CSE 237C Project3 report

Yiming Ren

Y5ren@ucsd.edu A59005465

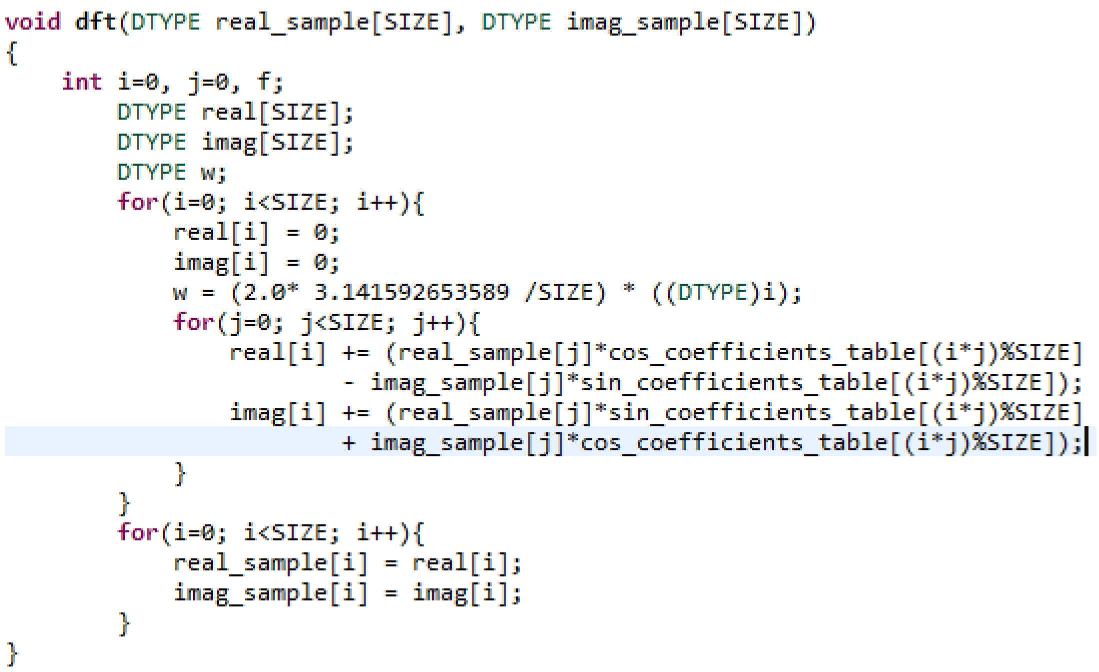
10/31/2020

Introduction

This project focuses on implementing and optimizing a design architecture for Discrete Fourier Transform (DFT). DFT is a common operation in signal processing which generates a frequency domain representation of the discrete input signal. Using the given coefficients array, we can simply generate the sampled cos and sin functions in frequency domain. They can also be expressed by complex numbers, and the real and imaginary part of the complex numbers form the output arrays.

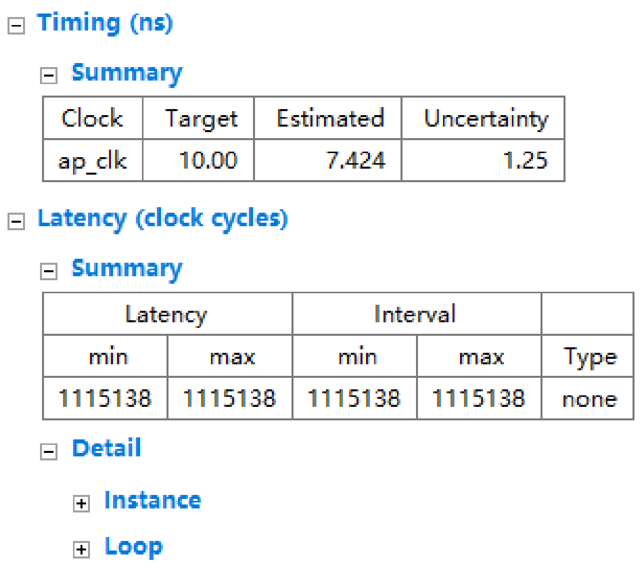
Baseline:

The below ‘dft’ function is used to generate DFT for a 256 sampled points function:



```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    int i=0, j=0, f;
        DTYPE real[SIZE];
        DTYPE imag[SIZE];
        DTYPE w;
        for(i=0; i<SIZE; i++){
            real[i] = 0;
            imag[i] = 0;
            w = (2.0* 3.141592653589 /SIZE) * ((DTYPE)i);
            for(j=0; j<SIZE; j++){
                real[i] += (real_sample[j]*cos_coefficients_table[(i*j)%SIZE]
                        - imag_sample[j]*sin_coefficients_table[(i*j)%SIZE]);
                imag[i] += (real_sample[j]*sin_coefficients_table[(i*j)%SIZE]
                        + imag_sample[j]*cos_coefficients_table[(i*j)%SIZE]);
            }
        }
        for(i=0; i<SIZE; i++){
            real_sample[i] = real[i];
            imag_sample[i] = imag[i];
        }
}
```

The following synthesis result shows the performance and resource usage conditions of the implementation. It will be used to compare the efficiency of each following optimizations.



