Briefly introduction to the algorithm or overall system

In this lab, we are required to do a discrete Fourier transform (DFT) kernel, and implemented it in PYNQ. The algorithm is shown in the formula below.

$$X[k] = \sum_{n=0}^{N-1} e^{-\frac{i2\pi}{N}nk} x[n]$$

We perform the real part and imaginary part separately, so the formula can be rewritten in this way.

$$X[k] = \sum_{n=0}^{N-1} [\cos\left(\frac{2\pi}{N}nk\right) - i\sin\left(\frac{2\pi}{N}nk\right)](x_R[n] + ix_I[n])$$

$$X_R[k] = \sum_{n=1}^{N-1} \cos\left(\frac{2\pi}{N}nk\right) x_R[n] + \sin\left(\frac{2\pi}{N}nk\right) x_I[n]$$

$$X_I[k] = \sum_{n=1}^{N-1} \cos\left(\frac{2\pi}{N}nk\right) x_I[n] - \sin\left(\frac{2\pi}{N}nk\right) x_R[n]$$

Explain the original code/system/pragmas and how you implement it

Here, I will show the baseline code. In the next section, I will implement optimization step by step following the questions of the guide.

```
void read(DTYPE real_in[SIZE], DTYPE imag_in[SIZE], DTYPE real_out[SIZE], DTYPE imag_out[SIZE])
{
    int i;

READ_LOOP:
    for (i = 0; i < SIZE; i++) {
        real_out[i] = real_in[i];
        imag_out[i] = imag_in[i];
    }

    return;
}
```

The function in the above figure is used to read the input.

```
void dft_compute(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE], DTYPE Real_freq[SIZE], DTYPE Imag_freq[SIZE])
{
    int k = 0;
    int n = 0;

DFT_OUTER_LOOP:
    for (k = 0; k < SIZE; k++) {
        Real_freq[k] = 0;
        Imag_freq[k] = 0;
DFT_INNER_LOOP:
        for (n = 0; n < SIZE; n++) {
            Real_freq[k] += real_sample[n] * cos(2*M_PI*n*k/SIZE) + imag_sample[n] * sin(2*M_PI*n*k/SIZE);
            Imag_freq[k] += imag_sample[n] * cos(2*M_PI*n*k/SIZE) - real_sample[n] * sin(2*M_PI*n*k/SIZE);
        }
    }

    return;
}
```

The function in the above figure is used to compute DFT.

```
void write(DTYPE real_in[SIZE], DTYPE imag_in[SIZE], DTYPE real_out[SIZE], DTYPE imag_out[SIZE])
{
    int i;

WRITE_LOOP:
    for (i = 0; i < SIZE; i++) {
        real_out[i] = real_in[i];
        imag_out[i] = imag_in[i];
    }

    return;
}
```

The function in the figure above is used to output the result.

```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    //Write your code here
    DTYPE real_in[SIZE], imag_in[SIZE];
    DTYPE real_out[SIZE], imag_out[SIZE];

    read(real_sample, imag_sample, real_in, imag_in);

    dft_compute(real_in, imag_in, real_out, imag_out);

    write(real_out, imag_out, real_sample, imag_sample);

    return;
}
```

The 3 functions introduced above are called here.

Analyze the timing/performance/utilization

1. LookUpTable

```
DFT_INNER_LOOP:
        for (n = 0; n < SIZE; n++) {
            Real_freq[k] += real_sample[n] * cos_coefficients_table[n*k%SIZE] - imag_sample[n] * sin_coefficients_table[n*k%SIZE];
            Imag_freq[k] += imag_sample[n] * cos_coefficients_table[n*k%SIZE] + real_sample[n] * sin_coefficients_table[n*k%SIZE];
        }
```

The inner loop is modified in this way so that we don't need to calculate cos and sin functions.

How does this change the throughput and resource utilization?

Throughput

Baseline

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ dft	-	0.00	7015171	7.015e+07	-	7015172	-	no	20 (7%)	158 (71%)	15552 (14%)	18256 (34%)	-
o READ_LOOP	-	7.30	512	5.120e+03	2	-	256	no	-	-	-	-	-
o DFT_OUTER_LOOP	-	7.30	7014144	7.014e+07	27399	-	256	no	-	-	-	-	-
o DFT_INNER_LOOP	-	7.30	27392	2.739e+05	107	-	256	no	-	-	-	-	-
+ sin_or_cos_double_s	-	0.25	55	550.000	-	55	-	no	8 (2%)	54 (24%)	5933 (5%)	6480 (12%)	-
o Loop 1	-	7.30	9	90.000	3	-	3	no	-	-	-	-	-
o Loop 2	-	7.30	4	40.000	1	-	4	no	-	-	-	-	-
o Loop 3	-	7.30	9	90.000	2	-	4	no	-	-	-	-	-
+ sin_or_cos_double_s	-	0.25	55	550.000	-	55	-	no	8 (2%)	54 (24%)	5933 (5%)	6480 (12%)	-
o Loop 1	-	7.30	9	90.000	3	-	3	no	-	-	-	-	-
o Loop 2	-	7.30	4	40.000	1	-	4	no	-	-	-	-	-
o Loop 3	-	7.30	9	90.000	2	-	4	no	-	-	-	-	-
o WRITE_LOOP	-	7.30	512	5.120e+03	2	-	256	no	-	-	-	-	-

Lookup table

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ dft	-	0.04	1050115	1.050e+07	-	1050116	-	no	6 (2%)	16 (7%)	1500 (1%)	2509 (4%)	-
o READ_LOOP	-	7.30	512	5.120e+03	2	-	256	no	-	-	-	-	-
o DFT_OUTER_LOOP	-	7.30	1049088	1.049e+07	4098	-	256	no	-	-	-	-	-
o DFT_INNER_LOOP	-	7.30	4096	4.096e+04	16	-	256	no	-	-	-	-	-
o WRITE_LOOP	-	7.30	512	5.120e+03	2	-	256	no	-	-	-	-	-

From the interval, we can calculate the throughput.

