CSE237C: Lab 3 Discrete Fourier Transform

Weihong Xu
wexu@ucsd.edu

November 3, 2020

1 Introduction

This report includes several types of designed architectures that implement the Discrete Fourier Transform (DFT) by matrix-vector multiplication. The goal is to develop functionally correct and high-throughput DFT architectures.

2 Baseline

2.1 Implementation

The DFT design with transform length $N = 256$ is implemented. The cos and sin values of are claculated using the HLS math functions. The folder ***dft_256_baseline*** include functionally correct designs of the DFT baseline implementations. The core function in *dft.cpp* is given as follows:

```
const float w = 2 * 3.1415926535 / SIZE;

void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
        //Write your code here
        unsigned char i, j;
        DTYPE dft_real[SIZE];
        DTYPE dft_img[SIZE];
        float angle;

        DFT:
        for (i = 0; i < SIZE; i++)
        {
                dft_real[i] = 0.0;
                dft_img[i] = 0.0;

                for (j = 0; j < SIZE; j++)
                {
                        angle = w * i * j;
                        dft_real[i] += real_sample[j] * cos(angle);
                        dft_img[i] += -real_sample[j] * sin(angle);
                }
        }

        for (i = 0; i < SIZE; i++)
        {
                real_sample[i] = dft_real[i];
                imag_sample[i] = dft_img[i];
        }
}
```

The baseline implementation is general and can be applied to arbitrary transform lengths. The functionality of the designs is verified using the testbench. Given the clock period constraint of 10 ns, the synthesis results are shown as follows:

Table 1: Synthesis results and throughput comparison for baseline designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	$4,590,850$	$4,590,850$	18	207	$13,026$	$19,122$	0.025

It is observed that DFT baseline designs with different lengths consume approximately the equal hardware resources. However, the serial nature of computation and the long transform length of Baseline $N = 256$ degrade the throughput to only 0.025 KHz.

2.2 Question 1

- *What changes would this code require if you were to use a custom CORDIC similar to what you designed for Project 2?*

  If the custom CORDIC were used to calculate cos() and sin() values, three modifications should be made to fit the requirements of DFT computation. First, the input and output of CORDIC function should be changed. The CORDIC function in Project 2 accepts the coordinate as input and yields the angle as well as R. For Project 3, the input should be the to-be computed angle and the output should be the associated cos() or sin() values. To this end, the second change is to modify the termination condition, where the starting angle is $\pi/2$ or 0 while the terminated angle is the to-be computed angle. Finally, due to the calculated angle of DFT could exceed the range of $[-\pi, \pi]$, additional pre-processings should be created to rotate the initial angle that falls into the $[-\pi, \pi]$ range.

- *Compared to a baseline code with HLS math functions for cos() and sin(), would changing the accuracy of your CORDIC core make the DFT hardware resource usage change? How would it affect the performance? Note that you do not need to implement the CORDIC in your code, we are just asking you to discuss potential tradeoffs that would be possible if you used a CORDIC that you designed instead of the one from Xilinx.*

  The hardware resource usage could be decreased if fixed-point CORDIC designs were used. To change the accuracy of CORDIC core, the rotation number should be adjusted, which affect the consumption of memory resources like BRAM, FF, and LUT.

But the arithmetic resources may keep the same. Considering CORDIC is calculated in the iterative manner, the overall latency of DFT that adopts CORDIC would become larger. Thus the resulting performance would be degraded compared to baseline DFT designs. To shorten the processing latency, the CORDIC can be realized using table lookup method.

3 Design and Optimization

3.1 Question 2

3.1.1 Implementation

The complexity of DFT baseline comes from the math functions to calculate sin() and cos(). The complex math function calls are eliminated by looking up the pre-computed sin() and cos() values. The folder ***dft_256_precomputed*** includes the associated implementation.

```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
  //Write your code here
  int i, j;
  DTYPE dft_real[SIZE];
  DTYPE dft_img[SIZE];
  int index;

  DFT:
  for (i = 0; i < SIZE; i++)
  {
    dft_real[i] = 0.0;
        dft_img[i] = 0.0;

        for (j = 0; j < SIZE; j++)
        {
          index = (i * j) % SIZE;
          dft_real[i] += real_sample[j] * cos_coefficients_table[index];
          dft_img[i] += real_sample[j] * sin_coefficients_table[index];
        }
  }

  for (i = 0; i < SIZE; i++)
  {
        real_sample[i] = dft_real[i];
        imag_sample[i] = dft_img[i];
  }
}
```

The design passes the simulation and matches the ground truth results. The synthesis results and corresponding performance for baseline and precomputed designs are given in Table 2. The design presented in this section is called **Precomputed, N=256**.