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.424	1.25

Latency (clock cycles)

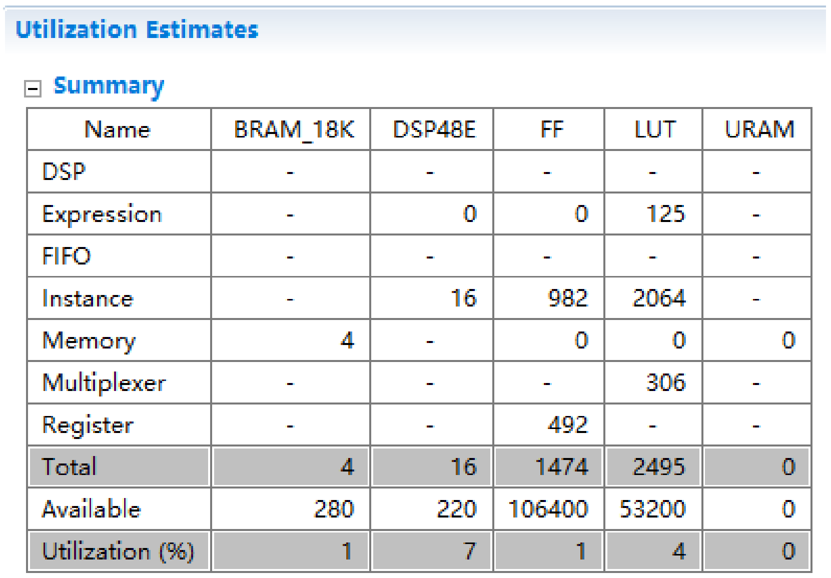
Summary

Latency		Interval		
min	max	min	max	Type
1115138	1115138	1115138	1115138	none

Detail

Instance

Loop



Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	0	0	125	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	306	-
Register	-	-	492	-	-
Total	4	16	1474	2495	0
Available	280	220	106400	53200	0
Utilization (%)	1	7	1	4	0

From the result, we can calculate the throughput of baseline implementation using clock period and cycles:

Throughput f = 1000/7.424/1115138 = 0.1208kHz

Question 1:

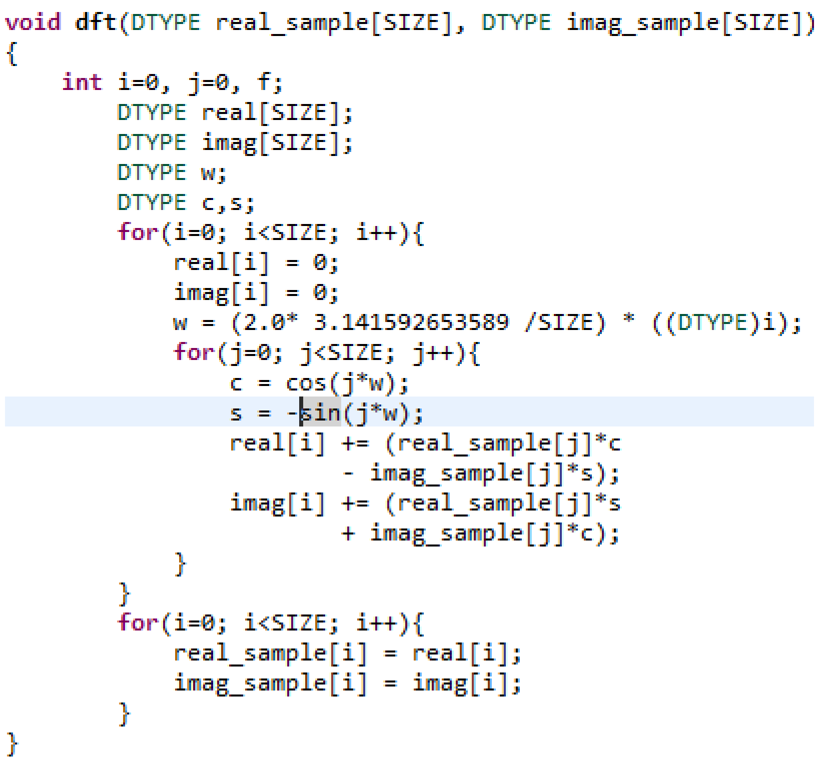
Unlike CORDIC, the DFT function saves the real and imaginary parts of all sampled points. To deploy a CORDIC in a DFT function, we first need to create an array to save the coordinates of every rotation in order. Then we need to modify details in the CORDIC. Since we want to use the given coefficients to form a matrix, instead of the decreasing factors in CORDIC, we have to modify the existing 'Kvalues' and 'angles' arrays, to make them corresponds to the angle and distance to the origin of sampled points. Now, we are good to replace the inner loop of DFT to CORDIC.

Cos() and sin() functions in <math.h> are complex functions that requires more resources than simple operations such as '+' and shift. Obviously, cos() and sin() use less BRAM space since they do not require memory for a coefficient table, but they do need far more FFs and DSP48E slices. So overall, the resources usage is better on a CORDIC core.

The influence on performance depends on the number of cycles needed for doing math functions and accessing memories. Usually, accessing BRAM requires less clock cycles than doing arithmetic operations on DSP48E. Therefore, when pipelining the whole function, CORDIC core could obtain a shorter II and hence a better performance.

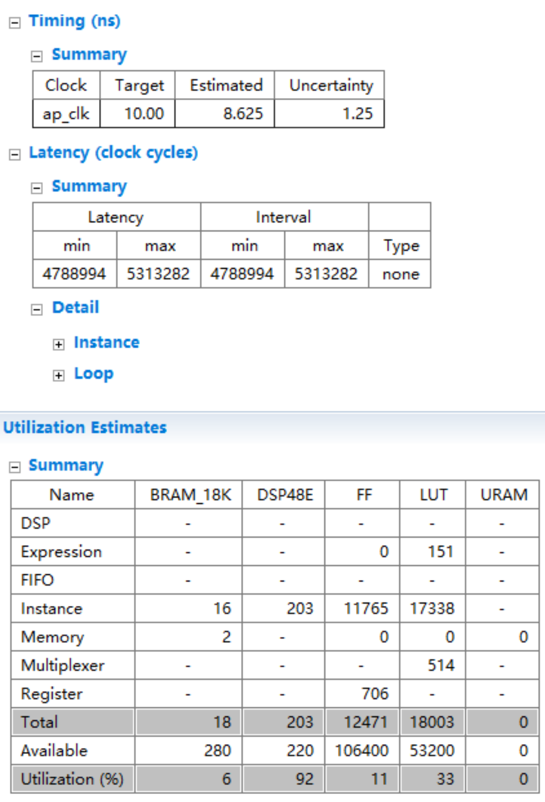
Question 2:

Since I already eliminate all cos() and sin() functions in my baseline, I will re-insert them into my code for this question. The code with them and the synthesis reports will be in the optimized1 folder. Code with cos() and sin():



```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    int i=0, j=0, f;
        DTYPE real[SIZE];
        DTYPE imag[SIZE];
        DTYPE w;
        DTYPE c,s;
        for(i=0; i<SIZE; i++){
            real[i] = 0;
            imag[i] = 0;
            w = (2.0* 3.141592653589 /SIZE) * ((DTYPE)i);
            for(j=0; j<SIZE; j++){
                c = cos(j*w);
                s = -sin(j*w);
                real[i] += (real_sample[j]*c
                        - imag_sample[j]*s);
                imag[i] += (real_sample[j]*s
                        + imag_sample[j]*c);
            }
        }
        for(i=0; i<SIZE; i++){
            real_sample[i] = real[i];
            imag_sample[i] = imag[i];
        }
}
```

The code functions correctly and passes the C-simulation. Compare its synthesis with baseline's:



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.625	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
4788994	5313282	4788994	5313282	none

Detail

Instance

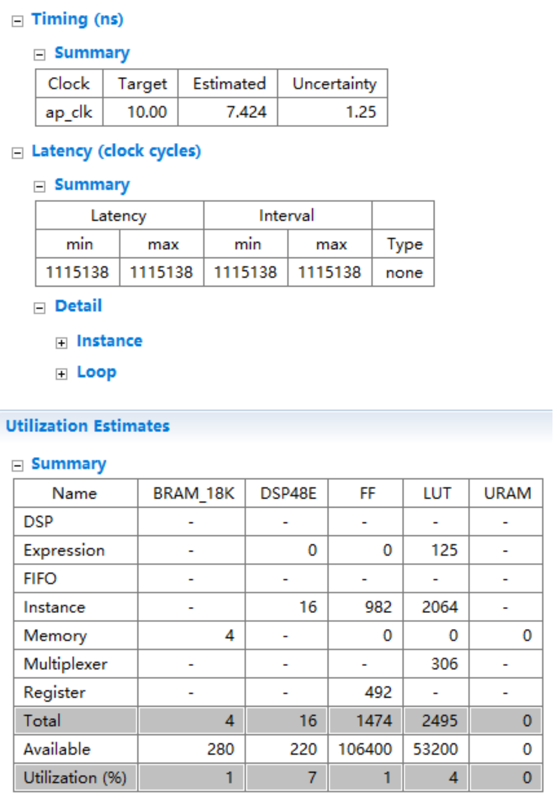
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	151	-
FIFO	-	-	-	-	-
Instance	16	203	11765	17338	-
Memory	2	-	0	0	0
Multiplexer	-	-	-	514	-
Register	-	-	706	-	-
Total	18	203	12471	18003	0
Available	280	220	106400	53200	0
Utilization (%)	6	92	11	33	0

Optimized 1



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.424	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1115138	1115138	1115138	1115138	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	0	0	125	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	306	-
Register	-	-	492	-	-
Total	4	16	1474	2495	0
Available	280	220	106400	53200	0
Utilization (%)	1	7	1	4	0

baseline

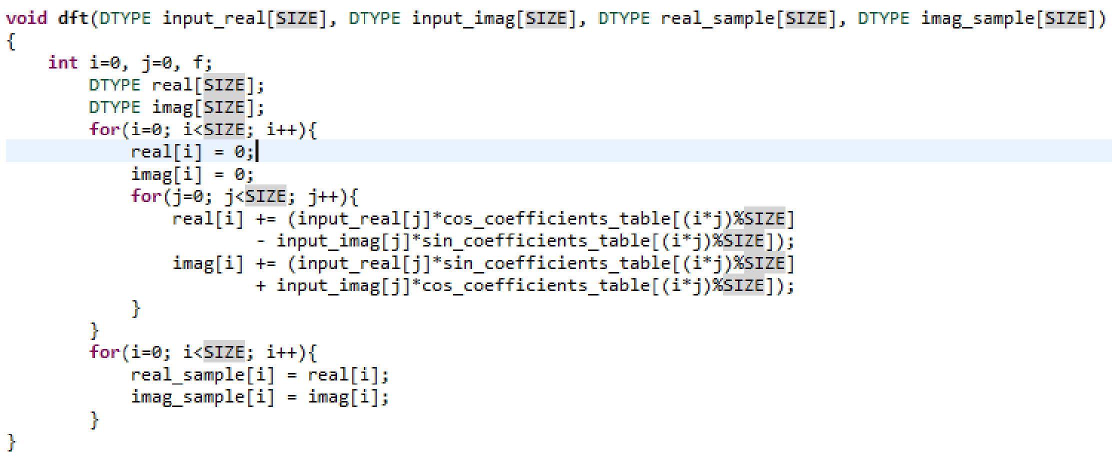
Optimized 1 throughput: 1000/8.625/5313282 = 21.82 Hz

Compared to the baseline, optimization 1 seems like a disaster to both overall resource usage and performance. The surge of latency proves my assumption in Question 1, which is that the arithmetic operations in DSP48E for cos() and sin() functions requires more clock cycles than accessing the BRAM. This could also be explained by the large number of basic blocks used in cos() and sin(). First, cos() and sin() functions require a lot of FFs to hold the data in arithmetic operation as expect. They also take more LUTs to look for the result of arithmetic calculation. The amount of time taken to access those units together lead to the explosion of the time spent by cos() and sin() functions.