Baseline: $1 \div (7015172 \times 10ns) = 14.25$

Lookup table: $1 \div (1050116 \times 10ns) = 95.22$

Resource allocation

Baseline

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	102	-
FIFO	-	-	-	-	-
Instance	16	158	14494	17586	-
Memory	4	-	0	0	0
Multiplexer	-	-	-	568	-
Register	-	-	1058	-	-
Total	20	158	15552	18256	0
Available	280	220	106400	53200	0
Utilization (%)	7	71	14	34	0

Lookup table

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	115	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	330	-
Register	-	-	518	-	-
Total	6	16	1500	2509	0
Available	280	220	106400	53200	0
Utilization (%)	2	7	1	4	0

From the above result, we can see that the throughput is greatly increased. Besides, the resource utilization also decreases a lot. Therefore, lookup table is very helpful for DFT.

What happens to the table lookup when you change the size of your DFT?

The size of lookup table is the same as the size of DFT.

2. IOArray

Original

```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE])
{
    //Write your code here
    DTYPE real_in[SIZE], imag_in[SIZE];
    DTYPE real_out[SIZE], imag_out[SIZE];

    read(real_sample, imag_sample, real_in, imag_in);

    dft_compute(real_in, imag_in, real_out, imag_out);

    write(real_out, imag_out, real_sample, imag_sample);

    return;
}
```

New

```
void dft(DTYPE real_sample[SIZE], DTYPE imag_sample[SIZE], DTYPE Real_freq[SIZE], DTYPE Imag_freq[SIZE])
{
    //Write your code here
    DTYPE real_in[SIZE], imag_in[SIZE];
    DTYPE real_out[SIZE], imag_out[SIZE];

    read(real_sample, imag_sample, real_in, imag_in);

    dft_compute(real_in, imag_in, real_out, imag_out);

    write(real_out, imag_out, Real_freq, Imag_freq);

    return;
}
```

The function arguments are modified in this way.

How does it change the performance?

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval	Trip Count	Pipelined	BRAM	DSP	FF	LUT	URAM
+ dft	-	0.04	1050115	1.050e+07	-	1050116	-	no	6 (2%)	16 (7%)	1500 (1%)	2481 (4%)	-
o READ_LOOP	-	7.30	512	5.120e+03	2	-	256	no	-	-	-	-	-
o DFT_OUTER_LOOP	-	7.30	1049088	1.049e+07	4098	-	256	no	-	-	-	-	-
o DFT_INNER_LOOP	-	7.30	4096	4.096e+04	16	-	256	no	-	-	-	-	-
o WRITE_LOOP	-	7.30	512	5.120e+03	2	-	256	no	-	-	-	-	-

The throughput is $1 \div (1050116 \times 10ns) = 95.22$. Compared to the throughput of the last question, 95.22, it doesn't change. It is reasonable since the difference between them is where to output the new array.

How does the resource usage change?

Lookup table

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	115	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	330	-
Register	-	-	518	-	-
Total	6	16	1500	2509	0
Available	280	220	106400	53200	0
Utilization (%)	2	7	1	4	0

IOArray

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	115	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	302	-
Register	-	-	518	-	-
Total	6	16	1500	2481	0
Available	280	220	106400	53200	0
Utilization (%)	2	7	1	4	0

From the above tables, we can see that the LUT of multiplexer becomes smaller. Take a look at their tables.

* Multiplexer:

Name	LUT	Input Size	Bits	Total Bits
add14_i_reg_235	9	2	32	64
add3723_i_reg_248	9	2	32	64
ap_NS_fsm	113	23	1	23
grp_fu_272_opcode	14	3	2	6
grp_fu_272_p0	14	3	32	96
grp_fu_272_p1	14	3	32	96
grp_fu_276_p0	14	3	32	96
grp_fu_276_p1	14	3	32	96
i_1_fu_78	9	2	9	18
i_fu_54	9	2	9	18
imag_in_address0	14	3	8	24
imag_out_address0	14	3	8	24
imag_sample_address0	14	3	8	24
k_fu_74	9	2	9	18
n_reg_224	9	2	9	18
phi_mul_reg_261	9	2	8	16
real_in_address0	14	3	8	24
real_out_address0	14	3	8	24
real_sample_address0	14	3	8	24
Total	330	70	287	773

* Multiplexer:

Name	LUT	Input Size	Bits	Total Bits
add14_i_reg_249	9	2	32	64
add3723_i_reg_262	9	2	32	64
ap_NS_fsm	113	23	1	23
grp_fu_286_opcode	14	3	2	6
grp_fu_286_p0	14	3	32	96
grp_fu_286_p1	14	3	32	96
grp_fu_290_p0	14	3	32	96
grp_fu_290_p1	14	3	32	96
i_1_fu_82	9	2	9	18
i_fu_58	9	2	9	18
imag_in_address0	14	3	8	24
imag_out_address0	14	3	8	24
k_fu_78	9	2	9	18
n_reg_238	9	2	9	18
phi_mul_reg_275	9	2	8	16
real_in_address0	14	3	8	24
real_out_address0	14	3	8	24
Total	302	64	271	725

* Register:

We can see that real_sample_address0 and imag_sample_address0 are gone. I think that it is because we write output to another array in this case, we don't need to access the address based on the read/write state.

3. Loop optimization

```
DFT_OUTER_LOOP:
    for (n = 0; n < SIZE; n++) {
DFT_INNER_LOOP:
        for (k = 0; k < SIZE; k++) {
            Real_freq[k] += real_sample[n] * cos_coefficients_table[n*k%SIZE] - imag_sample[n] * sin_coefficients_table[n*k%SIZE];
            Imag_freq[k] += imag_sample[n] * cos_coefficients_table[n*k%SIZE] + real_sample[n] * sin_coefficients_table[n*k%SIZE];
        }
    }
```

To apply loop unroll and array partition, I exchange the outer loop and inner loop so that the inner loop can write to different element if unrolled.

