FPGA-Based Real-Time Video 2D FFT Accelerator

A Design Project Report

Presented to the School of Electrical and Computer Engineering of Cornell University

in Partial Fulfillment of the Requirements for the Degree of Master of Engineering, Electrical and Computer Engineering

Submitted by

Yibin Xu; Ruyi Zhou

MEng Field Advisor: Hunter Adams

Degree Date: May 2024

Abstract

Master of Engineering Program

School of Electrical and Computer Engineering

Cornell University

Design Project Report

Project Title: FPGA-Based Real-Time Video 2D FFT Accelerator

Author: Yibin Xu

Ruyi Zhou

Abstract: Advanced Semiconductor Materials Lithography Holding N.V. company (ASML) seeks a solution to compress the size of the input images for real-time alignment. To address this, we propose the implementation of two-dimensional (2D) Fast Fourier Transform (FFT) on real-time input images. However, the computationally intensive nature of the FFT algorithm can increase latency. To counter this, we aim to develop an Field Programmable Gate Array (FPGA) based accelerator, leveraging the parallel computing capabilities of FPGA for efficient execution of 2D FFT on live image data streams [1]. These streams, sourced directly from camera feeds, will be processed on a ZYNQ Z7010 FPGA, enabling rapid and efficient 2D FFT computation suitable for various applications. We found open source FFT algorithm online which is implemented in C++, we utilize the Vitis High-Level-Synthesis (HLS) tool to generate Register-Transfer-Level (RTL) code [2], which then be synthesized on the ZYNQ FPGA board. We anticipate that this FPGA-based FFT approach will significantly outperform CPU-based FFT computations, effectively aiding ASML in the alignment of the real-time input images.

Summary

Introduction

We're focusing on reducing input image sizes for real-time alignment using the FFT algorithm. FPGAs are crucial for their efficiency and configurability, enabling parallel processing. Our FPGA-based accelerator, designed for FFT computations, has significantly reduced execution times. It processes real-time input images efficiently, overcoming challenges like environment setup and interface management.

Implementation

The FFT algorithm leverages the symmetry properties of complex exponential basis functions to efficiently compute the Discrete Fourier Transform (DFT). Implemented in C++, our approach utilizes open-source code for reliability and time efficiency, acknowledging the contributions of the community. The algorithm, following the Cooley-Tukey variant, recursively decomposes the input data and computes the DFT using complex arithmetic operations. Utilizing functions like 'rader' for Radix-2 FFT optimization, the algorithm pairs elements symmetrically, crucial for the butterfly operation. After FFT computation, the main function orchestrates data processing, reading input from a file, performing FFT on rows and columns of a 2D complex array, and writing transformed data to files. Additionally, it executes inverse FFT, generating output suitable for further processing, such as image conversion in MATLAB, demonstrating the practicality and versatility of FFT in the real-time image alignment. This algorithm is finally implemented on the Zynq 7010 FPGA board.

Results

Our software implementation delivers satisfactory results, but our FPGA-based solution has encountered some bugs. Possible causes include camera layer distortion and complex HDL structures leading to data accuracy mismatches. Despite these issues, the FPGA solution provides real-time responses, with operation times corresponding to the camera refresh rate. Compared to the C++ implementation's 8365ms computation time, the FPGA accelerator achieves over 100 times speedup.

Conclusion

In conclusion, this report details our approach to developing an FPGA-based accelerator for ASML, focusing on real-time image compression using a 2D FFT algorithm. We've addressed initial challenges and achieved significant speedup compared to CPU-based systems. Despite some bugs in the FPGA solution, it still provides real-time responses, with operation times over 100 times faster than the software implementation. We anticipate surpassing CPU performance, positioning our team and ASML at the forefront of semiconductor manufacturing advancements.

Introduction

In the rapidly advancing field of modern semiconductors, ASML has emerged as a leader among chip fabrication companies. Our collaborative effort with ASML focuses on developing a solution to compress the size of input images for their photolithography machines. To achieve this, we have implemented the Fast Fourier Transform (FFT) algorithm. At this juncture, the high configurability and flexibility of Field-Programmable Gate Arrays (FPGAs) play a crucial role. FPGAs, known for their ability to be custom-programmed for specific applications, offer more efficient data processing and algorithm execution compared to CPUs. This capability grants FPGAs considerable advantages in parallel processing and real-time data processing. Consequently, we are employing FPGAs to design an accelerator specifically for the computation-intensive aspects of the FFT algorithm. We anticipate a significant reduction in execution time, and thus a substantial increase in performance, once the intensive computational part of the FFT algorithm is transferred to the FPGA platform.

To achieve our goal, we initially implement the Fast Fourier Transform (FFT) algorithm on a software platform to verify its feasibility before expanding it into a 2D FFT, enabling the processing of input images. Subsequently, we utilize High-Level Synthesis (HLS) tools to generate the Register Transfer Level (RTL) code for the ZYNQ FPGA as the second step. After conducting the post-synthesis simulation of the algorithm, we program the FPGA with the generated bitstream to construct our first prototype, resulting in a significant reduction in the FFT's execution time, as expected. The FPGA-based accelerator is configured to receive real-time input images from a camera and employs programmed hardware to perform parallel computing for the FFT algorithm on the input images. It then outputs compressed images in the frequency domain. This setup ensures efficient and timely processing, leveraging the strengths of FPGA in handling complex computational tasks in real time.

Background

Our first challenge was setting up the environment for the ZYNQ board and configuring the test bench with Vitis HLS. We chose to use HLS to generate the HDL code because the Xilinx platform offers demos with multiple IP cores, each containing various ports. Writing the HDL manually would require us to input the port assignments ourselves, which would be a significant amount of work. Therefore, we decided to use Vitis HLS to generate the HDL code, which will automatically adapt to the Xilinx platform. This was essential to verify our FFT algorithm and its conversion from C++ to RTL code. Additionally, C++ coding for HLS differs from standard C++ coding. We needed to modify the algorithm to ensure that the software code aligns optimally with the board's hardware structure to achieve maximum throughput. Moreover, it was crucial to ensure that the pragmas or directives added did not cause the design to exceed the board's hardware resource limits and to verify that the generated code was synthesizable. After successfully programming the FPGA, another challenge arose: designing test benches that could cover corner cases and pass both the functionality coverage test and the line coverage test.

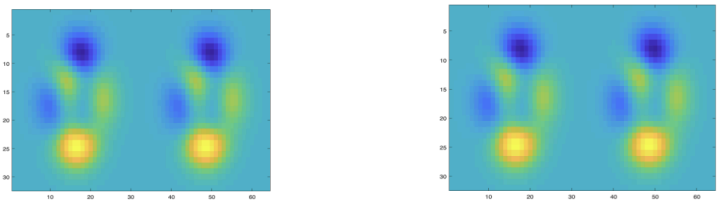
Once the FFT accelerator functions as expected, our next challenge is managing the interface between the real-time input and the FPGA board. Given that the entire system operates in real-time, it's imperative to ensure that the interface protocol is effective and provides the FPGA with accurate input data. This involves careful calibration and testing to maintain seamless data flow and system integrity in a real-time environment.