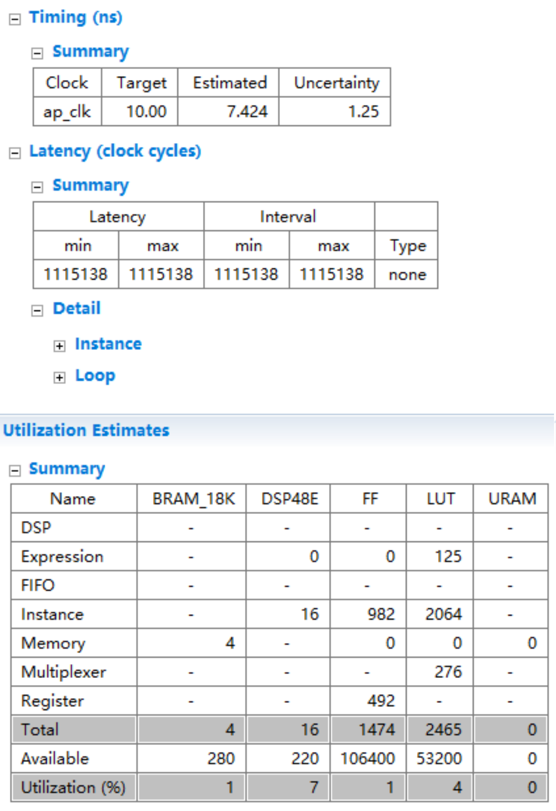
When I change the size of DFT, the density of the table lookup also changes. Specifically, larger size of DFT means denser table lookup, because larger size of DFT needs more sampled points. Those sampled points are from the division of the sine or cosine function in one period. If I need more sampled points, I need to cut out the function more frequently. The values of these points are also closer as the points are getting closer. Consequently, the table lookup would contain more and closer numbers, being ‘denser’.

Question 3:

The implementation of isolated input and output array is easy. The only thing I need to change is assigning the results in the temporary local pointers ‘real’ and ‘imag’ to another 2 new pointers rather. Code and synthesis result compared to previous optimization:



```
void dft(DTYPE input_real[SIZE], DTYPE input_imag[SIZE], DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    int i=0, j=0, f;
        DTYPE real[SIZE];
        DTYPE imag[SIZE];
        for(i=0; i<SIZE; i++){
            real[i] = 0;
            imag[i] = 0;
            for(j=0; j<SIZE; j++){
                real[i] += (input_real[j]*cos_coefficients_table[(i*j)%SIZE]
                        - input_imag[j]*sin_coefficients_table[(i*j)%SIZE]);
                imag[i] += (input_real[j]*sin_coefficients_table[(i*j)%SIZE]
                        + input_imag[j]*cos_coefficients_table[(i*j)%SIZE]);
            }
        }
        for(i=0; i<SIZE; i++){
            real_sample[i] = real[i];
            imag_sample[i] = imag[i];
        }
}
```



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.424	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1115138	1115138	1115138	1115138	none

Detail

Instance

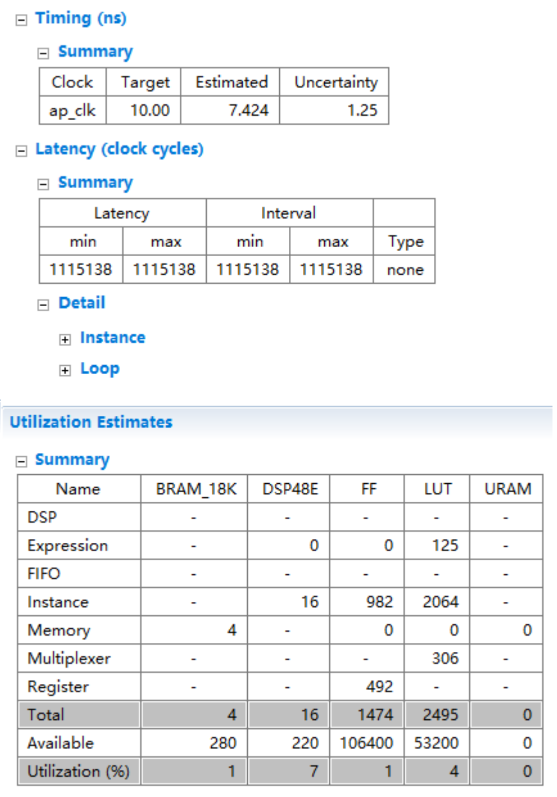
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	0	0	125	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	276	-
Register	-	-	492	-	-
Total	4	16	1474	2465	0
Available	280	220	106400	53200	0
Utilization (%)	1	7	1	4	0

Optimized 2



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.424	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
1115138	1115138	1115138	1115138	none

Detail

Instance

Loop

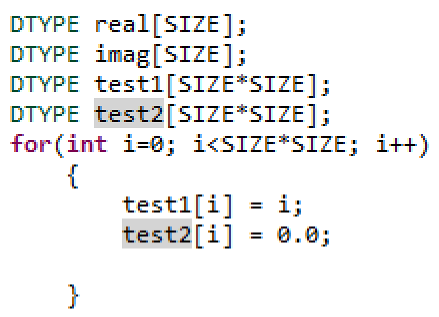
Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	0	0	125	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	306	-
Register	-	-	492	-	-
Total	4	16	1474	2495	0
Available	280	220	106400	53200	0
Utilization (%)	1	7	1	4	0

baseline

The result shows that separated input and output variables have nothing to do with the performance. This should not happen since for united input and output, we do save a space of sizeof(float)*SIZE in memory, and for separated input and output, we need extra time accessing another bunch of memory. To explain this, I created several DTYPE arrays along with the temporary real and imag arrays and copy the initialization loop in testbench to make sure I've access them:



```
DTYPE real[SIZE];
DTYPE imag[SIZE];
DTYPE test1[SIZE*SIZE];
DTYPE test2[SIZE*SIZE];
for(int i=0; i<SIZE*SIZE; i++)
    {
        test1[i] = i;
        test2[i] = 0.0;

    }
```

I am not going to show the synthesis result since they are totally same as the optimization 2. Now the truth is, the declared pointers are not stored in neither the BRAMs nor the LUTs. They also do not use any extra FFs. My guess is that they are in a cache or memory chip outside the FPGA in Vivado simulation, and the access time is also not counted in the Latency. When compiling, they do not occupy any registers. Instead, the compiler follows the pointer, reaching the memory and taking or writing the calculation results. These procedures do not spend clock cycles or are not counted in the Latency. So they do not influence the performance and areas.