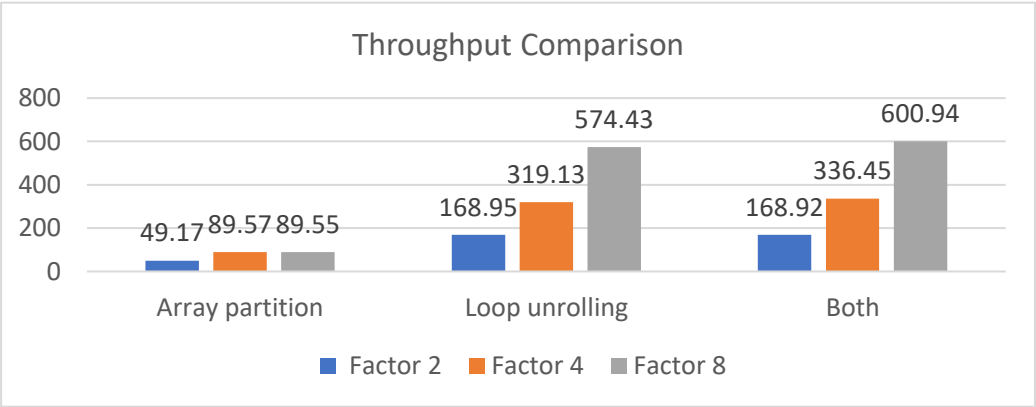
```
read
    READ_LOOP
dft_compute
    DFT_INIT_LOOP
    DFT_OUTER_LOOP
        DFT_INNER_LOOP
            HLS UNROLL factor=8
write
    WRITE_LOOP
dft
    HLS PIPELINE off
    HLS TOP name=dft
    real_sample
    imag_sample
    Real_freq
    Imag_freq
    real_in
    imag_in
    real_out
    HLS ARRAY_PARTITION dim=1 factor=8 cyclic variable=real_out
    imag_out
    HLS ARRAY_PARTITION dim=1 factor=8 cyclic variable=imag_out
```

What is the relationship between array partitioning and loop unrolling?

When we use loop unrolling, we have a chance to access a block of data that can only be accessed one by one. If we make use of array partitioning, we can access those data at the same time, and increase the throughput.

Plot the performance in terms of number of DFT operations per second (throughput) versus the unroll and array partitioning factor. Plot the same trend for resources (showing LUTs, FFs, DSP blocks, BRAMs).

Throughput



Resource usage

Array partitioning

<table>
<tr><th>factor</th><th>resource</th></tr>
<tr><td>2</td><td>
<table>
<tr><th>Name</th><th>BRAM_18K</th><th>DSP</th><th>FF</th><th>LUT</th><th>URAM</th></tr>
<tr><td>DSP</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Expression</td><td>-</td><td>-</td><td>0</td><td>140</td><td>-</td></tr>
<tr><td>FIFO</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Instance</td><td>-</td><td>8</td><td>491</td><td>1068</td><td>-</td></tr>
<tr><td>Memory</td><td>8</td><td>-</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Multiplexer</td><td>-</td><td>-</td><td>-</td><td>485</td><td>-</td></tr>
<tr><td>Register</td><td>-</td><td>-</td><td>541</td><td>-</td><td>-</td></tr>
<tr><td>Total</td><td>8</td><td>8</td><td>1032</td><td>1693</td><td>0</td></tr>
<tr><td>Available</td><td>280</td><td>220</td><td>106400</td><td>53200</td><td>0</td></tr>
<tr><td>Utilization (%)</td><td>2</td><td>3</td><td>~0</td><td>3</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>4</td><td>
<table>
<tr><th>Name</th><th>BRAM_18K</th><th>DSP</th><th>FF</th><th>LUT</th><th>URAM</th></tr>
<tr><td>DSP</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Expression</td><td>-</td><td>-</td><td>0</td><td>140</td><td>-</td></tr>
<tr><td>FIFO</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Instance</td><td>-</td><td>16</td><td>982</td><td>2144</td><td>-</td></tr>
<tr><td>Memory</td><td>12</td><td>-</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Multiplexer</td><td>-</td><td>-</td><td>-</td><td>585</td><td>-</td></tr>
<tr><td>Register</td><td>-</td><td>-</td><td>637</td><td>-</td><td>-</td></tr>
<tr><td>Total</td><td>12</td><td>16</td><td>1619</td><td>2869</td><td>0</td></tr>
<tr><td>Available</td><td>280</td><td>220</td><td>106400</td><td>53200</td><td>0</td></tr>
<tr><td>Utilization (%)</td><td>4</td><td>7</td><td>1</td><td>5</td><td>0</td></tr>
</table>
</td></tr>
<tr><td>8</td><td>
<table>
<tr><th>Name</th><th>BRAM_18K</th><th>DSP</th><th>FF</th><th>LUT</th><th>URAM</th></tr>
<tr><td>DSP</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Expression</td><td>-</td><td>-</td><td>0</td><td>140</td><td>-</td></tr>
<tr><td>FIFO</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Instance</td><td>-</td><td>16</td><td>982</td><td>2232</td><td>-</td></tr>
<tr><td>Memory</td><td>20</td><td>-</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Multiplexer</td><td>-</td><td>-</td><td>-</td><td>901</td><td>-</td></tr>
<tr><td>Register</td><td>-</td><td>-</td><td>680</td><td>-</td><td>-</td></tr>
<tr><td>Total</td><td>20</td><td>16</td><td>1662</td><td>3273</td><td>0</td></tr>
<tr><td>Available</td><td>280</td><td>220</td><td>106400</td><td>53200</td><td>0</td></tr>
<tr><td>Utilization (%)</td><td>7</td><td>7</td><td>1</td><td>6</td><td>0</td></tr>
</table>
</td></tr>
</table>

Loop unrolling

<table>
<tr><th>factor</th><th>resource</th></tr>
<tr><td>2</td><td>
<table>
<tr><th>Name</th><th>BRAM_18K</th><th>DSP</th><th>FF</th><th>LUT</th><th>URAM</th></tr>
<tr><td>DSP</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Expression</td><td>-</td><td>-</td><td>0</td><td>137</td><td>-</td></tr>
<tr><td>FIFO</td><td>-</td><td>-</td><td>-</td><td>-</td><td>-</td></tr>
<tr><td>Instance</td><td>-</td><td>32</td><td>1964</td><td>4169</td><td>-</td></tr>
<tr><td>Memory</td><td>8</td><td>-</td><td>0</td><td>0</td><td>0</td></tr>
<tr><td>Multiplexer</td><td>-</td><td>-</td><td>-</td><td>442</td><td>-</td></tr>
<tr><td>Register</td><td>-</td><td>-</td><td>865</td><td>-</td><td>-</td></tr>
<tr><td>Total</td><td>8</td><td>32</td><td>2829</td><td>4748</td><td>0</td></tr>
<tr><td>Available</td><td>280</td><td>220</td><td>106400</td><td>53200</td><td>0</td></tr>
<tr><td>Utilization (%)</td><td>2</td><td>14</td><td>2</td><td>8</td><td>0</td></tr>
</table>
</td></tr>
</table>