This design is complex, so we decided to adopt an incremental approach, starting with simple modules and progressively moving to more complex ones. This incremental design strategy enables us to test each unit thoroughly and advance more smoothly through the development process. By building and validating in stages, we can ensure the integrity and functionality of the design at each step, making the overall process more manageable and efficient.

Approach and Expected Results

We aim to implement a 2D FFT function on the Zybo board by utilizing Vivado HLS for translating the function from C++ into Verilog. This will enable the Zybo to process an input image and produce its FFT equivalent as output. There are a few major milestones we expected to achieve:

- FFT in C++:
    - Develop a 1D FFT algorithm in C++ and verify its accuracy against MATLAB's built-in FFT library.[4]
    - Write a 2D FFT algorithm in C++ and validate its correctness with MATLAB's built-in FFT library.[5]
    - Conduct edge case testing.
    - Transition from integer to fixed-point representation for variables.



C++ IFFT Matlab IFFT

Figure 1. The Inverse-FFT from C++ and Matlab

The figures above show the inverse-FFT results from C++ and Matlab, which indicate the functionality verification of the C++ FFT is done and as expected.

- Vivado HLS Application:
    - Create a C++-based LED blink project, convert it to Verilog using HLS, and test it on an FPGA board.

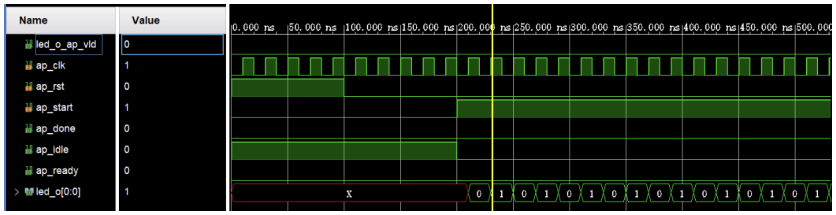


Figure 2. The waveform of the LEB blink program

The waveform above shows the post-synthesis simulation of the LED blink project, which indicates the Vitis HLS can be successfully done for a C++ program.

    - Develop an ARM-controlled C++ program and deploy it on an FPGA board.
    - Import images into the ARM program, process the image data through the program's Logic Unit, and verify the output.
- 2D FFT Implementation on Zybo Board:

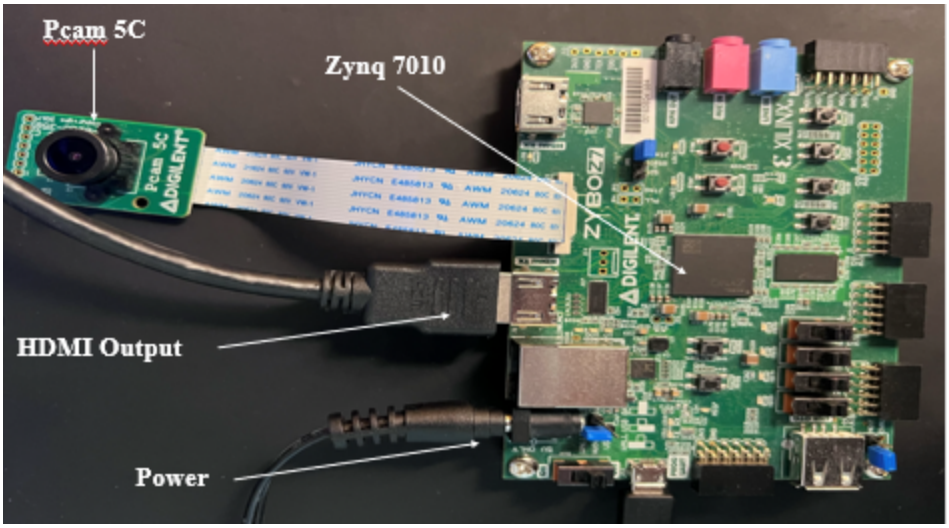


Figure 3. The Zybo Z7010 System-on-Chip board

- Convert the 2D FFT C++ code into HLS format for compatibility with the Zybo board.
- Integrate the code with the Zybo board, ensuring correct pin outputs, and test the implementation.

- Real-time Video Processing:
  - Assemble a camera module to test the FFT function's integration.
  - Transform the camera's input into a matrix format and route it to the FFT function's input.
  - Perform tests and debugging to ensure functionality.
- Testing and Debugging:
  - Assess the FFT function's overall performance and establish a performance baseline.
- HLS Acceleration for FFT:
  - Implement an accelerated FFT algorithm using HLS.
  - Compare the accelerated version's performance against the baseline to gauge efficiency improvements.

Outcome Expectation:
Anticipate that performing 2D FFT processing will reduce image size with minimal quality loss and improve processor efficiency. Aim to process 30 images per second for smooth video playback. This restructured outline divides your plan into clear stages and objectives, making it simple and systematic.

Process and Project Flow:
In general, our project design begins with the development of the FFT algorithm using a software language such as C++. We then verify its correctness using MATLAB. Once the functionality is confirmed, we utilize the Vitis HLS tool to generate the RTL code, followed by conducting post-synthesis simulation. After successful functionality verification, we implement the design on the FPGA board.

The essence of the project is that the device will accept images as input and execute the FFT algorithm. This part of the process will be carried out on the FPGA. Subsequently, the FPGA will output the frequency-domain data, enabling the client to apply filters as needed. The device also has the capability to perform the inverse FFT, converting the data back from the frequency domain to the time domain. The figures below illustrate the process flow.