Question 4:

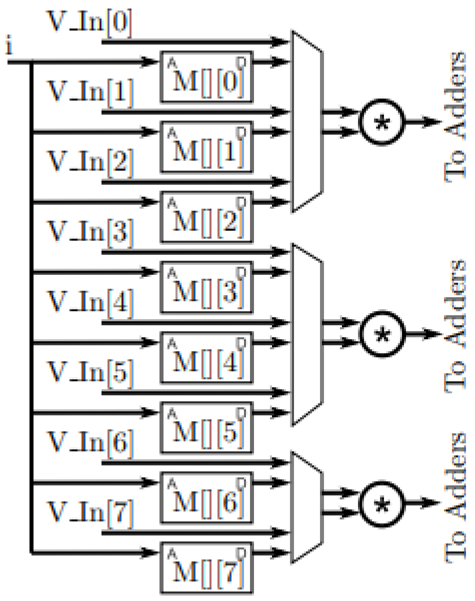
Loop unrolling and array partitioning are 2 basic methods before performing loop pipelines. They both could optimize the operations related to array structure in FPGA and increase local memory bandwidth. Loop unrolling would be nonsense without array partitioning, since loop pipeline requires multiple read/write ports for single array, while array is a single port memory structure in FPGA by default.

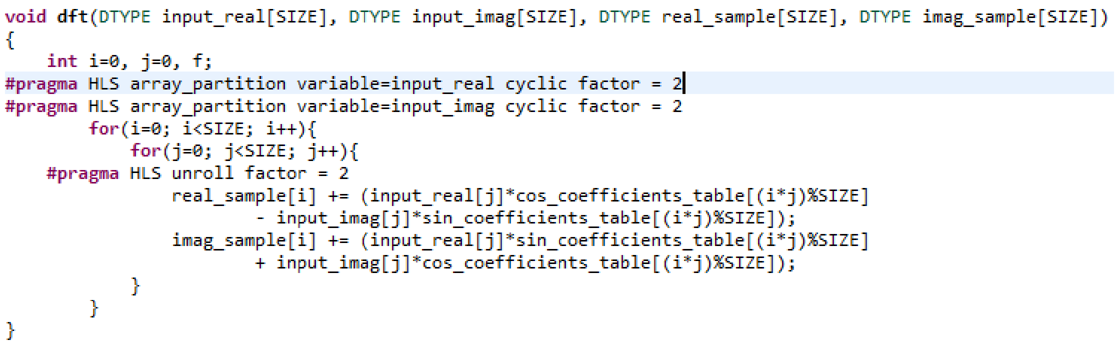
Array partitioning instruction in Vivado HLS:

#pragma HLS array_partition variable = <variable> <block, cyclic, complete>

factor = <int> dim = <int>

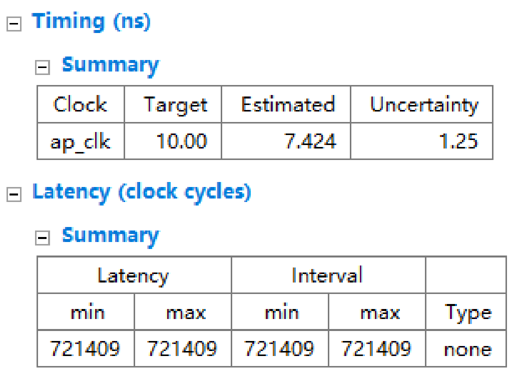
Theoretically, ‘Complete’ partitioning is not a good choice for a large size array since it could cause long runtime and large multiplexers (right figure). Similarly, I will also ignore the other partitioning methods with a factor of 256 or 128. Moreover, loop unrolling with a factor > 16 brings so many pressures to HLS that the clock period exceeds the requirement of 10ns. So the modified code would start testing begin with factor = 2, and gradually increase the factor till 16:





```
void dft(DTYPE input_real[SIZE], DTYPE input_imag[SIZE], DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    int i=0, j=0, f;
#pragma HLS array_partition variable=input_real cyclic factor = 2
#pragma HLS array_partition variable=input_imag cyclic factor = 2
        for(i=0; i<SIZE; i++){
            for(j=0; j<SIZE; j++){
    #pragma HLS unroll factor = 2
                real_sample[i] += (input_real[j]*cos_coefficients_table[(i*j)%SIZE]
                        - input_imag[j]*sin_coefficients_table[(i*j)%SIZE]);
                imag_sample[i] += (input_real[j]*sin_coefficients_table[(i*j)%SIZE]
                        + input_imag[j]*cos_coefficients_table[(i*j)%SIZE]);
            }
        }
}
```

The timing result of each:



⊟ Timing (ns)

⊟ Summary

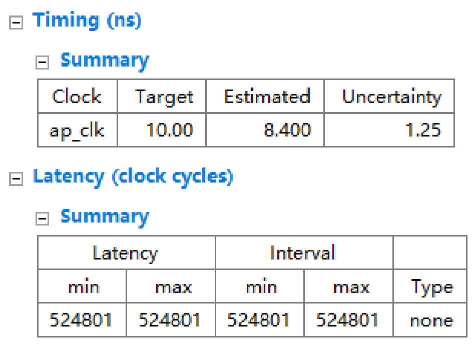
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	7.424	1.25

⊟ Latency (clock cycles)

⊟ Summary

Latency		Interval		
min	max	min	max	Type
721409	721409	721409	721409	none

Factor = 2. Throughput=1000/(721409*7.424)=0.1867kMz



⊟ **Timing (ns)**

⊟ **Summary**

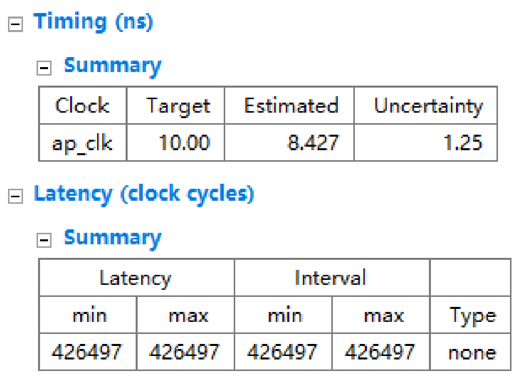
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.400	1.25

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
524801	524801	524801	524801	none