4

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Instance	2	32	3464	4978	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	221	-
Register	-	-	43	-	-
Total	8	32	3507	5249	0
Available	280	220	106400	53200	0
Utilization (%)	2	14	3	9	0

8

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Instance	2	32	4810	5480	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	221	-
Register	-	-	43	-	-
Total	8	32	4853	5751	0
Available	280	220	106400	53200	0
Utilization (%)	2	14	4	10	0

Array partitioning and loop unrolling

factor | resource

2

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	129	-
FIFO	-	-	-	-	-
Instance	-	32	1964	4187	-
Memory	8	-	0	0	0
Multiplexer	-	-	-	494	-
Register	-	-	928	-	-
Total	8	32	2892	4810	0
Available	280	220	106400	53200	0
Utilization (%)	2	14	2	9	0

4

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Instance	12	64	5417	9138	-
Memory	10	-	0	0	0
Multiplexer	-	-	-	412	-
Register	-	-	110	-	-
Total	22	64	5527	9600	0
Available	280	220	106400	53200	0
Utilization (%)	7	29	5	18	0

8

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	50	-
FIFO	-	-	-	-	-
Instance	24	112	9718	16036	-
Memory	18	-	0	0	0
Multiplexer	-	-	-	708	-
Register	-	-	111	-	-
Total	42	112	9829	16794	0
Available	280	220	106400	53200	0
Utilization (%)	15	50	9	31	0

First, I want to discuss the result of array partitioning. First, let's take a look at the performance tables.

factor	performance table
No	

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	1116164	1.116E7	- 1
READ_LOOP				-	512	5.120E3	2
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	1114880	1.115E7	4355
WRITE_LOOP				-	512	5.120E3	2

2

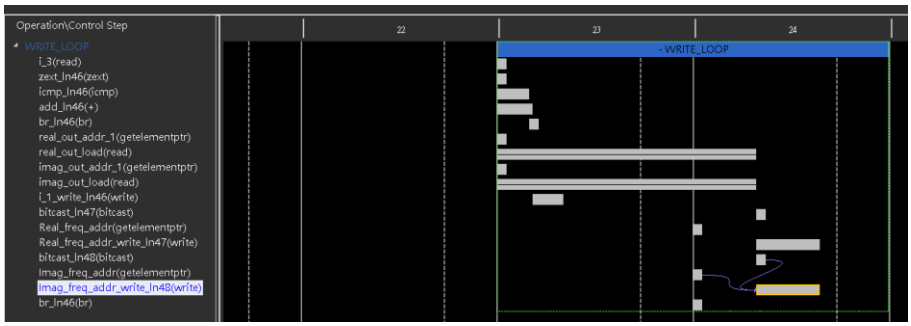
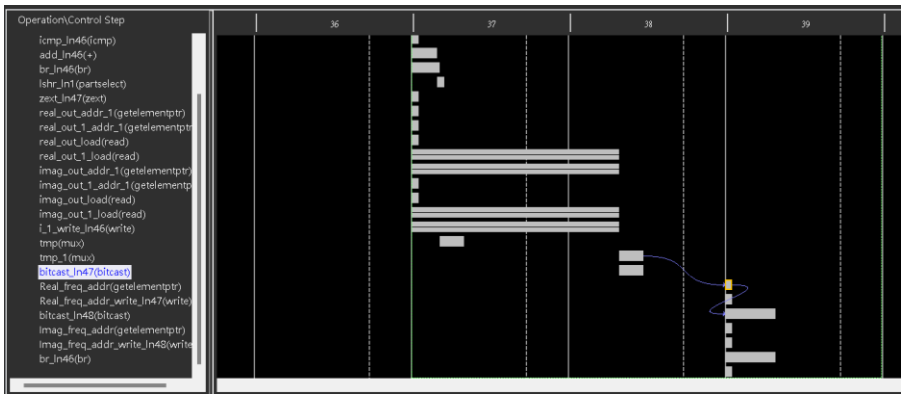
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	2033924	2.034E7	- 2
READ_LOOP				-	512	5.120E3	2
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	2032384	2.032E7	7939
WRITE_LOOP				-	768	7.680E3	3

4

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	1116420	1.116E7	- 1
READ_LOOP				-	512	5.120E3	2
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	1114880	1.115E7	4355
WRITE_LOOP				-	768	7.680E3	3

There are two things I want to discuss.

