4.3 Fast Fourier transform accelerator

1. Experiment purpose

In this lesson, we mainly learn function of Fast Fourier transform accelerator of K210.

2.Experiment preparation

2.1 components

Fast Fourier transform accelerator in K210 chip

2.2 Component characteristics

K210 built-in fast Fourier transform accelerator FFT Accelerater.

The FFT accelerator uses hardware to realize the radix-2 time division operation of FFT.

- Support multiple operation lengths, 64 points, 128 points, 256 points and 512 points.
- Support two operation modes, FFT and IFFT operation.
- Support configurable input data width, 32-bit and 64-bit input.
- Support configurable input data arrangement methods, that is, support three data arrangement methods of imaginary part, real part alternate, pure real part, and real part and imaginary part separated
- Support DMA transfer

2.3 Corresponding API function in SDK

The corresponding header file aes.h

We will provide users with the following interfaces:

- fft_complex_uint16_dma: FFT operation.

3. Experiment principle

At present, the module can support 64-point, 128-point, 256-point and 512-point FFT and IFFT.

There are two blocks of 512*32bit SRAM inside the FFT.

After the configuration is completed, the FFT will send a TX request to the DMA, and put the data sent by the DMA into one of the SRAMs until it meets the requirements of the current FFT operation. Then, start FFT operation.

Butterfly operation unit reads data from the SRAM containing valid data, and writes the data to another SRAM after the operation is over. The next butterfly operation reads from the newly written SRAM Output the data and write it into another SRAM after the operation is over. Repeat this alternately until the entire FFT operation is completed.

4. Experiment procedure

4.1 Obtain a set of complex numbers through trigonometric functions.

```
int32_t i;
float tempf1[3];
fft_data_t *output_data;
fft_data_t *input_data;
uint16_t bit1_num = get_bit1_num(FFT_FORWARD_SHIFT);
complex_hard_t data_hard[FFT_N] = {0};
complex data_soft[FFT_N] = {0};
/* Obtain a set of complex numbers  */
for (i = 0; i < FFT_N; i++)
{
    tempf1[0] = 0.3 * cosf(2 * PI * i / FFT_N + PI / 3) * 256;
    tempf1[1] = 0.1 * cosf(16 * 2 * PI * i / FFT_N - PI / 9) * 256;
    tempf1[2] = 0.5 * cosf((19 * 2 * PI * i / FFT_N) + PI / 6) * 256;
    data_hard[i].real = (int16_t)(tempf1[0] + tempf1[1] + tempf1[2] + 10);
    data_hard[i].imag = (int16_t)0;
    data_soft[i].real = data_hard[i].real;
    data_soft[i].imag = data_hard[i].imag;
}
```

4.2 Convert the obtained complex number into a fast Fourier transform data structure as input data (to be calculated).

```
/* Convert complex numbers to Fourier data structure RIRI */
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
```

4.3 Using hardware and software to perform FFT operation respectively, and record the running time.

```
/* Hardware processes FFT data and records time */
cycle[FFT_HARD][FFT_DIR_FORWARD] = read_cycle();
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_FORWARD_SHIFT, FFT_DIR_FORWARD, buffer_input, FFT_N, buffer_output);
cycle[FFT_HARD][FFT_DIR_FORWARD] = read_cycle() - cycle[FFT_HARD][FFT_DIR_FORWARD];

/* Software processes FFT data and records time */
cycle[FFT_SOFT][FFT_DIR_FORWARD] = read_cycle();
fft_soft(data_soft, FFT_N);
cycle[FFT_SOFT][FFT_DIR_FORWARD] = read_cycle() - cycle[FFT_SOFT][FFT_DIR_FORWARD];
```

4.4 Take the module of the output data.

```
for (i = 0; i < FFT_N; i++)
{
    hard_power[i] = sqrt(data_hard[i].real * data_hard[i].real + data_hard[i].imag * data_hard[i].imag) * 2;
    soft_power[i] = sqrt(data_soft[i].real * data_soft[i].real + data_soft[i].imag * data_soft[i].imag) * 2;
}
```