Table 2: Synthesis results and throughput comparison for various designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	$4,590,850$	$4,590,850$	18	207	$13,026$	$19,122$	0.025
Precomputed, $N = 256$	7.424	$787,458$	$787,458$	4	10	$1,087$	$1,743$	0.171

3.1.2 Analysis

- *Rewrite the code to eliminate these math function calls (i.e. cos() and sin()) by utilizing a table lookup. How does this change the throughput and area? What happens to the table lookup when you change the size of your DFT?*

  After rewriting the code with table lookup, the throughput increases to 0.171 KHz by a speedup factor of $6.84\times$. The improvement of throughput is due to the simplified arithmetic operations that dramatically shorten the iteration cycles. Meanwhile, the area (hardware resource consumption) is also reduced. When the DFT size is changed from n to N, the latency and interval increase with a factor of $(\frac{N}{n})^2$ while the costed DSP48E, FF, and LUT basically stay unchanged. For long lengths, part of constant cos and sin coefficients will be synthesized as BRAM.

3.2 Question 3

3.2.1 Implementation

The input and output interfaces are separated in this section. The corresponding design is called **Opt. 1, N=256**. The synthesis results and performance comparison are listed in Table 3.

Table 3: Synthesis results and throughput comparison for different designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	$4,590,850$	$4,590,850$	18	207	$13,026$	$19,122$	0.025
Precomputed, $N = 256$	7.424	$787,458$	$787,458$	4	10	$1,087$	$1,743$	0.171
Opt. 1, $N = 256$	7.424	$786,945$	$786,945$	2	10	$1,058$	$1,599$	0.171

3.2.2 Analysis

- *Modify the DFT function interface so that the input and outputs are stored in separate arrays. How does this affect the optimizations that you can perform? How does it change the performance? What about the area results? Modify your testbench to accommodate this change to DFT interface.*

  After separating the input and output interfaces, some part of data dependence is removed, which is potentially beneficial for the pipelining optimization. The DFT results do not have to wait until the whole computing process is finished. Instead, the data can be continuously streamed to the output as every outer iteration completes. The BRAM consumption reduces from 4 to 2 because the arrays to store temporary results are removed. But other metrics, namely DSP, FF and LUT, only receive marginally improvement.

3.3 Question 4

3.3.1 Implementation

The loop unrolling and array partitioning are investigated in this section. First, the inner loop of DFT is unrolled while the array partitioning is not used. The unrolling factors range from 2 to 64. The comparison is shown in Figure 1.

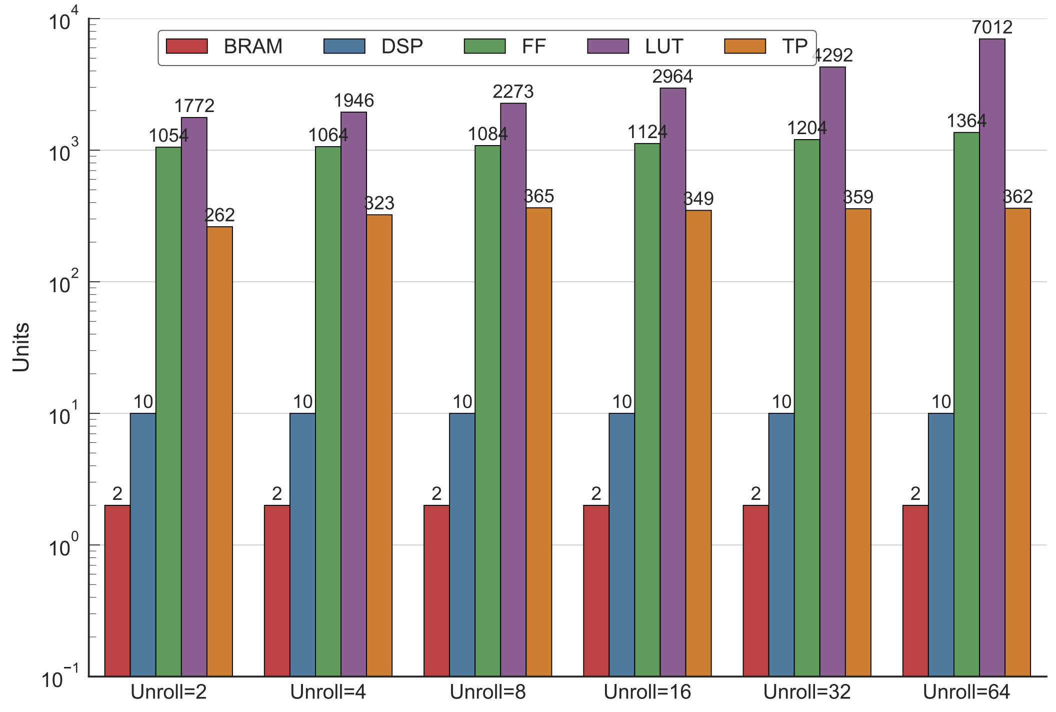

Figure 1: Trend of resource consumption and matrix-vector multiplication throughput under different unroll factors.