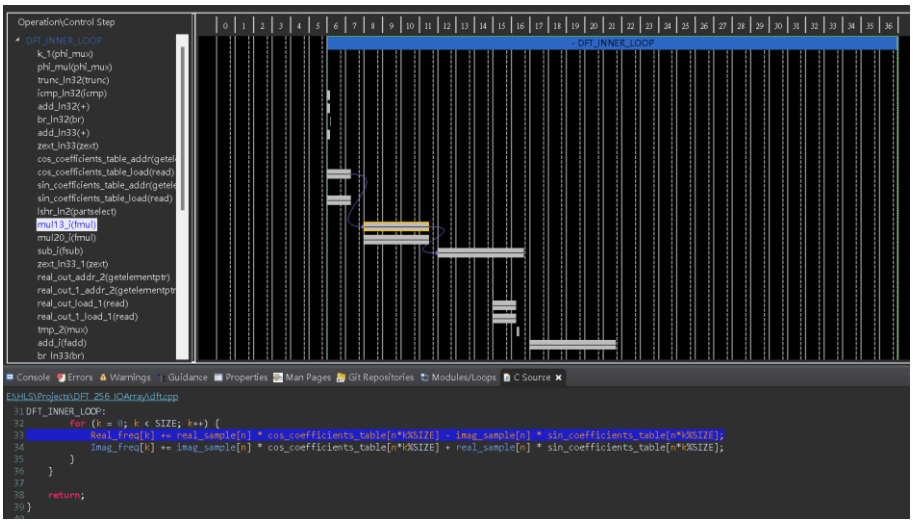
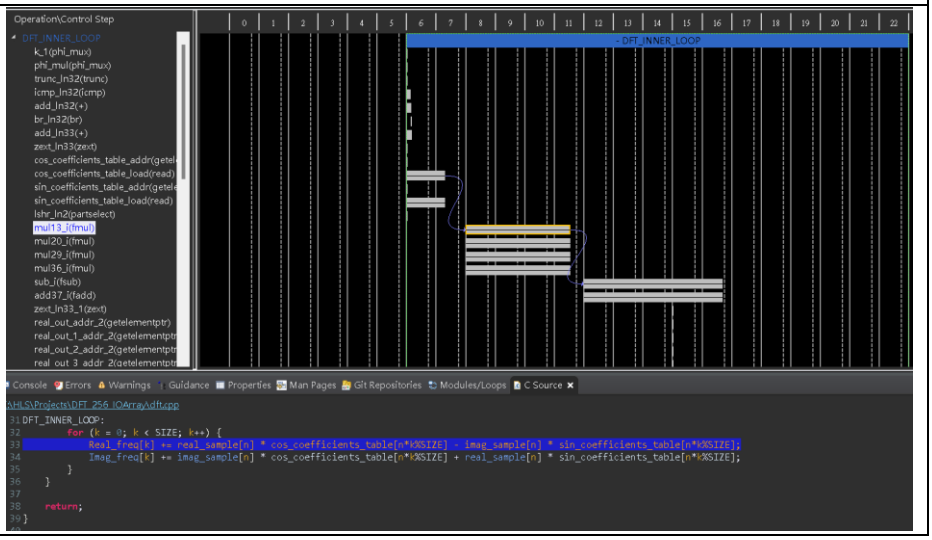
The first thing is that the iteration latency of WRITE_LOOP. We can see that after array partitioning is set, the iteration increases from 2 to 3.

factor	result
No	
2	

From the schedule viewer, we can see that when there is no array partitioning, write is executed as soon as the output data is prepared. However, in the result of factor 2, there is a MUX such that the write operation cannot be done in the

same cycle. Therefore, 3 cycles are required.

The second thing is that the latency of factor 2 is twice larger than that of factor 4.

factor	schedule viewer
2	
4	

From the schedule viewers, we can see that in the inner loop, the result of factor 2 calculate the multiplication of Real_freq and Imag_freq sequentially, and the result of factor 4 calculate at the same time. Therefore, the latency of the result of factor 2 is almost twice larger than that of factor 4. However, I have no idea why it operates in this way.

Now, I want to discuss the performance of “loop unrolling” and “array partitioning and loop unrolling”. We can see that the “loop unrolling” result of factor 2 is a bit better than the “array partitioning and loop unrolling” result. However, it is not the case in factor 4 and 8. Let’s take a look at the performance tables of factor 2 and 4.

Factor 2:

loop unrolling

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	591876	5.919E6	-
READ_LOOP				-	512	5.120E3	2
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	590592	5.906E6	2307
WRITE_LOOP				-	512	5.120E3	2

array partition and loop unrolling

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	592132	5.921E6	-
READ_LOOP				-	512	5.120E3	2
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	590592	5.906E6	2307
WRITE_LOOP				-	768	7.680E3	3

From this table, we can see that the only difference is the WRITE_LOOP. The latency of DFT_OUTER_LOOP is the same. I guess it is because the memory is 2-port, the data can be written at the same time.

Factor 4:

loop unrolling

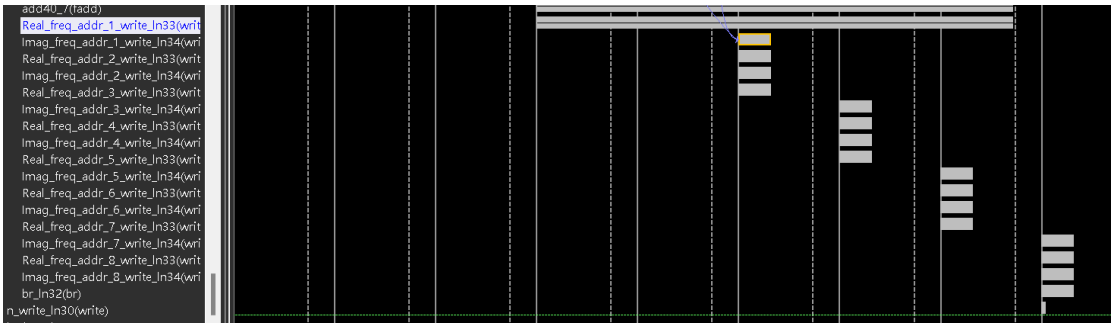
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	313349	3.133E6	-
dft_compute				-	312322	3.123E6	-
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	312064	3.121E6	1219
DFT_INNER_LOOP				-	1216	1.216E4	19
READ_LOOP				-	512	5.120E3	2
WRITE_LOOP				-	512	5.120E3	2

array partition and loop unrolling

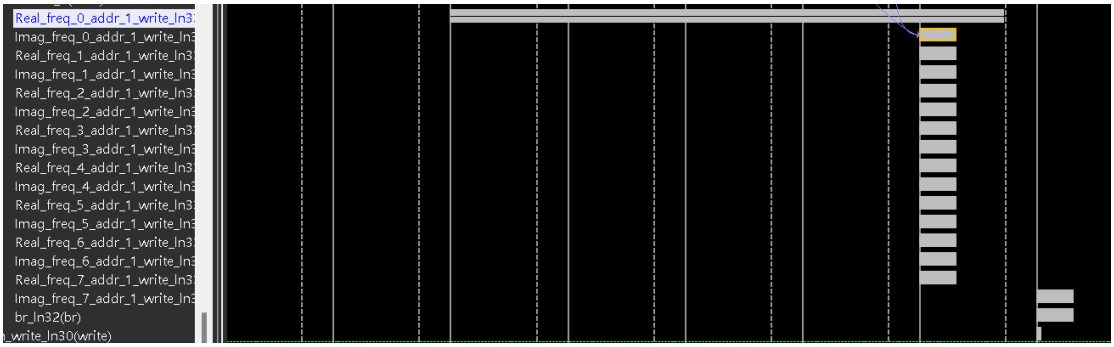
Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency
dft				-	297221	2.972E6	-
dft_compute				-	295938	2.959E6	-
DFT_INIT_LOOP				-	256	2.560E3	1
DFT_OUTER_LOOP				-	295680	2.957E6	1155
DFT_INNER_LOOP				-	1152	1.152E4	18
READ_LOOP				-	512	5.120E3	2
WRITE_LOOP				-	768	7.680E3	3

From this table, we can see that the decrease in dft_compute is larger than the increase in WRITE_LOOP. Therefore, the overall latency is decreased. To know where the improvement comes from, let’s take a look at the schedule viewer of factor 8.