4.5 Print data of real and imaginary parts of complex Numbers, as well as information such as

module and phases.

```
/* Print the real and imaginary parts of software and hardware complex Numbers */
printf("\n[hard fft real][soft fft real][hard fft imag][soft fft imag]\n");
for (i = 0; i < FFT_N / 2; i++)
    printf("%3d:%7d %7d %7d %7d\n",
           i, data_hard[i].real, (int32_t)data_soft[i].real, data_hard[i].imag, (int32_t)data_soft[i].imag);

printf("\nhard power  soft power:\n");
printf("%3d : %f  %f\n", 0, hard_power[0] / 2 / FFT_N * (1 << bit1_num), soft_power[0] / 2 / FFT_N);
for (i = 1; i < FFT_N / 2; i++)
    printf("%3d : %f  %f\n", i, hard_power[i] / FFT_N * (1 << bit1_num), soft_power[i] / FFT_N);

/* Print phase */
printf("\nhard phase  soft phase:\n");
for (i = 0; i < FFT_N / 2; i++)
{
    hard_angel[i] = atan2(data_hard[i].imag, data_hard[i].real);
    soft_angel[i] = atan2(data_soft[i].imag, data_soft[i].real);
    printf("%3d : %f  %f\n", i, hard_angel[i] * 180 / PI, soft_angel[i] * 180 / PI);
}
```

4.6 inverse operation of the FFT.

```
/*FFT inverse operation */
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}

/* Hardware and software FFT operations */
cycle[FFT_HARD][FFT_DIR_BACKWARD] = read_cycle();
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_BACKWARD_SHIFT, FFT_DIR_BACKWARD, buffer_input, FFT_N, buffer_output);
cycle[FFT_HARD][FFT_DIR_BACKWARD] = read_cycle() - cycle[FFT_HARD][FFT_DIR_BACKWARD];
cycle[FFT_SOFT][FFT_DIR_BACKWARD] = read_cycle();
ifft_soft(data_soft, FFT_N);
cycle[FFT_SOFT][FFT_DIR_BACKWARD] = read_cycle() - cycle[FFT_SOFT][FFT_DIR_BACKWARD];
```

4.7 Print the output data of the inverse inverse of FFT.

```
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
}

printf("\n[hard ifft real][soft ifft real][hard ifft imag][soft ifft imag]\n");
for (i = 0; i < FFT_N / 2; i++)
    printf("%3d:%7d  %7d %7d %7d\n",
           i, data_hard[i].real, (int32_t)data_soft[i].real, data_hard[i].imag, (int32_t)data_soft[i].imag);
```

4.8 Printing hardware and software took time to compute the Fourier transform and the inverse as a comparison.

```
printf("[hard fft test] [%d bytes] forward time = %ld us, backward time = %ld us\n",
        FFT_N,
        cycle[FFT_HARD][FFT_DIR_FORWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000),
        cycle[FFT_HARD][FFT_DIR_BACKWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000));

printf("[soft fft test] [%d bytes] forward time = %ld us, backward time = %ld us\n",
        FFT_N,
        cycle[FFT_SOFT][FFT_DIR_FORWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000),
        cycle[FFT_SOFT][FFT_DIR_BACKWARD]/(sysctl_clock_get_freq(SYSCTL_CLOCK_CPU)/1000000));
while (1)
    ;
```

9. Compile and debug, burn and run

Copy the fft to the src directory in the SDK.

Then, enter the build directory and run the following command to compile.

cmake .. -DPROJ=fft -G "MinGW Makefiles"

make

```
Generating .bin file ...
[100%] Built target fft
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build>
```