Then I also tested the optimized designs with loop unrolling and array partitioning. For convenience, the unroll factor is equal to partitioning factor and the array partitioning type is *cyclic*. The comparison results are shown in Figure 2.

I select the the design with unrolling factor 16 as the optimal design, which is **Opt. 2, N=256**. The synthesis results are shown in Table 4.

3.3.2 Analysis

- *What is the relationship between array partitioning and loop unrolling? Does it help to perform one without the other? Plot the performance in terms of number of matrix vector multiply operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for area (showing LUTs, FFs, DSP blocks, BRAMs). What is the general trend in both cases? Which design would you select? Why?*

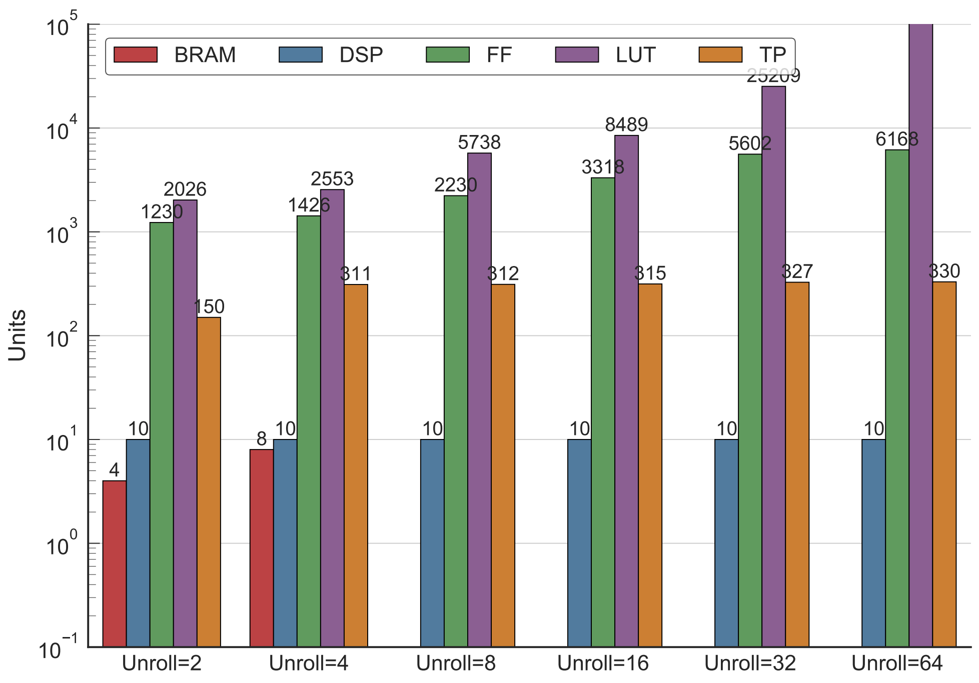

Figure 2: Trend of resource consumption and matrix-vector multiplication throughput under different unroll and partition factors, where the partition factor equals to unroll factor.

Table 4: Synthesis results and throughput comparison for different designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	4,590,850	4,590,850	18	207	13,026	19,122	0.025
Precomputed, $N = 256$	7.424	787,458	787,458	4	10	1,087	1,743	0.171
Opt. 1, $N = 256$	7.424	786,945	786,945	2	10	1,058	1,599	0.171
Opt. 2, $N = 256$	8.317	377,345	377,345	2	10	1,084	2,273	0.357

The array partitioning does not help loop unrolling to gain additional throughput improvements. As the unrolling factors go up, the consumption of FF and LUT grow exponentially. However, the throughput in terms of matrix-vector multiplication plateaus after unrolling factor 8. The design with loop unrolling factor 8 is selected because it achieves the highest hardware efficiency.

3.4 Question 5

3.4.1 Implementation

3.4.2 Analysis

- *How much improvement can you make with it? How much does your design use resources? What about BRAM usage? Please describe your architecture with figures on your report.*

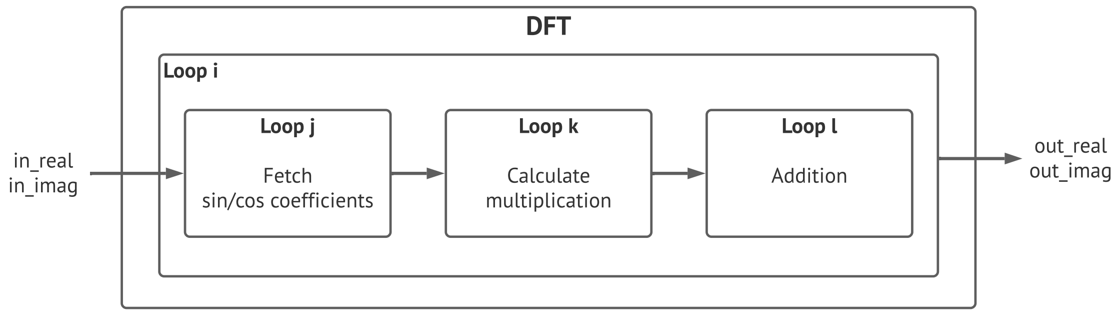

Figure 3: DFT dataflow architecture.