Factor = 4. Throughput = 1000/(524801*8.4)=0.2268kMz



⊟ **Timing (ns)**

⊟ **Summary**

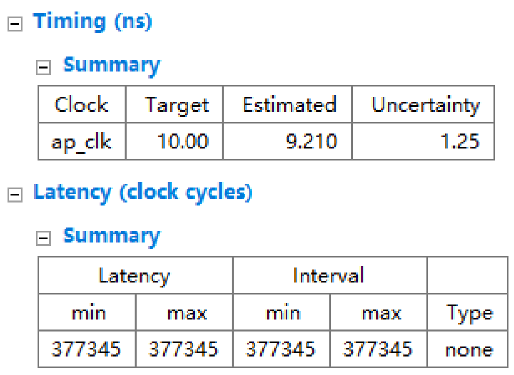
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.427	1.25

⊟ **Latency (clock cycles)**

⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
426497	426497	426497	426497	none

Factor = 8. Throughput = 1000/(426497*8.427)=0.2782kMz



⊟ **Timing (ns)**

⊟ **Summary**

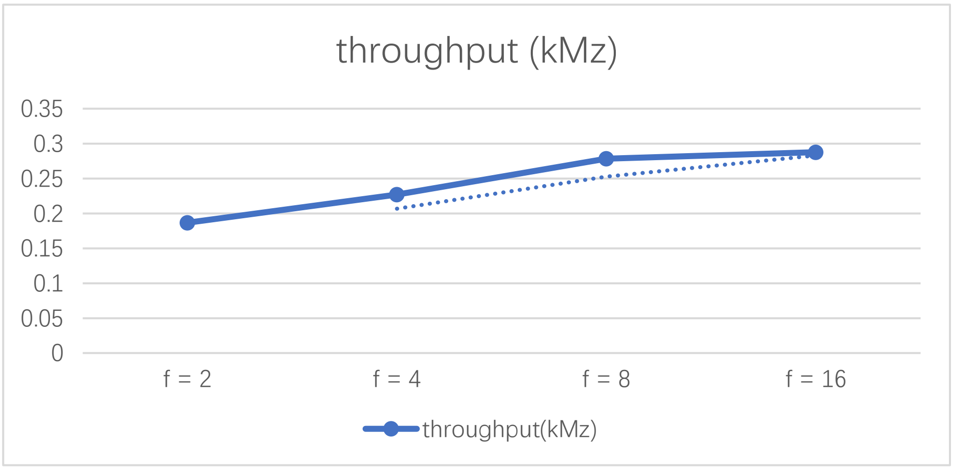
Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.210	1.25

⊟ **Latency (clock cycles)**

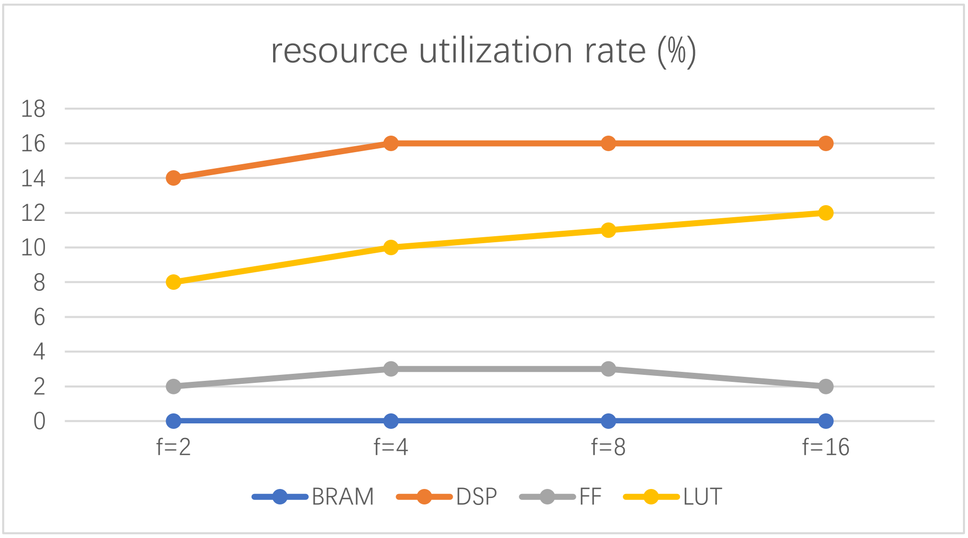
⊟ **Summary**

Latency		Interval		
min	max	min	max	Type
377345	377345	377345	377345	none

Factor = 16. Throughput = 1000/(9.21*377345)=0.2877kMz



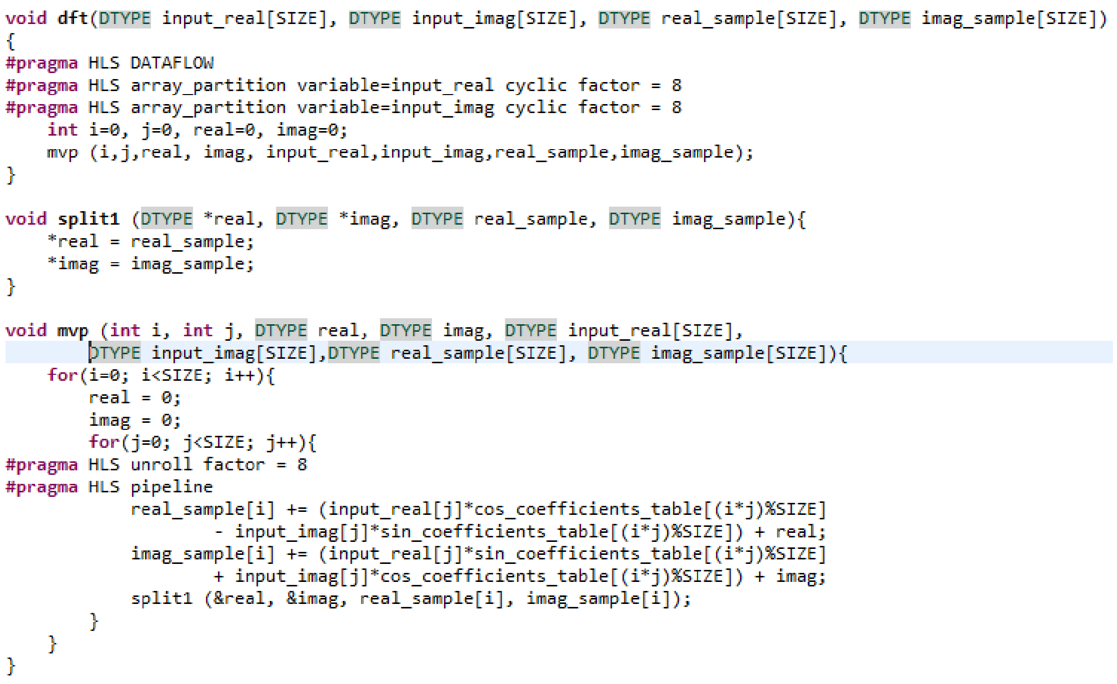
Area result plot:



By comparation, the best solution for loop unrolling and array partitioning is at factor = 8. At this time, the throughput starts to increase slowly. The throughput also approaches the throughput of factor = 16 but much more than throughput of factor = 4. Furthermore, considering the tradeoff between performance and area, the resources used at factor = 8 is more worthy than the others. The loop unrolling and array partitioning with factor = 8 are selected as the optimization 3.

Question 5:

Since dataflow is only allowed for local declaration and function call, I have to move the whole outer loop into a distinct function. Since I've already merged the 'copy' loop and 'calculation' loop, the only dataflow that can be realized is the dataflow from previous imag/real_sample[i] to the new imag/real_sample[i]. By creating a new 'split' function, I can make the data directly go to the next loop without 'storeward' instruction in current loop. The new implementation:

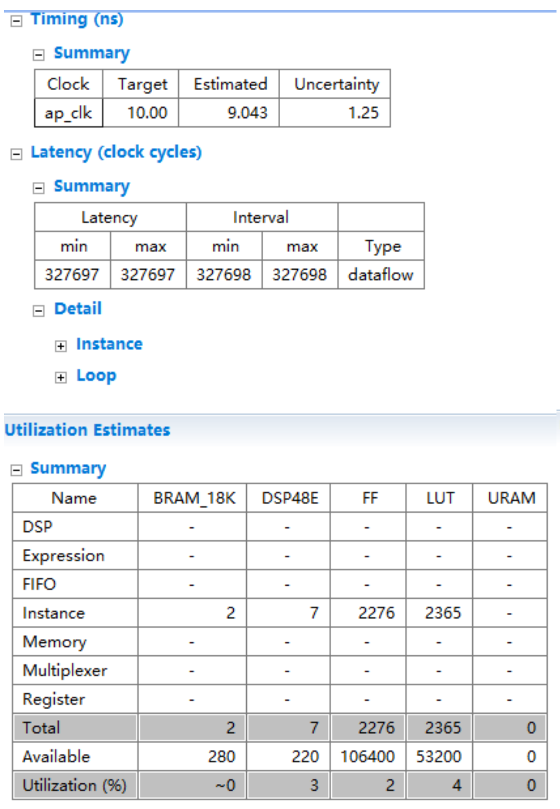


```
void dft(DTYPE input_real[SIZE], DTYPE input_imag[SIZE], DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
#pragma HLS DATAFLOW
#pragma HLS array_partition variable=input_real cyclic factor = 8
#pragma HLS array_partition variable=input_imag cyclic factor = 8
    int i=0, j=0, real=0, imag=0;
    mvp (i,j,real, imag, input_real,input_imag,real_sample,imag_sample);
}

void split1 (DTYPE *real, DTYPE *imag, DTYPE real_sample, DTYPE imag_sample){
    *real = real_sample;
    *imag = imag_sample;
}

void mvp (int i, int j, DTYPE real, DTYPE imag, DTYPE input_real[SIZE],
        DTYPE input_imag[SIZE],DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE]){
    for(i=0; i<SIZE; i++){
        real = 0;
        imag = 0;
        for(j=0; j<SIZE; j++){
#pragma HLS unroll factor = 8
#pragma HLS pipeline
            real_sample[i] += (input_real[j]*cos_coefficients_table[(i*j)%SIZE]
                    - input_imag[j]*sin_coefficients_table[(i*j)%SIZE]) + real;
            imag_sample[i] += (input_real[j]*sin_coefficients_table[(i*j)%SIZE]
                    + input_imag[j]*cos_coefficients_table[(i*j)%SIZE]) + imag;
            split1 (&real, &imag, real_sample[i], imag_sample[i]);
        }
    }
}
```

Comparation with previous optimization:



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.043	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
327697	327697	327698	327698	dataflow

Detail

Instance

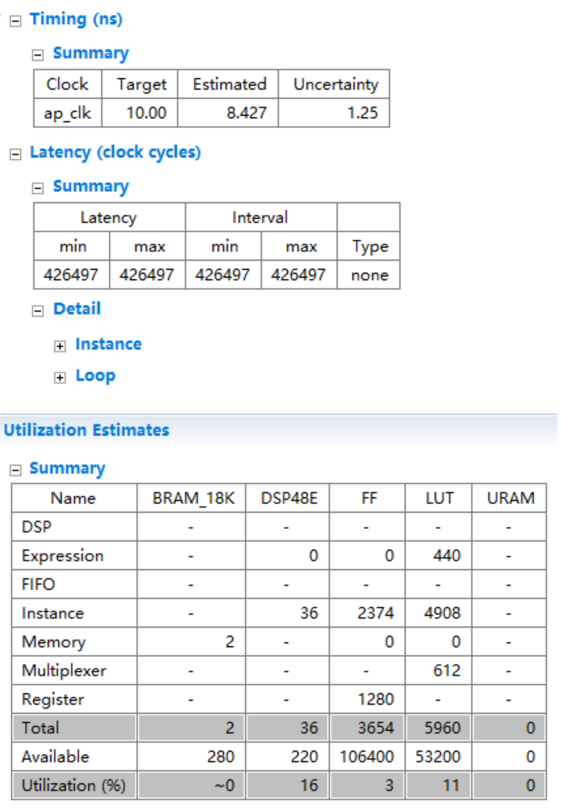
Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	2	7	2276	2365	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	-	-
Register	-	-	-	-	-
Total	2	7	2276	2365	0
Available	280	220	106400	53200	0
Utilization (%)	~0	3	2	4	0