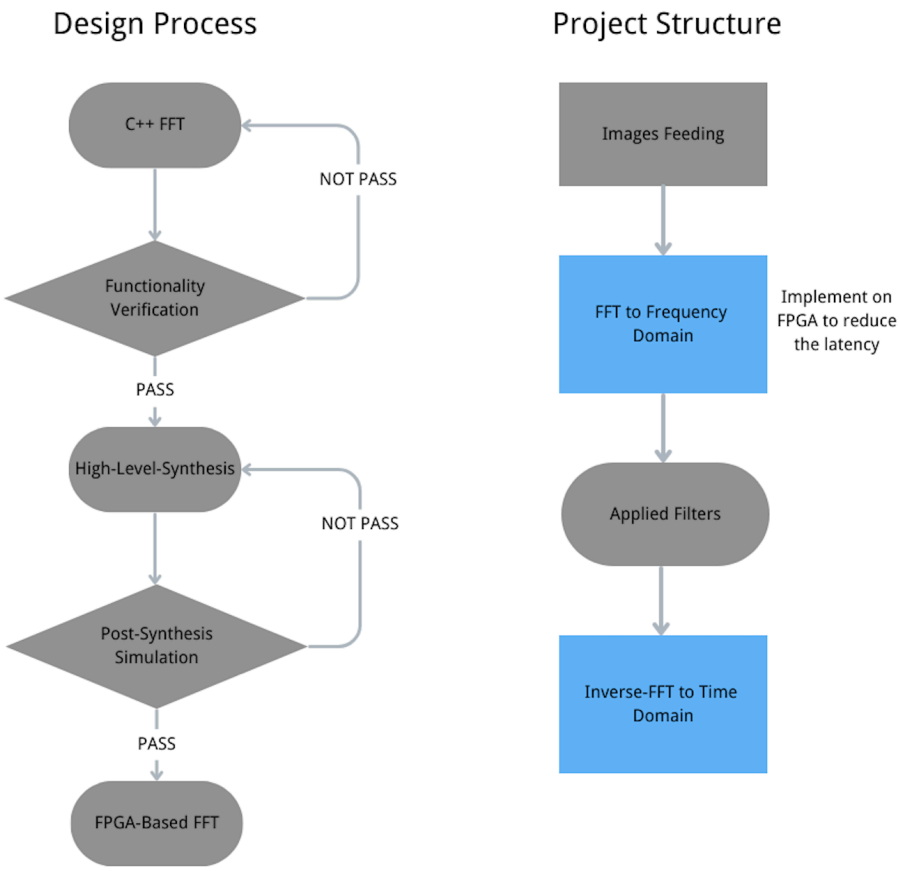


Figure 4. The Design Process Flow of the FPGA-Based FFT Accelerator

Fast Fourier Transform (FFT) Algorithm

The FFT algorithm exploits the symmetry properties of the complex exponential basis functions to reduce the number of arithmetic operations needed to compute the Discrete Fourier Transform (DFT). The principle behind the FFT algorithm is based on the divide-and-conquer strategy, which involves recursively breaking down the DFT computation into smaller subproblems. This is achieved by decomposing the original sequence into even and odd indexed elements, and then computing the DFT of each subsequence separately. These smaller DFTs are then combined using twiddle factors, which are complex exponential values that represent the phase shifts between the elements of the subproblems.

The FFT algorithm requires the following steps:

1. Divide the input sequence into even and odd indexed elements:

The input sequence is split into two subsequences: one containing the elements with even indices and the other containing elements with odd indices. This division is essential for subsequent processing steps.

2. Compute the Discrete Fourier Transform (DFT) of the even and odd subsequences separately:

The DFT of each subsequence is calculated independently. This involves applying the Fourier transform to each subsequence to obtain its frequency-domain representation. The DFT computation typically involves complex arithmetic operations.

3. Combine the DFTs of the even and odd subsequences using twiddle factors to obtain the DFT of the original sequence:

Twiddle factors, which represent phase shifts, are applied to the DFT coefficients of the odd subsequence before combining them with the DFT coefficients of the even subsequence. This combination is performed using the butterfly operation, where pairs of DFT coefficients are multiplied by twiddle factors and added or subtracted based on their positions in the sequence.

4. Repeat the process recursively for each subsequence until the base case is reached:

The FFT algorithm operates recursively, dividing the input sequence into smaller subsequences and computing their DFTs iteratively. This process continues until the base case is reached, typically when the subsequence length is 1. At this point, the DFT of the original sequence is fully computed.

By following these steps and incorporating optimizations, the FFT algorithm efficiently computes the frequency-domain representation of the input sequence, making it a fundamental tool in signal processing, communications, and numerous other fields.

To combine the subsequences of the odd and even DFT elements, an operation called 'butterfly' is utilized. The principle is shown in the following picture:

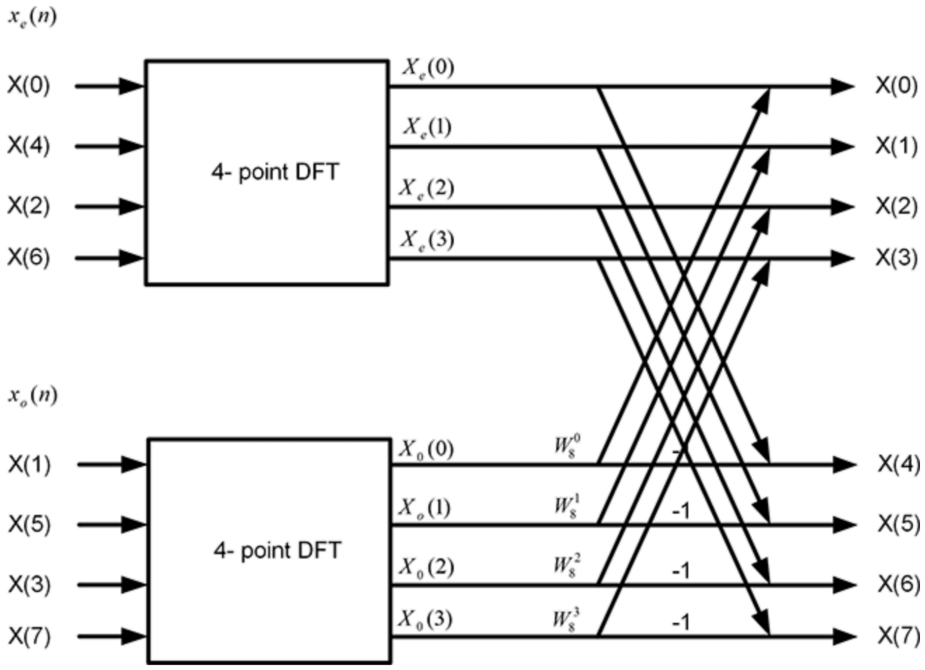


Figure 5. The butterfly algorithm

Source: *https://www.intechopen.com/chapters/53909*

The butterfly algorithm is a key component of the FFT, which involves performing complex multiplication between the Discrete Fourier Transform (DFT) coefficients of the even and odd elements, followed by the addition or subtraction of the results based on their positions in the sequence. Specifically, the butterfly operation pairs the DFT coefficients of even and odd elements and multiplies them by twiddle factors, which represent phase shifts. These twiddle factors, derived from the roots of unity in the complex plane, are crucial for adjusting the phase relationships between frequency components.

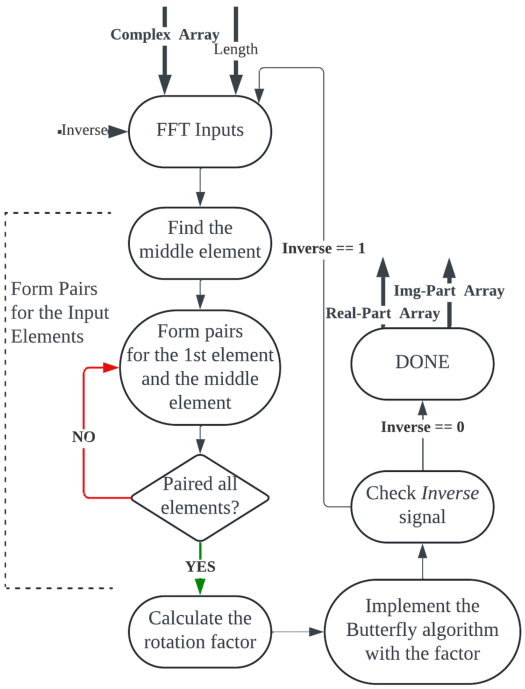
The resulting products are then either added or subtracted based on the positions of the elements within the sequence. In essence, products of coefficients in even positions are added, while those in odd positions are subtracted. This process effectively combines the frequency components to compute the DFT of the original sequence.

