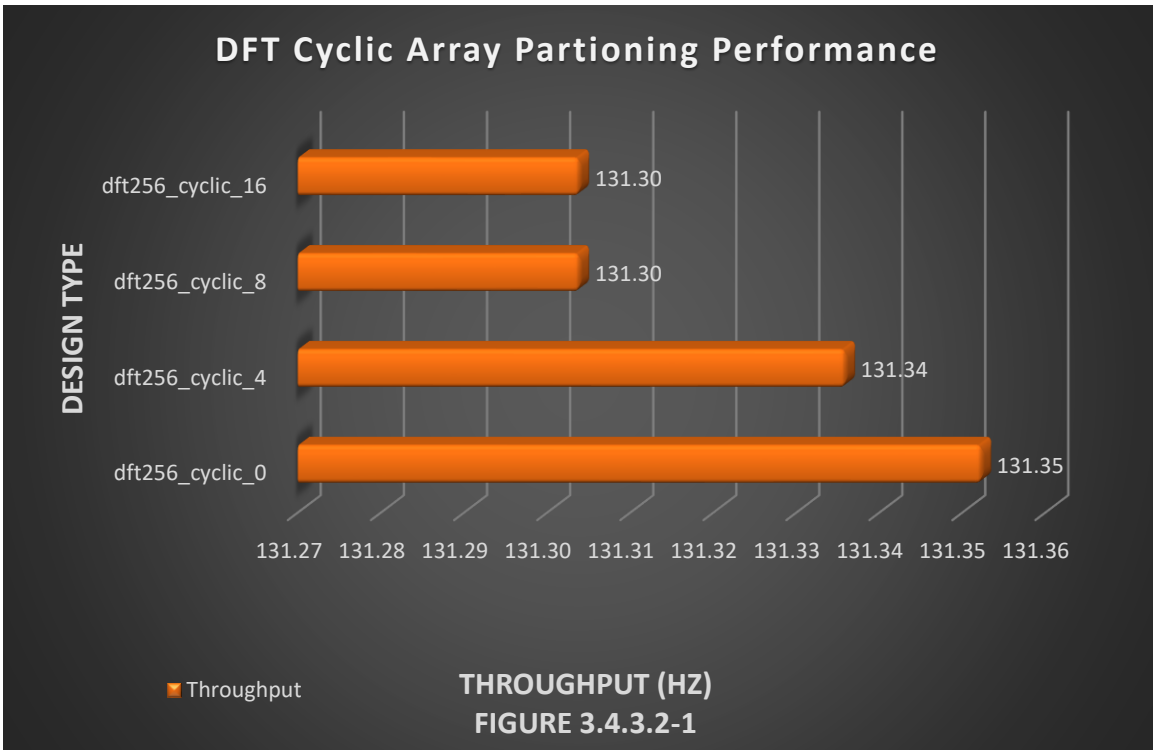


FIGURE 3.4.3.2-1

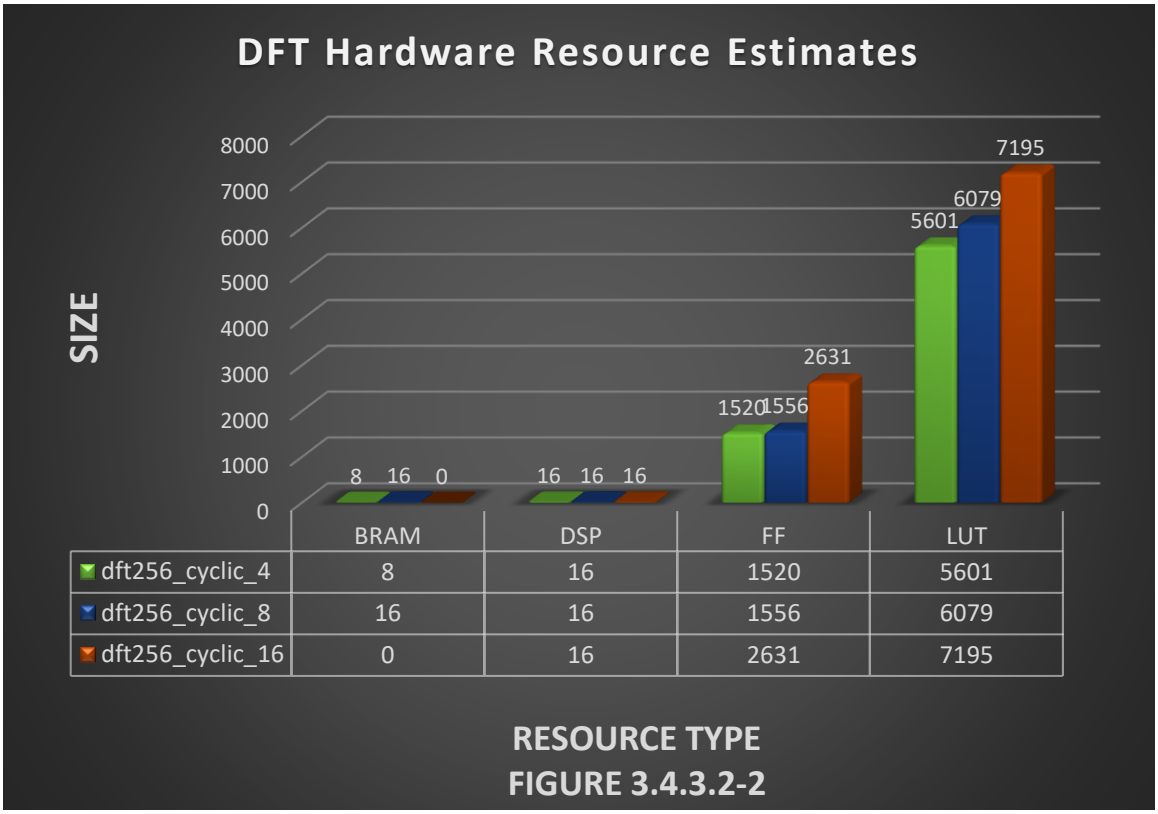


FIGURE 3.4.3.2-2

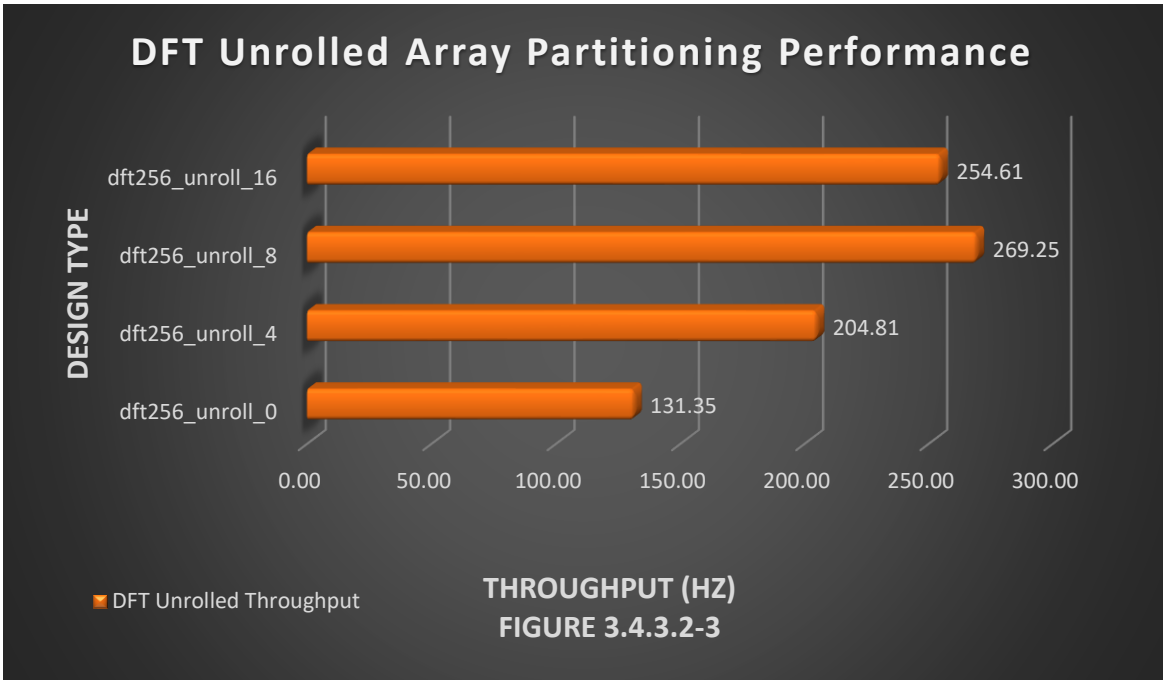


FIGURE 3.4.3.2-3

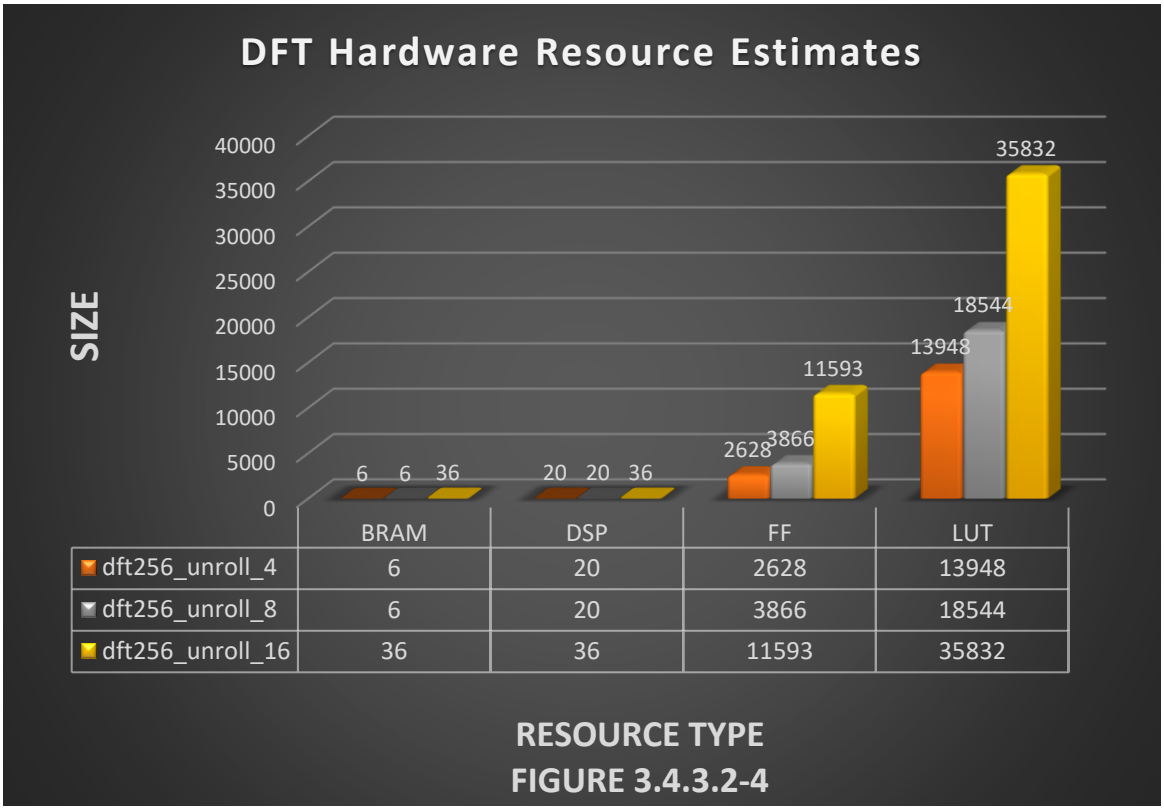


FIGURE 3.4.3.2-4

3.5. Question 5 – Dataflow

3.5.1. How much improvement does dataflow provide? Dataflow provided a moderate amount of improvement over the baseline as seen in Figure 2.9.1. Using dataflow decreased the resource usage in every area except for BRAMs which saw its resource usage double as seen in Figure 2.9.4-1. The code was modified to allow the cos and sin coefficient initialization step of the DFT inner loop to occur in a pipelined like fashion.

3.6. Question 6 – Best Architecture

3.6.1. The best DFT architecture is one that produces a high number of samples per cycle. I've opted to maximize throughput for this reason. The primary optimizations used to do this were loop unrolling and array partitioning. Choosing these optimizations lead to a high throughput usage especially in the area of flip flops which is an acceptable tradeoff due to the significant increase in throughput over the baseline and other architectures as seen in Figure 3.6.1.