Table 5: Synthesis results and throughput comparison for different designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	$4,590,850$	$4,590,850$	18	207	$13,026$	$19,122$	0.025
Precomputed, $N = 256$	7.424	$787,458$	$787,458$	4	10	$1,087$	$1,743$	0.171
Opt. 1, $N = 256$	7.424	$786,945$	$786,945$	2	10	$1,058$	$1,599$	0.171
Opt. 2, $N = 256$	8.317	$377,345$	$377,345$	2	10	$1,084$	$2,273$	0.357
Dataflow, $N = 256$	7.424	$461,574$	$461,574$	6	10	$1,131$	$2,140$	0.291

From the synthesis results in Table 5, the dataflow architecture increases the BRAM consumption since it requires multiple FIFOs/RAMs to store and synchronize the intermediate results. The DSP, FF, and LUT consumption stays almost the same.

After inserting the dataflow pragma into the inner loop of DFT, the dataflow architecture for DFT is illustrated in Figure 3. The inner with dataflow pragma consists of three main loops. These three loops are executed sequentially. The dataflow realizes module-level pipelining to shortens the critical path of the system to 7.424 ns.

3.5 Question 6

Table 6: Synthesis results and throughput comparison for different designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	$4,590,850$	$4,590,850$	18	207	$13,026$	$19,122$	0.025
Precomputed, $N = 256$	7.424	$787,458$	$787,458$	4	10	$1,087$	$1,743$	0.171
Opt. 1, $N = 256$	7.424	$786,945$	$786,945$	2	10	$1,058$	$1,599$	0.171
Opt. 2, $N = 256$	8.317	$377,345$	$377,345$	2	10	$1,084$	$2,273$	0.357
Dataflow, $N = 256$	7.424	$461,574$	$461,574$	6	10	$1,131$	$2,140$	0.291
Best, $N = 256$	8.317	$329,229$	$329,229$	10	14	$2,767$	$4,026$	0.365
Best, $N = 1024$	8.372	$5,249,037$	$5,249,037$	12	14	$3,320$	5578	0.023

3.5.1 Analysis

- *Briefly describe your "best" architecture. In what way is it the best? What optimizations did you use to obtain this result? What is tradeoff you consider for the best architecture?*

  On the basis of design **Opt. 3**, the pipelining and loop unrolling with factor 16 are inserted into the three most inner sub-loops. The results are summarized in Table 6. The best design slightly improves the throughput to 0.365 KHz at the cost of additional BRAM, DSP, FF, and LUT resources. The definition of "best" is in terms of the peak throughput that the design can yield.

3.6 Question 7 - Bonus

Table 7: Synthesis results and throughput comparison for different designs.

Design	Est. Clk (ns)	Clock Cycles		Utilization				T/P (KHz)
		Latency	Interval	BRAM	DSP48E	FF	LUT	
Baseline, $N = 256$	8.683	$4,590,850$	$4,590,850$	18	207	$13,026$	$19,122$	0.025
Precomputed, $N = 256$	7.424	$787,458$	$787,458$	4	10	$1,087$	$1,743$	0.171
Opt. 1, $N = 256$	7.424	$786,945$	$786,945$	2	10	$1,058$	$1,599$	0.171
Opt. 2, $N = 256$	8.317	$377,345$	$377,345$	2	10	$1,084$	$2,273$	0.357
Dataflow, $N = 256$	7.424	$461,574$	$461,574$	6	10	$1,131$	$2,140$	0.291
Best, $N = 256$	8.317	$329,229$	$329,229$	10	14	$2,767$	$4,026$	0.365
Best, $N = 1024$	8.372	$5,249,037$	$5,249,037$	12	14	$3,320$	5578	0.023
Stream, $N = 256$	7.424	$461,832$	$461,832$	7	10	$1,214$	2331	0.292

3.6.1 Analysis

- *Please briefly describe what benefit you can achieve with hls::stream and why?*

  The first benefit is the streaming architecture is more convenient and flexible to scale and invoke as an IP core since it has regular dataflow. The second benefit is the streaming architecture has fewer I/O ports, which saves the area caused by peripheral devices.