The Cooley-Tukey algorithm, a widely used variant of the FFT, further enhances efficiency by decomposing the DFT computation into smaller sub-problems. Specifically, it breaks down the DFT calculation of a sequence of length N into smaller DFT calculations of sequences of length N/2. This recursive approach reduces the number of arithmetic operations required, especially for sequences with lengths that are powers of 2.

Through the recursive application of the butterfly operation at each stage of the Cooley-Tukey FFT algorithm, the DFT of the original sequence is efficiently computed with significantly fewer arithmetic operations than the straightforward approach. This computational efficiency arises from exploiting the symmetry and periodicity properties of the Fourier transform, as well as the recursive nature of the Cooley-Tukey algorithm.

In our implementation, we use a function called 'rader' from open source to pair the elements [3], which will be discussed in the following section. This function utilizes the Radix-2 FFT algorithm, a specific implementation of the Cooley-Tukey algorithm that is well-suited for sequences of data points whose lengths are powers of 2. By leveraging 'rader' and the butterfly algorithm, we achieve efficient computation of the FFT, making it highly suitable for real-time signal processing tasks where swift computation is paramount.

FFT C++ Design



The FFT C++ code comes from an open-source platform called 51CTO and was contributed by user wx62f49e890a843. We chose to use open-source code because the FFT algorithm is well-established and reliable. Instead of writing the algorithm ourselves, using the code from the open-source community ensures greater accuracy and saves us considerable time, allowing us to focus on implementing FFT in hardware. We extend our gratitude to wx62f49e890a843 for their valuable contribution, which significantly expedited our process[3].

As depicted in Figure X, an image is initially converted into complex arrays, comprising both real and imaginary arrays. Subsequently, the FFT module receives these arrays, along with the array length and an inverse signal, as inputs. The module then locates the middle element of the array, as the Cooley-Tukey FFT algorithm necessitates pairing elements symmetrically to implement the butterfly algorithm. To facilitate this process, we utilize an open-source function called 'rader,' which implements the Radix-2 FFT algorithm. This algorithm is known for its efficiency in computing the FFT of a sequence of data points [3]. This function rearranges the input data into bit-reversed order, a necessary step in optimizing the computation efficiency of the FFT. After pairing the elements, the program calculates the rotation(twiddle) factor in the algorithm and then implements the butterfly algorithm on the paired elements. The FFT function is implemented to perform the Fast Fourier Transform. This function is

essential for converting the input data from the spatial domain to the frequency domain. Within the FFT function, the Radix-2 FFT algorithm is utilized, which

Figure 6. The flow chart of the FFT algorithm

involves recursive decomposition of the input data and computation of the Discrete Fourier Transform (DFT) using complex arithmetic operations. The Radix-2 algorithm efficiently handles input sizes that are powers of two, making it suitable for many practical applications. Another crucial function is 'Conv', which computes the convolution of two input sequences using FFT. Convolution is a fundamental operation in signal processing and is widely used in various applications, including image processing. In addition, another function called 'gao' which performs the convolution operation and generates the output sequence. It utilizes the FFT algorithm to perform convolution on two input sequences stored in arrays va and vb. It then processes the convolution output to obtain the final result, considering carry propagation for numbers greater than 10 and determining the highest non-zero digit. Finally, it outputs the result to the console to check the correctness.

After finishing the butterfly algorithm, the FFT function checks the inverse signal. If the inverse signal is flagged, it will apply the inverse FFT function to the output arrays to get the original arrays. Then the program outputs a real array and an imaginary array.

In addition to the algorithm, we create a main function that serves as the entry point of the program and orchestrates the execution of various tasks.

```
int main()
{
  int n;
  std::scanf("%d", &n);
  // read
  std::ifstream inputFile;
  // Open the file for reading
  inputFile.open("2D.txt");
  float line;
  int count = 0;
  //  Read and print the contents of the file line by line 2D FFT
  Complex complexArray[N][N];
  for(int j = 0; j < COL; j++) {
      for(int i = 0; i < ROW; i++) {
          inputFile >> line;
          complexArray[i][j].r = line;
          complexArray[i][j].i = 0;
          count++;
      }
  }
  inputFile.close();
  // FFT for each row
  for(int i = 0; i < ROW; i++) {
      FFT(complexArray[i], COL, 1);
  }
  // Then, FFT for each column
  for (int j = 0; j < COL; j++) {
      Complex col[N];
      for (int i = 0; i < ROW; i++) {
          col[i] = complexArray[i][j];
      }
      FFT(col,ROW, 1);
```

```
    for (int i = 0; i < ROW; i++) {
        complexArray[i][j] = col[i];
    }
}
// write in to file 2d FFT
std::ofstream myfile;
myfile.open ("Real_out.txt");
for (int j = 0; j < COL; j++) {
    for(int i = 0; i < ROW; i++) {
        myfile << complexArray[i][j].r << "\n";
    }
}
myfile.close();
// write in to file
std::ofstream myfile2;
myfile2.open ("Img_out.txt");
for (int j = 0; j < COL; j++) {
    for(int i = 0; i < ROW; i++) {
        myfile2 << complexArray[i][j].i << "\n";
    }
}
myfile2.close();
// FFT for each row
for(int i = 0; i < ROW; i++) {
    FFT(complexArray[i], COL, -1);
}
// Then, FFT for each column
for (int j = 0; j < COL; j++) {
    Complex col[N];
    for (int i = 0; i < ROW; i++) {
        col[i] = complexArray[i][j];
    }
    FFT(col,ROW, -1);
    for (int i = 0; i < ROW; i++) {
        complexArray[i][j] = col[i];
    }
}
// write in to file 2d fft
std::ofstream myfile3;
myfile3.open ("Rev_R.txt");
for (int j = 0; j < COL; j++) {
    for(int i = 0; i < ROW; i++) {
        myfile3 << complexArray[i][j].r << "\n";
    }
}
myfile3.close();
return 0;
}
```

As shown above, the ‘main’ function reads an integer input from the user, reads input data from a file called ‘2D.txt’ into a 2D complex array, performs FFT on each row and column of the array, writes the FFT-transformed data to files. It also performs inverse FFT on each row and column of the array, and writes the inverse FFT-transformed data

to another file. The FFT-transformed data is written to the real_out.txt file and the img_out.txt files, which are fed into the Matlab to convert to images.

-Matlab Verification

Initially, we used Python to merge the data from the real_out.txt and img_out.txt files into a complex array, and then converted it into an image. However, the image generated by Python exhibited significant noise. This discrepancy may stem from differences in data type definitions or declarations between the Python and C++ libraries. Consequently, we transitioned to using Matlab to perform the array-to-image conversion.

Matlab is employed for functionality verification purposes, utilizing the fft() function within its library. Initially, an image is inputted, and Matlab is utilized to convert it into a .txt file called 2D.txt, housing a real-number array. Subsequently, the fft() function is applied to transform the real-number array into a complex array. This complex array is then converted back into an image, resulting in the FFT-transformed image. This process yields both the input image and the correctly transformed FFT image for comparison and analysis.