The schedule viewer of loop unrolling.



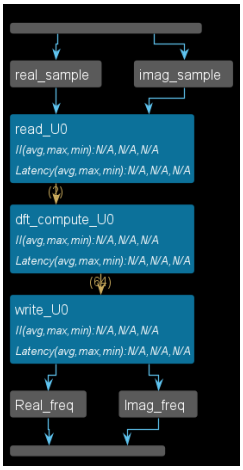
The schedule viewer of array partitioning and loop unrolling.



We can see that write operation with array partitioning is done in parallel, so the latency is reduced.

The general trend is that if we use larger factor, the throughput will increase, and the resource usage will increase as well.

4. Dataflow



I apply the dataflow pragma to my top function. The top function includes read, dft_compute, and write functions. I encountered a problem when doing co-simulation on the synthesized result without pipeline, so I enable pipeline at first. Now, let's compare the performance with and without dataflow.

without dataflow	

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval
dft				-	8771	8.771E4	-	8772
dft_Pipeline_READ_LOOP				-	258	2.580E3	-	258
dft_compute				-	8249	8.249E4	-	8249
dft_Pipeline_WRITE_LOOP				-	259	2.590E3	-	259

with dataflow	

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval
dft				-	8508	8.508E4	-	8250
read_r				-	258	2.580E3	-	258
dft_compute				-	8249	8.249E4	-	8249
write_r				-	259	2.590E3	-	259

We can see that although the latency of 3 submodules is the same, the total latency is different. Besides, the interval is smaller.

Throughput	without dataflow	11399.91
	with dataflow	12121.21

Let's take a look at the resource utilization.

without dataflow

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	-	-	-
FIFO	-	-	-	-	-
Instance	28	160	15882	25011	-
Memory	34	-	0	0	0
Multiplexer	-	-	-	847	-
Register	-	-	9	-	-
Total	62	160	15891	25858	0
Available	280	220	106400	53200	0
Utilization (%)	22	72	14	48	0

with dataflow

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	82	-
FIFO	-	-	-	-	-
Instance	28	160	15883	25030	-
Memory	66	-	0	0	0
Multiplexer	-	-	-	162	-
Register	-	-	18	-	-
Total	94	160	15901	25274	0
Available	280	220	106400	53200	0
Utilization (%)	33	72	14	47	0

We can see that with dataflow, we will need more BRAM.

I also have tried to change my code into more submodules, but end up less efficient.

5. Best architecture

   The best architecture of mine is the one with smallest interval, that is the result with dataflow in the previous question. The optimization I used is shown in the figure below.

```
read
  READ_LOOP
dft_compute
  DFT_INIT_LOOP
    HLS UNROLL factor=8
  DFT_OUTER_LOOP
    DFT_INNER_LOOP
      HLS UNROLL factor=8
write
  WRITE_LOOP
dft
  HLS TOP name=dft
  HLS DATAFLOW
  real_sample
  imag_sample
  Real_freq
  Imag_freq
  real_in
  imag_in
  real_out
  HLS ARRAY_PARTITION variable=real_out cyclic factor=8 dim=1
  imag_out
  HLS ARRAY_PARTITION variable=imag_out cyclic factor=8 dim=1
```

The tradeoffs are the resource utilization. Let's compare the performance table and resource utilization of the original architecture (lookup table) and the last architecture.

original

Modules & Loops	Issue Type	Slack	Latency (cycles)	Latency (ns)	Iteration Latency	Interval
+ dft	-	0.04	1050115	1.050e+07	-	1050116
o READ_LOOP	-	7.30	512	5.120e+03	2	-
o DFT_OUTER_LOOP	-	7.30	1049088	1.049e+07	4098	-
o DFT_INNER_LOOP	-	7.30	4096	4.096e+04	16	-
o WRITE_LOOP	-	7.30	512	5.120e+03	2	-

last

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval
dft				-	8508	8.508E4	-	8250
read_r				-	258	2.580E3	-	258
dft_compute				-	8249	8.249E4	-	8249
write_r				-	259	2.590E3	-	259

Throughput

original	$1 \div 10501160ns = 95.23$
last	$1 \div 82500ns = 12121.21$

original

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	115	-
FIFO	-	-	-	-	-
Instance	-	16	982	2064	-
Memory	6	-	0	0	0
Multiplexer	-	-	-	330	-
Register	-	-	518	-	-
Total	6	16	1500	2509	0
Available	280	220	106400	53200	0
Utilization (%)	2	7	1	4	0

last

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	82	-
FIFO	-	-	-	-	-
Instance	28	160	15883	25030	-
Memory	66	-	0	0	0
Multiplexer	-	-	-	162	-
Register	-	-	18	-	-
Total	94	160	15901	25274	0
Available	280	220	106400	53200	0
Utilization (%)	33	72	14	47	0

We can see that the throughput increases a lot, but the resource utilization also increases about 10 times of the original usage.

6. Streaming interface synthesis

Describe the major changes that you made to your code to implement the streaming interface. What benefits does the streaming interface provide? What are the drawbacks?

```
void dft(hls::stream<DTYPE> &real_sample, hls::stream<DTYPE> &imag_sample, hls::stream<DTYPE> &Real_freq, hls::stream<DTYPE> &Imag_freq)
{
    //Write your code here
    int k = 0;
    int n = 0;
    DTYPE Real[SIZE];
    DTYPE Imag[SIZE];
    DTYPE real[SIZE];
    DTYPE imag[SIZE];

DFT_INIT_LOOP:
    for (k = 0; k < SIZE; k++) {
        Real[k] = 0;
        Imag[k] = 0;
        real[k] = real_sample.read();
        imag[k] = imag_sample.read();
    }

DFT_OUTER_LOOP:
    for (n = 0; n < SIZE; n++) {
DFT_INNER_LOOP:
        for (k = 0; k < SIZE; k++) {
            Real[k] += real[n] * cos_coefficients_table[n*k%SIZE] - imag[n] * sin_coefficients_table[n*k%SIZE];
            Imag[k] += imag[n] * cos_coefficients_table[n*k%SIZE] + real[n] * sin_coefficients_table[n*k%SIZE];
        }
    }

DFT_OUTPUT_LOOP:
    for (k = 0; k < SIZE; k++) {
        Real_freq << Real[k];
        Imag_freq << Imag[k];
    }

    return;
}
```