Dataflow



Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	8.427	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
426497	426497	426497	426497	none

Detail

Instance

Loop

Utilization Estimates

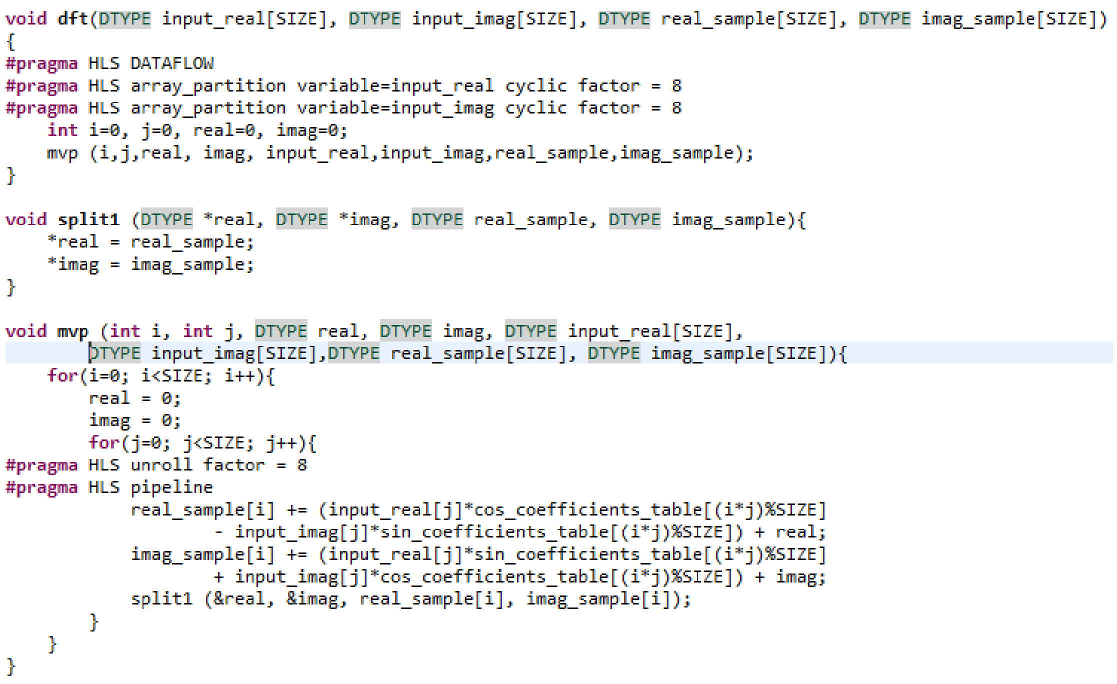
Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	0	0	440	-
FIFO	-	-	-	-	-
Instance	-	36	2374	4908	-
Memory	2	-	0	0	-
Multiplexer	-	-	-	612	-
Register	-	-	1280	-	-
Total	2	36	3654	5960	0
Available	280	220	106400	53200	0
Utilization (%)	~0	16	3	11	0

optimization 3 (unroll and array partition)

The throughput goes to 1000/(327698*9.043) = 0.3375kMz with dataflow, which is 21% more than optimization 3. The resource usage also improves a lot since I can avoid using resources other than instance resources. At this time, only utilization of BRAM stays the same, while utilization rates of others decrease more than a half. This in turn improves the performance since the machine does not have to access the memory, which costs much more time than adopting dataflow.

Question 6:



```
void dft(DTYPE input_real[SIZE], DTYPE input_imag[SIZE], DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
#pragma HLS DATAFLOW
#pragma HLS array_partition variable=input_real cyclic factor = 8
#pragma HLS array_partition variable=input_imag cyclic factor = 8
    int i=0, j=0, real=0, imag=0;
    mvp (i,j,real, imag, input_real,input_imag,real_sample,imag_sample);
}

void split1 (DTYPE *real, DTYPE *imag, DTYPE real_sample, DTYPE imag_sample){
    *real = real_sample;
    *imag = imag_sample;
}

void mvp (int i, int j, DTYPE real, DTYPE imag, DTYPE input_real[SIZE],
        DTYPE input_imag[SIZE],DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE]){
    for(i=0; i<SIZE; i++){
        real = 0;
        imag = 0;
        for(j=0; j<SIZE; j++){
#pragma HLS unroll factor = 8
#pragma HLS pipeline
            real_sample[i] += (input_real[j]*cos_coefficients_table[(i*j)%SIZE]
                    - input_imag[j]*sin_coefficients_table[(i*j)%SIZE]) + real;
            imag_sample[i] += (input_real[j]*sin_coefficients_table[(i*j)%SIZE]
                    + input_imag[j]*cos_coefficients_table[(i*j)%SIZE]) + imag;
            split1 (&real, &imag, real_sample[i], imag_sample[i]);
        }
    }
}
```

Now that every optimization is based on the previous optimization, optimization 4 in question 5 is the best architecture of my design. For this architecture, I gave up the cos() and sin() functions which require a lot of time and resource for arithmetic operations. Instead, a lookup table of coefficients is utilized. The loop unrolling and array partition are essential to pipeline the loops, so they are used for loop pipeline with a suitable factor. Dataflow is the last improvement in my architecture. It allows task-level pipelining and further decreases the loop interval.

Performance and area comparation:

	Best:	Basic:
Throughput (kMz):	0.3375	0.12
Utilization rate (%)		
BRAM:	~0	1
DSP:	3	7
FF:	2	1
LUT:	4	4

The best architecture spends a little bit more FFs than the baseline, but hugely improves the performance.