To ensure the accuracy of our software implementation, we commence by converting the identical input image into array format, which is then processed by our C++ FFT program. Subsequently, the C++ FFT program produces the output complex form, stored in .txt files called real_out.txt and img_out.txt . These files are then imported into Matlab. Matlab takes these two files and combines them into an complex array and converted back into an image. Finally, we conduct a comparative analysis between the transformed images generated by both C++ and Matlab. The resulting images are displayed below:

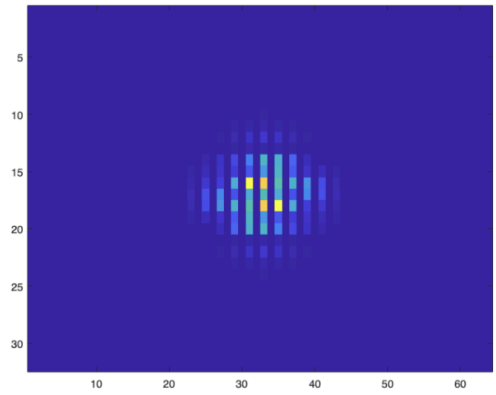


Figure 6. The Matlab generated image

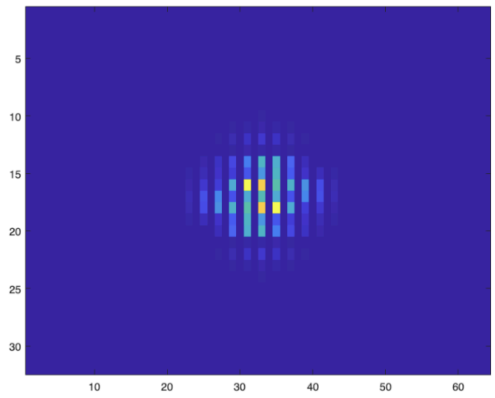


Figure 7. The C++ generated image

The pictures are exactly the same, indicating that the FFT functionality works as expected.

Furthermore, our C++ program also has the capability to perform the inverse FFT algorithm. This involves taking the complex array as input and applying the inverse FFT algorithm to generate a real-number array. The resulting array is then stored in a new .txt file called Rev_R.txt. Subsequently, we utilize Matlab to convert the .txt file into an image, allowing us to compare it with the original input image.

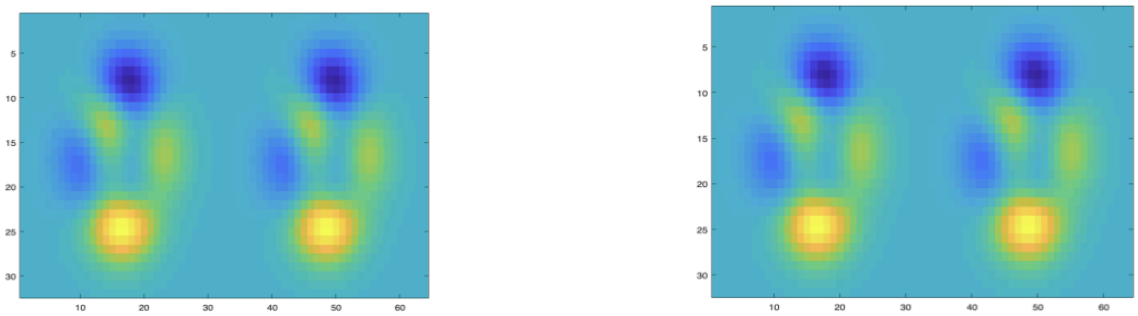


Figure 8. C++ Inverse FFT and Matlab Inverse FFT

As shown above, these images match, indicating that the functionality of our C++ implementation of the FFT algorithm is as expected.

-Software Verification

Initially, the FFT algorithm is implemented using C++ to assess the source code's functionality thoroughly. The testing process involves using a 128 x 64 scale to examine input images and validate the accuracy of the output image. This C++ program operates on a .txt file containing a 2D complex array. Upon applying the FFT algorithm to the 2D complex array, the program generates a new .txt file, storing the FFT-transformed complex array within it. Subsequently, the output .txt file is utilized in Matlab for comparison with the FFT-transformed figure generated by Matlab. The resulting figure from the C++ implementation precisely matches the one generated by Matlab, affirming the expected performance of our software implementation.

HLS and FPGA Design

Once we complete the Matlab testing phase, our next step involves deploying our forces to implement the 2D FFT code onto the ZYNQ development board.

-Camera module

Our focus turns to selecting an appropriate camera module, aligning with our goal of utilizing a camera to capture image data for input and seamlessly transmitting it to the monitor screen. Ultimately, we opted for the Pcam, the official camera available on the Digilent website. Choosing this option was facilitated by Digilent's provision of a comprehensive Pcam demo that functions seamlessly. Following the instructions provided in the documentation, which involve connecting the Pcam to the video input port of the ZYNQ board and linking the tx port to the display screen, yielded the expected video image output.(Figure 9)

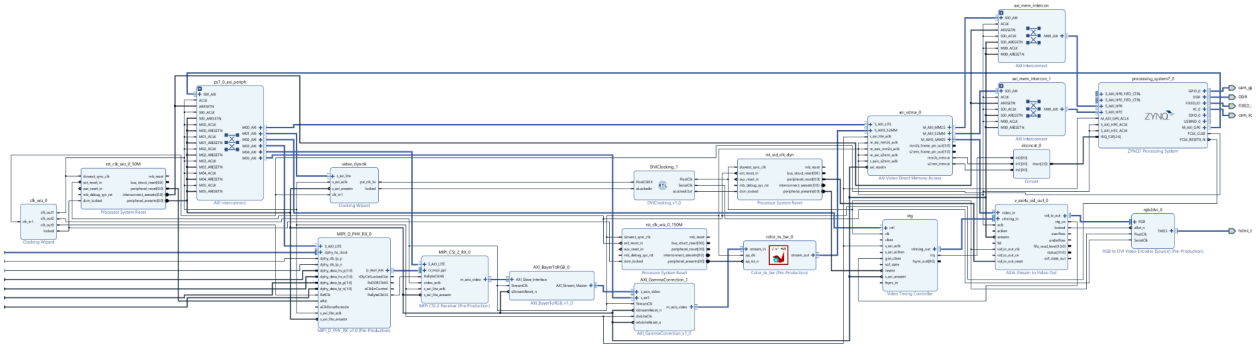


Figure 9. Camera demo for display output

This module transforms the physical camera input signal into a CSI (Camera Serial Interface) signal.(Figure 10)

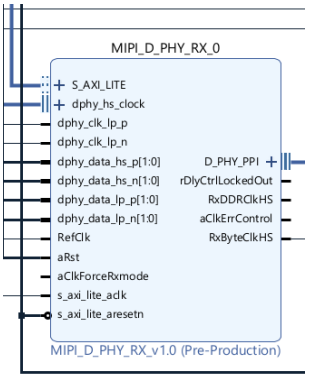


Figure 10. To CSI layer

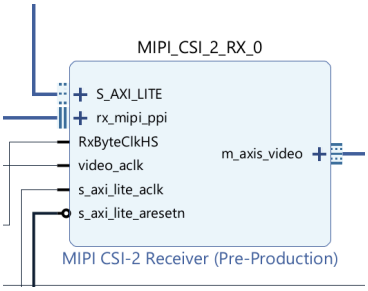


Figure 11. To AXI

It converts the CSI signal into an AXI (Advanced eXtensible Interface) video signal, enabling seamless integration with AXI-based video processing systems.(Figure 11)