I use 4 arrays to store the input streams and output streams. We must finish calculating a value before writing it into stream, so I use arrays to store the temporary value. real and imag must be loaded first. Otherwise, they need to be read in the DFT_OUTER_LOOP, and we cannot perform pipeline to lower the latency. Does it benefit? Let's take a look at the performance table.

without stream

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval
dft				-	8508	8.508E4	-	8250
read_r				-	258	2.580E3	-	258
dft_compute				-	8249	8.249E4	-	8249
write_r				-	259	2.590E3	-	259

with stream

Modules & Loops	Issue Type	Violation Type	Distance	Slack	Latency(cycles)	Latency(ns)	Iteration Latency	Interval
dft				-	8474	8.474E4	-	8475
Loop_DFT_INIT_LOOP_proc1				-	8474	8.474E4	-	8474
Loop_DFT_OUTPUT_LOOP_proc2				-	259	2.590E3	-	259

We can see that the total latency decreases. However, the interval increases. Since we can't start a new computation when output the values.

without stream	with stream

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	82	-
FIFO	-	-	-	-	-
Instance	28	160	15883	25030	-
Memory	66	-	0	0	0
Multiplexer	-	-	-	162	-
Register	-	-	18	-	-
Total	94	160	15901	25274	0
Available	280	220	106400	53200	0
Utilization (%)	33	72	14	47	0

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	70	-
FIFO	-	-	-	-	-
Instance	30	160	15784	25118	0
Memory	64	-	0	0	0
Multiplexer	-	-	-	144	-
Register	-	-	16	-	-
Total	94	160	15800	25332	0
Available	280	220	106400	53200	0
Utilization (%)	33	72	14	47	0

We can see that it uses less FF but more LUT.

The figure below is the co-simulation report

Cosimulation Report for 'dft'

General Information

Date: Sat Apr 1 11:13:06 2023
Version: 2022.1 (Build 3526262 on Mon Apr 18 15:48:16 MDT 2022)
Project: DFT_256_Stream
Status: Pass

Cosim Options

Tool: Vivado XSIM
RTL: Verilog
Dump Trace: all
Channel (PIP

Performance Estimates

Modules & Loops	Avg II	Max II	Min II	Avg Latency	Max Latency	Min Latency
dft				8728	8728	8728
Loop_DFT_INIT_LOOP_proc1				8470	8470	8470
Loop_DFT_OUTPUT_LOOP_proc2				257	257	257

Explain what you observed and learned

In this lab, I learned to optimize a design in several ways, including lookup table, array partition, unroll, pipeline, and dataflow. I also learned how to transmit float data through streams.

Explain what problem you encountered and how you solved it

PYNQ Demo failed.

In the PYNQ Demo section, we are required to implement the result of streaming interface in the last question, and we need to change the streaming data type and the problem size increases from 256 to 1024. The reason of changing data type is that to send a stream of elements, we need to use specific structures for the elements. The structures are shown in the figure below.

```
template<int D,int U,int TI,int TD>
struct ap_axis {
    ap_int<D>    data;
    ap_uint<D/8> keep;
    ap_uint<D/8> strb;
    ap_uint<U>   user;
    ap_uint<1>   last;
    ap_uint<TI>  id;
    ap_uint<TD>  dest;
};

template<int D,int U,int TI,int TD>
struct ap_axiu {
    ap_uint<D>   data;
    ap_uint<D/8> keep;
    ap_uint<D/8> strb;
    ap_uint<U>   user;
    ap_uint<1>   last;
    ap_uint<TI>  id;
    ap_uint<TD>  dest;
};
```

In this way, the stream signals can be synthesized as shown in the figure below.

AXIS

Interface	Register Mode	TDATA	TKEEP	TLAST	TREADY	TSTRB	TVALID
Imag_freq	both	32	4	1	1	4	1
Real_freq	both	32	4	1	1	4	1
imag_sample	both	32	4	1	1	4	1
real_sample	both	32	4	1	1	4	1

It contains necessary signals to use DMA. TLAST is a key signal that tells if the stream finishes sending or receiving elements.

However, in this lab, we need to transfer float data rather than integer through the stream, so the tutorial asks us to construct a structure as shown in the figure below.

```
struct DTYPE
{
    float data;
    ap_int<1> last;
};
```

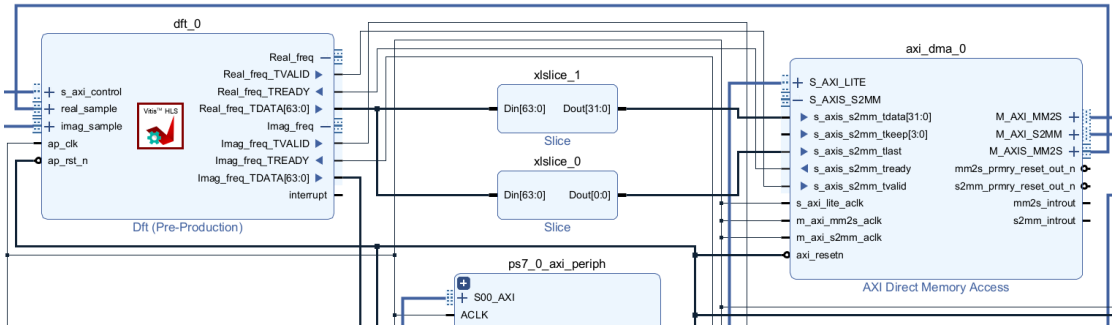
“last” of the last output element should be set to 1, others are set to 0.
After synthesis, the stream signals are shown in the figure below.