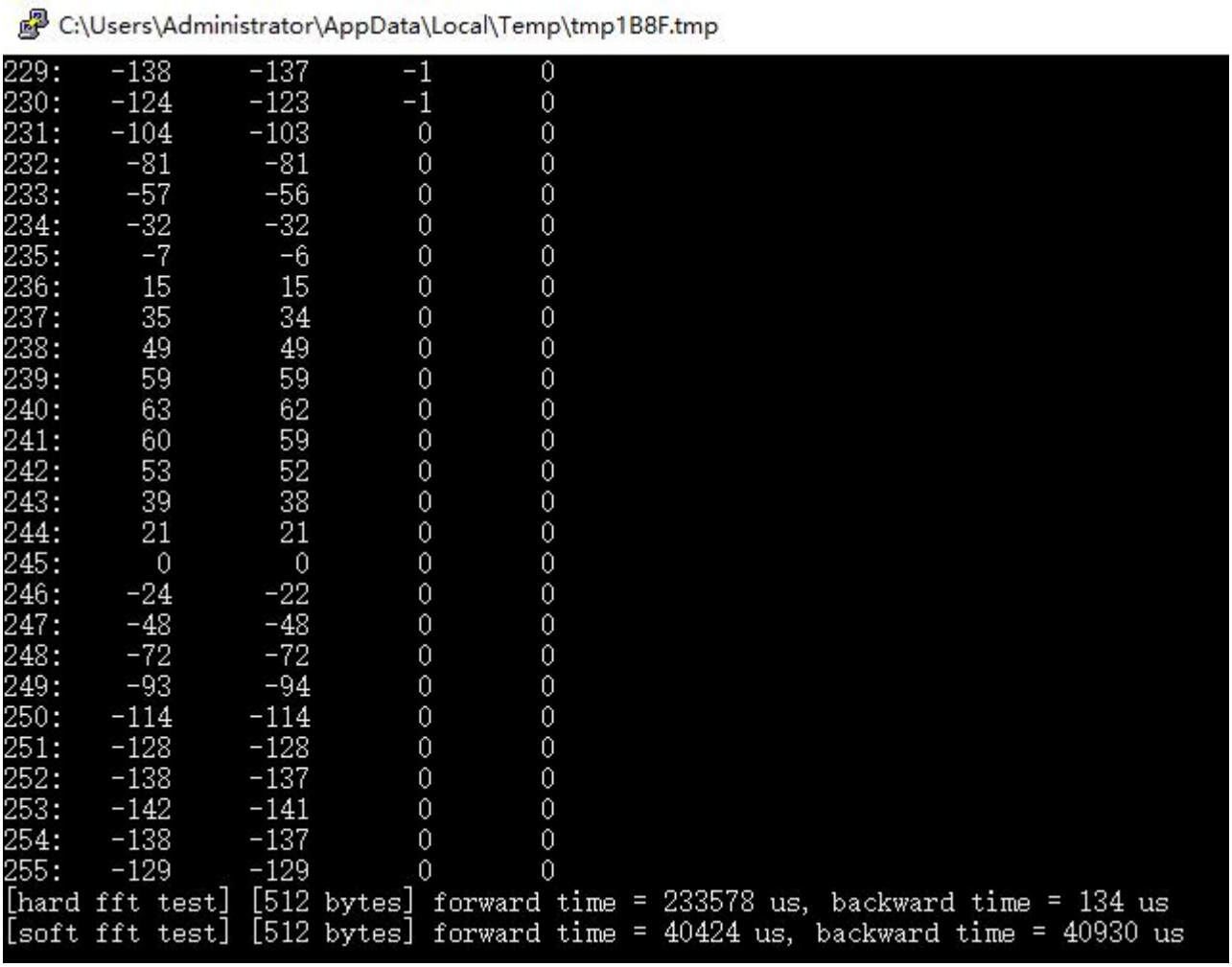
After the compilation is complete, the **fft.bin** file will be generated in the build folder.

We need to use the type-C data cable to connect the computer and the K210 development board. Open kflash, select the corresponding device, and then burn the **fft.bin** file to the K210 development board.

5. Experimental phenomenon

After the firmware is write, a terminal interface will pop up. It will print data of FFT and the inverse operation, as well as the time consumed by both operations,

As shown below.



6. Experiment summary

6.1 FFT calculation can also be realized by using CPU alone.

6.2 Software and hardware FFT calculation time will be quite different.

Appendix -- API

Header file is fft.h

fft_complex_uint16_dma

Description: FFT calculation.