To convert the Bayer light signal into an RGB signal, the module first digitizes the signal. This process involves capturing the raw light intensity data from the camera sensor, then applying a demosaicing algorithm to transform the digitized Bayer pattern into a full-color RGB image.(Figure 12)

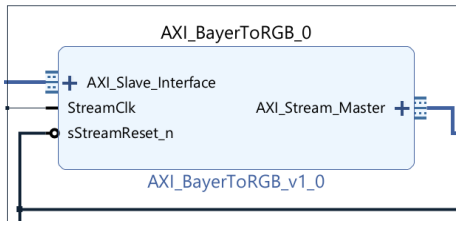


Figure 12. Bayer to RGB

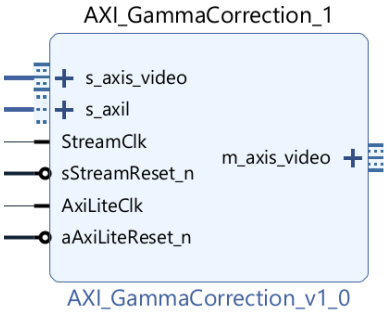


Figure 13.GamaCorrection

Ensures that what you see on your screen matches what the camera captures by mapping input values through a correction function tailored to the display device's characteristics.(Figure 13)

It store the image data, so the arm and FPGA can both use DMA to access the image data as need.(Figure 14)

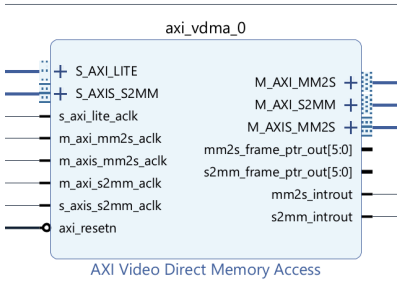


Figure 14 DMA

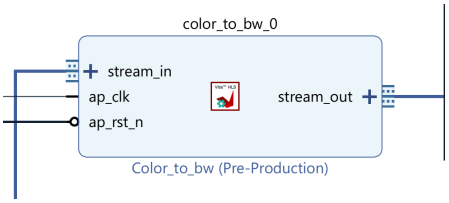


Figure 15. FFT IP core

The input and output port for the FFT IP core is really simple, because the data_steam will be captured inside the IP core itself. It will produce a continuous data stream from the steam outport. (Figure 15)

I chose to implement the FFT IP core in between because the output data needs to be modified before being written into memory. Once written into memory, modifying it becomes more complex.

-LED demo

Given the complexity of wire connections involved in the Camera demo, opting to write Verilog code directly would significantly increase the workload and make testing more challenging. Consequently, we made the decision to leverage HLS Vitis to generate the FFT IP core. As the initial step, we developed a straightforward LED module controlled by a button on the ARM core. The ARM core sent the control signal to a "signal flip module" on the FPGA, which, generated by HLS, flipped the input signal and triggered the LED accordingly. This step aimed to provide us with a better understanding of HLS functionality, IP core generation, and the utilization of HLS Vitis, VIVADO, and Vitis Classic software.

-Color to Gray and Gray to Color

The subsequent step involves implementing the alteration to convert the color image into a grayscale image for pre-FFT processing. In the camera demo, where the image data flows as a pipeline, the data is not stored within the module. Thus, we need to establish a storage space to accumulate the image data as a complete picture. This is achieved using the function ***xf::cv::AXIvideo2xfMat(stream_in, img_IN)***, which stores the input pixel in img_IN and halts the data flow once img_IN is filled. Following this, the function ***xf::cv::rgb2gray<XF_8UC3, XF_8UC1, MAX_HEIGHT, MAX_WIDTH>(img_IN, bw_img_IN)*** is employed. This function converts the 3-channel 24-bit color image to an 8-bit grayscale image and stores it in bw_img_IN.

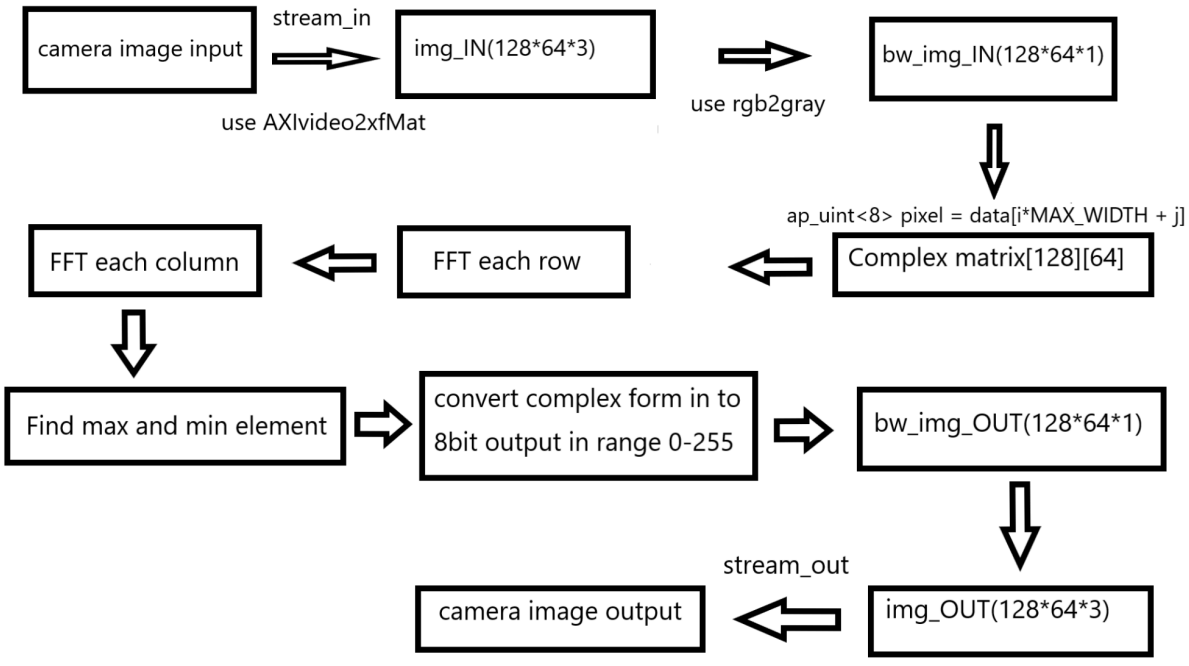


Figure 16. FFT IP core design workflow

-Convert Complex into grayscale

Once we obtain the bw_img_IN, we proceed to write the data into the complex array row[MAX_HEIGHT][MAX_WIDTH]. Subsequently, we perform FFT on the rows first, followed by FFT on the columns. Consequently, we obtain a complex 2D matrix array. To convert the real and imaginary parts to real pixel values, we utilize the following function:

output_matrix[i][j] = hls::log(1 + hls::abs(hls::sqrt(row[i][j].r * row[i][j].r + row[i][j].i * row[i][j].i)));