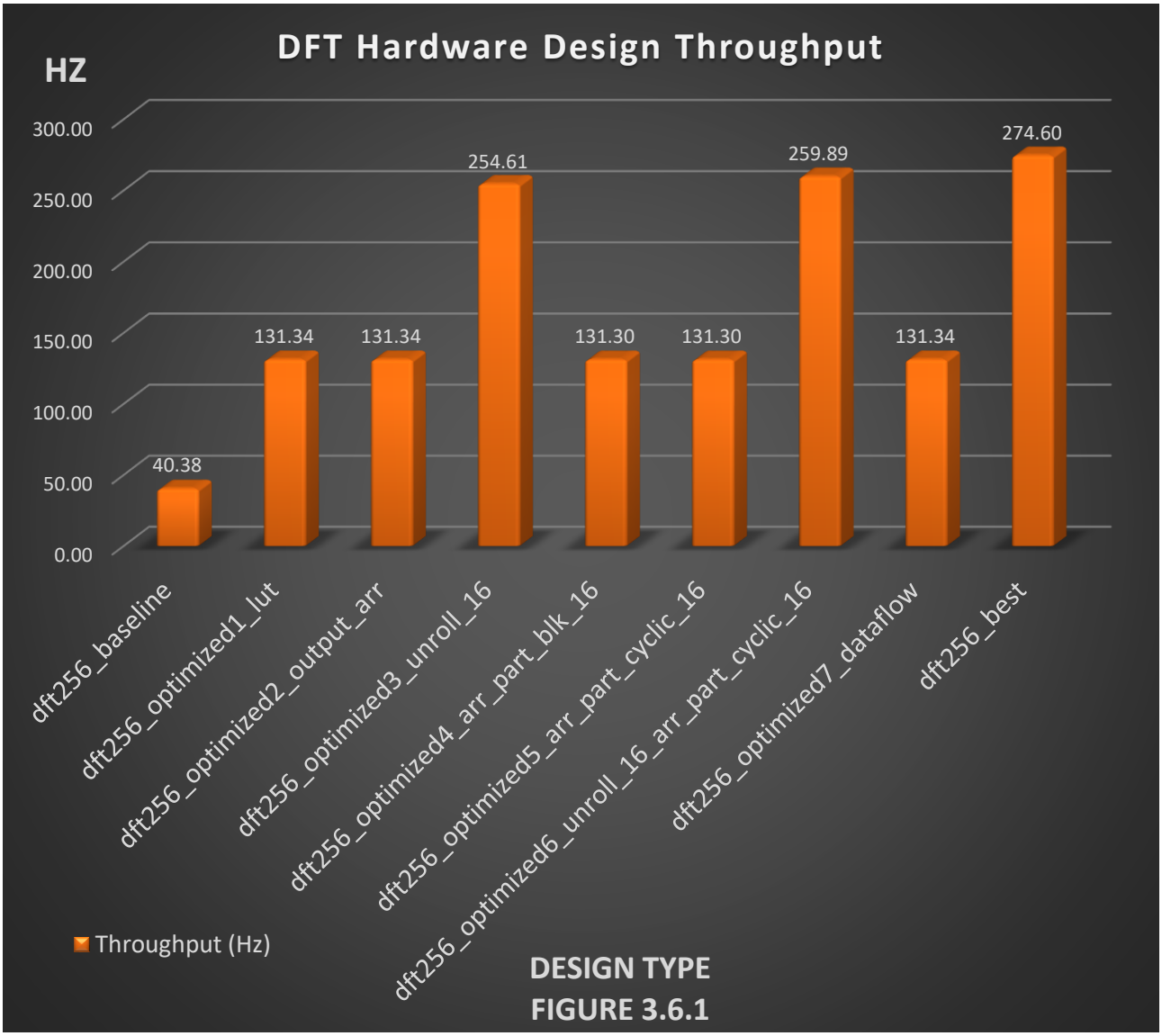


FIGURE 3.6.1

3.7. Question 7 – Streaming Interface Synthesis

3.7.1. Using the streaming interface required changes to the interface of the top-level function and updates to the test bench and top level function for reading and writing data to and from the stream. Another major change was the addition of a union that stored an integer and float value. This was needed to make sure that the data was moved through the DMA in the proper format. The bus expects the values to be integer format, but the DFT function operates on floating point values so using union allows for converting from one to the other.

3.7.2. A benefit of using a streaming interface is it allows you to communicate with the DMA directly, instead of relying on vivado to hopefully set the t_last bit on your last data value. It also makes it easier to create a design utilizing the data flow pragma since data streams from one task to the next.