Function prototype:

void fft_complex_uint16_dma(dmac_channel_number_t dma_send_channel_num, dmac_channel_number_t dma_receive_channel_num, uint16_t shift, fft_direction_t direction, const uint64_t *input, size_t point_num, uint64_t *output);

Parameter:

Parameter name	Description	Input/Output
dma_send_channel_num	DMA channel number used to send data	Input
dma_receive_channel_num	DMA channel number used to receive data	Input
shift	The 16-bit register data range (-32768~32767). The FFT transformation has 9 layers. “shift” determines which layer needs to be shifted (for example, 0x1ff means that all 9 layers do shift operations; 0x03 means that the first layer and the The second layer does shift operation) to prevent overflow. If it is shifted, the transformed amplitude is not the amplitude of the normal FFT transformation.	Input
direction	FFT operation or inverse operation	Input
input	The input data sequence, the format is RIRI.., the accuracy of the real part and the imaginary part are both 16bit	Input
point_num	The number of data points waiting to be calculated can only be 512/256/128/64	Input
output	The number of data points waiting to be calculated can only be 512/256/128/648/64	Output

Return value: no

Eg:

```
#define FFT_N                       512U
#define FFT_FORWARD_SHIFT     0x0U
#define FFT_BACKWARD_SHIFT   0x1ffU
#define PI                          3.14159265358979323846
complex_hard_t data_hard[FFT_N] = {0};
for (i = 0; i < FFT_N; i++)
{
    tempf1[0] = 0.3 * cosf(2 * PI * i / FFT_N + PI / 3) * 256;
    tempf1[1] = 0.1 * cosf(16 * 2 * PI * i / FFT_N - PI / 9) * 256;
    tempf1[2] = 0.5 * cosf((19 * 2 * PI * i / FFT_N) + PI / 6) * 256;
    data_hard[i].real = (int16_t)(tempf1[0] + tempf1[1] + tempf1[2] + 10);
    data_hard[i].imag = (int16_t)0;
}
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
```

```
}
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_FORWARD_SHIFT,
FFT_DIR_FORWARD, buffer_input, FFT_N, buffer_output);
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
}
for (int i = 0; i < FFT_N / 2; ++i)
{
    input_data = (fft_data_t *)&buffer_input[i];
    input_data->R1 = data_hard[2 * i].real;
    input_data->I1 = data_hard[2 * i].imag;
    input_data->R2 = data_hard[2 * i + 1].real;
    input_data->I2 = data_hard[2 * i + 1].imag;
}
fft_complex_uint16_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, FFT_BACKWARD_SHIFT,
FFT_DIR_BACKWARD, buffer_input, FFT_N, buffer_output);
for (i = 0; i < FFT_N / 2; i++)
{
    output_data = (fft_data_t*)&buffer_output[i];
    data_hard[2 * i].imag = output_data->I1 ;
    data_hard[2 * i].real = output_data->R1 ;
    data_hard[2 * i + 1].imag = output_data->I2 ;
    data_hard[2 * i + 1].real = output_data->R2 ;
}
```

Data type

The related data types and data structure are defined as follows:

fft_data_t: The format of the incoming data for the FFT operation.

fft_direction_t: FFT transform mode.

fft_data_t

Description: The format of the incoming data for the FFT operation.

Define

```
typedef struct tag_fft_data
{
    int16_t I1;
    int16_t R1;
    int16_t I2;
```

```
    int16_t R2;
} fft_data_t;
```

Member

Member name	Description
I1	Imaginary part of the first data
R1	Real part of the first data
I2	Imaginary part of the second data
R2	Real part of the second data

fft_direction_t

Description: FFT mode

Define

```
typedef enum _fft_direction
{
    FFT_DIR_BACKWARD,
    FFT_DIR_FORWARD,
    FFT_DIR_MAX,
} fft_direction_t;
```

Member

Member name	Member name
FFT_DIR_BACKWARD	FFT inverse transform
FFT_DIR_FORWARD	FFT forward transform