After this conversion, the results may vary from negative to positive. Hence, we need to design a function to adjust the data within the range of 0 to 255, which corresponds to the range of grayscale images. This is achieved by first finding the minimum and maximum values and then applying the following formula:

ap_uint<8> pixel_out = ((output_matrix[i][j] - minElement) / (maxElement - minElement)) * 255.0;

Questions and challenge

-Float to fix point

Understanding the paramount significance arises when transitioning C++ code to HLS, especially due to the shift from floating-point representation for Complex{real, img} to fixed-point representation mandated by HLS. The crucial step lies in meticulously analyzing the dynamic range and precision requisites of the data to ascertain the appropriate fixed-point format. I've come to realize that in the results of the C++ FFT, some values can reach up to 50,000, necessitating at least 16 bits of data to store the number accurately. Therefore, in comparison to precision, retaining all data states from the most significant bit seems more crucial. The position of the decimal point becomes less significant than the digit point in this context.

```
ap_fixed<20,6> r,i;
Complex(ap_fixed<20,6> r=0.0,ap_fixed<20,6> i=0.0):r(r),i(i) {};
```

-Resource

Embarking on image processing tasks demands substantial FPGA board resources. Sustaining a resolution of 1080 * 720 presents a clear challenge as each pixel requires approximately 24 bits of memory space. Moreover, the implementation of a graph for tasks like FFT algorithm necessitates the creation of two comprehensive 2D matrix graphs within the module, further amplifying resource constraints. After rigorous testing, we've determined that a resolution of 32 * 64 pixels represents the upper limit of image size achievable within the constraints of our available resources. One issue we've observed is that the FFT IP core utilizes more lookup tables (LUTs) than anticipated. Despite having 27,000 LUTs available on the FPGA board, when increasing the number of complex bits, we've noticed a significant rise in LUT resource consumption. A plausible explanation is that the High-Level Synthesis (HLS) process might not be leveraging block RAM to store complex numbers, instead opting to use LUTs directly. This could be resulting in substantial resource wastage.

HLS optimization strategy

In an effort to minimize resource consumption, I've adjusted the settings of Vitis HLS to optimize code execution. Specifically, I've attempted to disable auto-pipelining to conserve resources. However, I've observed that the resource usage remains unchanged regardless of whether the pipeline is enabled or disabled

Resource use for different ap_fix size

It is really weird that using ap_fix(32,8) causes less LUT than using ap_fix(20,6) for figures 12 and 13. The ap_fix(20,6) uses more than double of resources than the ap_fix(32,8) and we also see resource use increasing on BRAM and FF. My first guess is it may convert some of the LUT into BRAM because 32 bits is a power of 2. Then check with the Vivado website, one BRAM contains 64 bits. So it is not because 32 bits is the size of power 2. The other guess is about the HLS optimization strategy, since I ave no control over how the HLS converts the Verilog code, I can not confirm my guess.

Pipelined	BRAM	DSP	FF	LUT	UR
dataflow	25	45	10664	10717	
no	21	44	10023	9828	
no	0	0	17	90	
no	0	1	115	127	
no	0	0	22	184	
no	0	0	27	202	

Figure 17. ap_fix(32,8) with conversion function

Pipelined	BRAM	DSP	FF	LUT	U
dataflow	28	35	18401	26061	
no	9	22	7525	9054	
no	3	13	9575	15572	
no	0	0	689	635	
no	0	0	22	184	
no	0	0	27	202	

Figure 18. ap_fix(20,6) with conversion function

-Convert the complex number into pixel output

The FFT result will give you an array of complex values. The twice the magnitude (square root of sum of the complex components squared) of each array element is an amplitude. Or do a log magnitude if you want a dB scale. The array index will give you the center of the frequency bin with that amplitude. This part aim at converting complex values into a grayscale image output. Initially, it declares a 2D array called output_matrix to store the resultant image data. It then proceeds to iterate through each element of a complex input array row, computing the magnitude of each complex number using a logarithmic function. During this process, it also keeps track of the maximum and minimum magnitude values encountered. Following this, the code normalizes the computed magnitudes to fit within the range of grayscale intensities (0 to 255) and writes the normalized values to an output stream bw_img_OUT. This normalization ensures that the resulting grayscale image accurately represents the distribution of magnitudes within the input complex values, producing a visually interpretable grayscale output.

Test and Demonstration

To verify the ZYNQ board's image output accuracy, we procure a grayscale image online and resize it to 128*64 dimensions before performing Fast Fourier Transform (FFT) using Python. Subsequently, we save the original image data into a text file and feed it directly into the FFT IP core. Consequently, the FFT module now exclusively processes the text file's image data, conducting FFT on it and outputting the result to the display. This enables us to compare the FFT results obtained via Python with those from the IP core, ensuring accuracy validation.

Figure19. original grayscale image

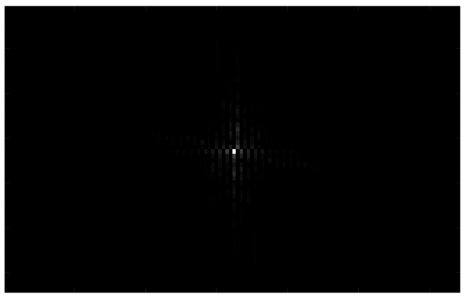


Figure 20. FFT by python

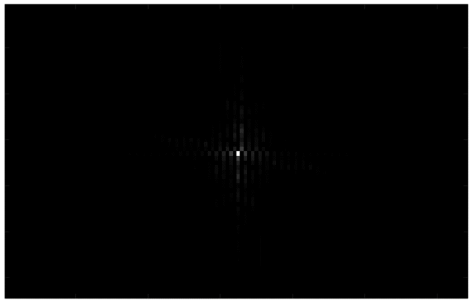


Figure 21. FFT by C++

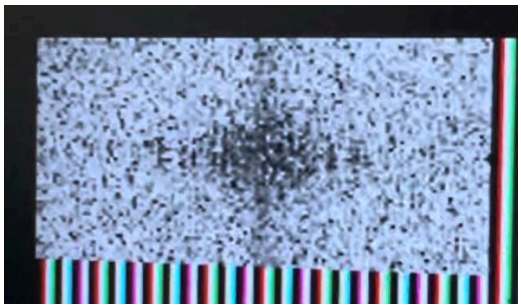


Figure 22. FFT by FPGA

To ensure the accuracy of the data loaded into the FPGA, we initially output the image without passing it through the FFT IP core.

Subsequently, we compare this image with both 1D and 2D FFT images. However, discrepancies arise as the FPGA output fails to align with the Python image for both FFT variants. Moreover, significant noise(black and white points) in the FPGA output complicates the determination of the correct FFT image.