3.7.3. Interface

```
#include "hls_stream.h"
#include "ap_axi_sdata.h"
typedef float DTYPE;
#define SIZE 256        /* SIZE OF DFT */
typedef ap_axis<32,2,5,6> transPkt;
union fp_int
{
    int i;
    float fp;
};
void dft(hls::stream<transPkt> &real_sample_in, hls::stream<transPkt> &imag_sample_in,
         hls::stream<transPkt> &real_sample_out, hls::stream<transPkt>
&imag_sample_out);
```

3.7.4. Test Bench

```
Rmse rmse_R,  rmse_I;

hls::stream<transPkt> In_R, In_I;
hls::stream<transPkt> Out_R, Out_I;

int main()
{
    int index;
    DTYPE gold_R, gold_I;
    fp_int real;     // for int<->float conversion
    fp_int imag;     // for int<->float conversion
    transPkt ipkt_r;
    transPkt ipkt_i;
    transPkt opkt_r;
    transPkt opkt_i;
    FILE * fp = fopen("out.gold.dat","r");
    // generate inputs and reference data
    for (int i = 0; i < SIZE; i++)
    {

        real.fp = i;                // prepare for "conversion"
        ipkt_r.data = real.i;       // pass bit-pattern *as-is*

        real.fp = 0.0;              // prepare for "conversion"
        ipkt_i.data = real.i;       // pass bit-pattern *as-is*

        In_R.write(ipkt_r);         // write data to stream
        In_I.write(ipkt_i);         // write data to stream
    }


    // DFT
```

```
    dft(In_R, In_I, Out_R, Out_I);

    // comparing with golden output
    for(int i=0; i<SIZE; i++)
    {

        opkt_r = Out_R.read();
        opkt_i = Out_I.read();
        real.i = opkt_r.data;
        imag.i = opkt_i.data;
        //printf("real.fp: %f \n",real.fp);
        fscanf(fp, "%d %f %f", &index, &gold_R, &gold_I);
        rmse_R.add_value(real.fp - gold_R);
        rmse_I.add_value(imag.fp - gold_I);

    }
    fclose(fp);
```

3.7.5. DFT Top-Level Function

```
void dft(hls::stream<transPkt> &real_sample_in, hls::stream<transPkt> &imag_sample_in,
         hls::stream<transPkt> &real_sample_out, hls::stream<transPkt> &imag_sample_out)
{
#pragma HLS INTERFACE mode=axis
port=real_sample_in,imag_sample_in,real_sample_out,imag_sample_out
#pragma HLS INTERFACE mode=s_axilite port=return
    int i, j;
    ap_uint<8> index;
    DTYPE w;
    DTYPE c, s;

    DTYPE temp_real[SIZE];
    DTYPE temp_imag[SIZE];
    DTYPE In_R[SIZE], In_I[SIZE];
    fp_int real;     // for int<->float conversion
    fp_int imag;     // for int<->float conversion
    transPkt ipkt_r;
    transPkt ipkt_i;
    transPkt opkt_r;
    transPkt opkt_i;

    for(int i=0; i<SIZE; i++)
    {
        ipkt_r = real_sample_in.read();
        ipkt_i = imag_sample_in.read();
        real.i = ipkt_r.data;
        imag.i = ipkt_i.data;
        In_R[i] = real.fp;
        In_I[i] = imag.fp;
    }


    for (i = 0; i < SIZE; i += 1)
    {
#pragma HLS PIPELINE off
        temp_real[i] = 0;
        temp_imag[i] = 0;
        for (j = 0; j < SIZE; j += 1)
        {
#pragma HLS PIPELINE off
            index = i*j;

            c = cos_coefficients_table[index];
            s = sin_coefficients_table[index];
            // Multiply the current phasor with the appropriate input sample and keep
            // running sum
            temp_real[i] += (In_R[j] * c - In_I[j] * s);
```

```
            temp_imag[i] += (In_R[j] * s + In_I[j] * c);

        }
    }

    for (i = 0; i < SIZE; i += 1)
    {
        real.i = temp_real[i];
        imag.i = temp_imag[i];
        opkt_r.data = real.i;
        opkt_i.data = imag.i;
        real_sample_out.write(opkt_r);
        imag_sample_out.write(opkt_i);
    }
}
```