AXIS

Interface	Register Mode	TDATA	TREADY	TVALID
Imag_freq	both	64	1	1
Real_freq	both	64	1	1
imag_sample	both	64	1	1
real_sample	both	64	1	1

We can see that even though we have added “last” in the struct, TLAST is still gone. What would happen if TLAST is gone? After we export IP, add it in the block diagram of Vivado, and then create wrapper, it will raise critical warning that TLAST in the DMA is not connected. If we run the bit file on Jupyter, the stream will get stuck there, since it doesn’t know if the stream is done.

After I tried several methods and searched on the internet, I found nothing helpful. In the end, when I was writing this report, I found that the size of TDATA in the above figure becomes 64, which means that the “last” is inside it. Therefore, when I connect the block diagram in Vivado, I split the TDATA[64:0] into TDATA[32] and TDATA[31:0], corresponding to “last” and “data” respectively (variables in a structure are arranged from lower bit field to higher bit filed). Part of the block diagram is shown below.



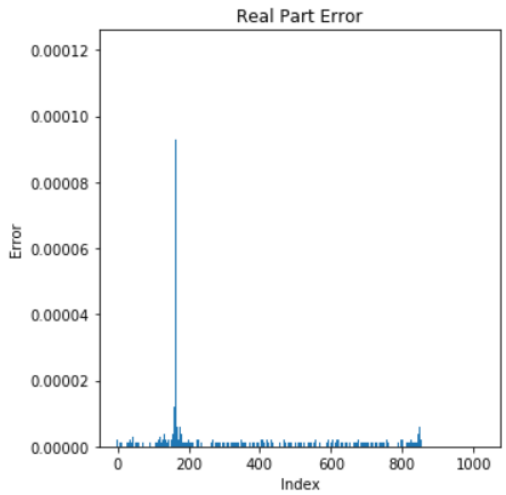
When we output the result from dft_0 to axi_dma_0, the result goes through xlclice_1 and xlslice_0, and send to tdata and tlast. Ready and valid signals are also connected by hand. In this way, we can get the correct bit file. The demo result is shown in the figures below.

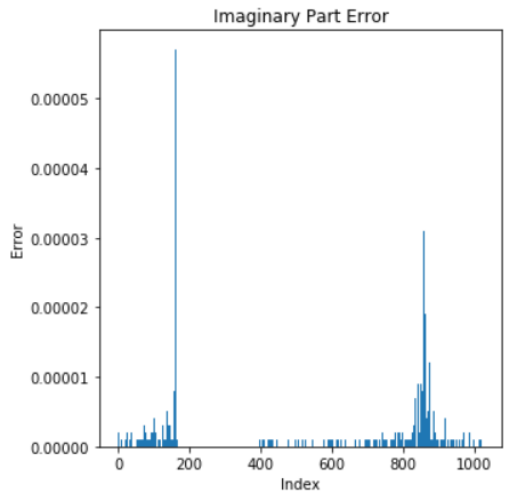
Verifying Functionality

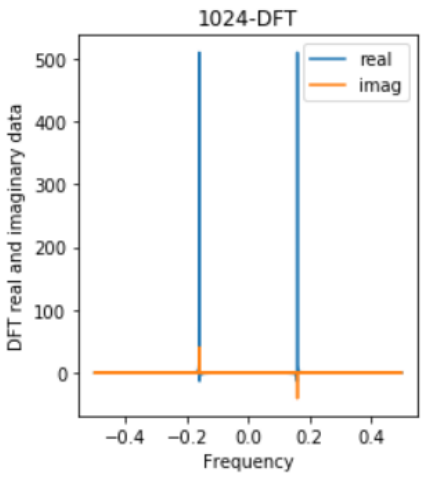
In [7]:
```
golden_op=np.fft.fft(a)

for i in range(NUM_SAMPLES):

    real_error[i]="{0:.6f}".format(abs(out_r[i]-golden_op.real[i]))
    imag_error[i]="{0:.6f}".format(abs(out_i[i]-golden_op.imag[i]))
```

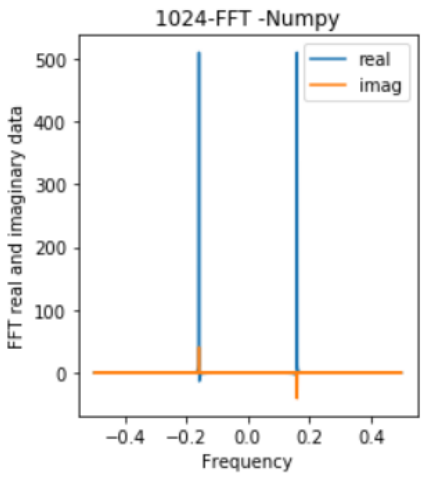
In [8]:
```
sum_sq_real=0
sum_sq_imag=0
for i in range(NUM_SAMPLES):
    sum_sq_real =sum_sq_real+(real_error[i]*real_error[i])
    real_rmse = np.sqrt(sum_sq_real / (i+1))
    sum_sq_imag =sum_sq_imag+(imag_error[i]*imag_error[i])
    imag_rmse = np.sqrt(sum_sq_imag / (i+1))
print("Real Part RMSE: ", real_rmse, "Imaginary Part RMSE:", imag_rmse)
if real_rmse<0.001 and imag_rmse<0.001:
    print("PASS")
else:
    print("FAIL")
```

```
Real Part RMSE:  1.2013298295534854e-05 Imaginary Part RMSE: 4.895988472719265e-06
PASS
```









Another problem is that the python code they provide is too old, the way to prepare IO stream has changed. The figures below show the original code and the new code.

Original code

```
from pynq import Xlnk
```

```
xlnk = Xlnk()
in_r = xlnk.cma_array(shape=(NUM_SAMPLES,), dtype=np.float32)
in_i = xlnk.cma_array(shape=(NUM_SAMPLES,), dtype=np.float32)
out_r = xlnk.cma_array(shape=(NUM_SAMPLES,), dtype=np.float32)
out_i = xlnk.cma_array(shape=(NUM_SAMPLES,), dtype=np.float32)
```

New code

```
from pynq import allocate
```

```
in_r = allocate(shape=(NUM_SAMPLES,), dtype=np.float32)
in_i = allocate(shape=(NUM_SAMPLES,), dtype=np.float32)
out_r = allocate(shape=(NUM_SAMPLES,), dtype=np.float32)
out_i = allocate(shape=(NUM_SAMPLES,), dtype=np.float32)
```