As previously noted, the FPGA board lacks the necessary resources to accommodate processing an image of 1080 * 720 resolution. Consequently, we've reduced both the image size and resolution to conserve resources. With a final image size of 64 * 32, we're limited to a 16-bit resolution, such as ap_fix(16, 4), wherein 12 bits are allocated for the integer part and 4 bits for the decimal part. The limitation here is that the LUT resource is not enough,

Another reason is the camera module has a lot of different layers that are used to stream the data information, so if there is one small error on the first layer, the error will grow exponentially as the reaches the output layer.

-Different Tests

1. With and without the conversion function

The image on the right(Figure 19) displays the original image pre-FFT alongside the function responsible for converting complex numbers into grayscale images ranging from 0 to 255. On the left, we see the original image before FFT alongside the conversion function. Why include this conversion function? Without it, the results would directly transition from 16-bit values to 8-bit values. Unsure which data portion would be discarded, I opt to include the conversion function. This allows me to preserve all 16-bit values by proportionally converting them to 8-bit numbers.

```
ap_uint<8> pixel_out = 100 * hls::log(1 + hls::abs(hls::sqrt(row[i][j].r *
row[i][j].r + row[i][j].i * row[i][j].i)));
```

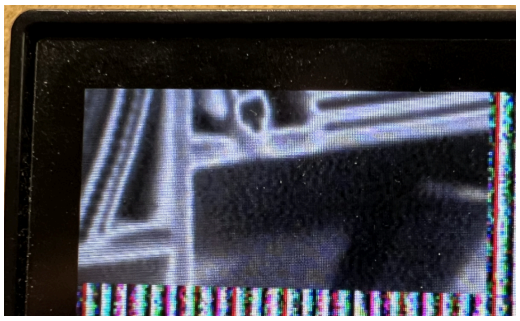


Figure 23. Without conversion function(128*64)

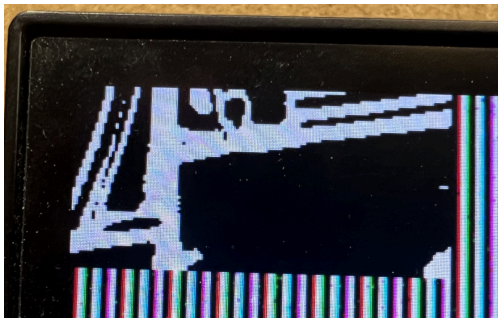


Figure 24. conversion function(128*64)

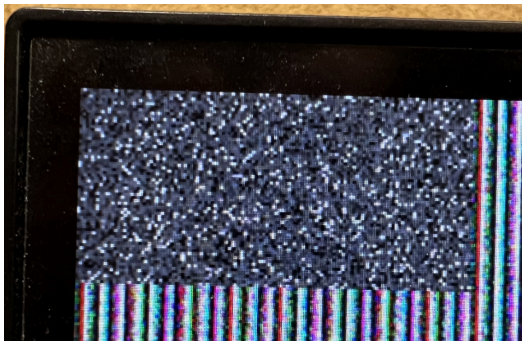


Figure 25. ap_fix(20, 6) without conversion(128*64)

Figure 26. ap_fix(16, 4) with conversion(128*64)

For Figure 20, the reason why it is (16,4) instead of (20, 9) is that after adding the conversion function, the LUT resource is not enough, so I have to reduce the size of the complex number. What I can tell from the four-figure is adding the conversion function really makes the figure different.

2. Increasing the complex number size.

Figure 27. ap_fix(32,8)(128*64)

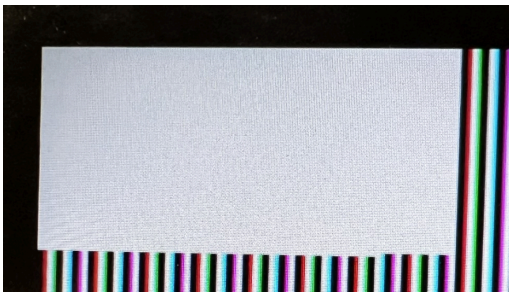


figure 28. ap_fix(32,14)(128*64)

To test the relationship between the size of complex numbers and the output image, we change the ap_fix(20, 6) into ap_fix(32, 8). In our expectation, it should increase the accuracy of the image because we directly output the original image, changing the size of the complex number is the only change we made(figure 27). However, the output result is a complete write which means it is all 255, I consider there may be an overflow that appeared during the calculation process. The output image remains the same as we increase the decimal point for the complex number, So I think the issue must occur during the conversion function from complex to image output.

Conclusions

In conclusion, this report outlines a comprehensive approach to develop an FPGA-based accelerator for ASML, focusing on the compression of real-time input images using a 2D Fast Fourier Transform (FFT) algorithm. Our strategy involves a meticulous process of developing, testing, and optimizing the FFT algorithm, first in a software environment and then transitioning to a hardware implementation on a ZYNQ Z7010 FPGA. We have addressed the initial challenges in setting up the environment and ensuring the algorithm's compatibility with the hardware. Our design process, adopting an incremental approach, ensures thorough testing and optimization at each stage.

Overall, our software implementation performs as expected, delivering satisfactory results. However, our FPGA-based solution exhibits some bugs, as indicated in the sections above. One potential cause may lie in the distinct layers of the camera, with each layer potentially magnifying noisy points. Additionally, the intricate HDL structure could lead to data accuracy mismatches among different modules. Therefore, further development is warranted. Despite these issues, the output monitor still provides real-time responses to the input. Consequently, the maximum (worst-case) operation time of the FPGA solution corresponds to the refresh rate of the camera, which stands at 15/30Hz (66/33ms). In contrast, the C++ implementation completes computation in 8365ms. Consequently, the FPGA accelerator still achieves over 100 times speedup in terms of operation time.

We anticipate that this FPGA-based solution will not only meet but exceed the performance capabilities of CPU-based systems, providing ASML with an efficient, real-time image compression tool. This project, with its focus on advanced semiconductor technology and innovative design methodologies, represents a significant step forward in the field of image processing and photolithography, positioning both our team and ASML at the forefront of technological advancements in semiconductor manufacturing.

Work Distribution

Yibin was responsible for:

- Finding the 2D FFT C++ source code.
- Writing C++ and Matlab code to test the correctness of the source code.
- Found and tested the Pcam camera module for onboard testing.
- Modify the C++ Code to the HLS version and generate the FFT IP core for Vivado.
- Write Python and HLS code to test the correctness of the FFT IP core.

Ruyi was responsible for:

- Testing part of the LED demo program on the Zynq board .
- Initial testing the HLS function for the LED demo program.
- Run software tests for different sizes of FFT modules.
- Writing and modifying the host program for the HLS code.
- Writing Python and Matlab script to convert the data to image to verify the correctness.

References

[1]Massachusetts Institute of Technology, “FFT Tutorial,” Nov. 12, 2002. [Online]. Available: https://web.mit.edu/6.111/www/f2017/handouts/FFTtutorial121102.pdf. [Accessed: Dec. 1, 2023].

[2]R. Kastner, J. Matai, and S. Neuendorffer, "Parallel Programming for FPGAs," arXiv:1805.03648v1 [cs.AR], May 9, 2018.

[3] wx62f49e890a843, 51CTO. (2022). 51CTO, https://blog.51cto.com/u_15748864/5567883?articleABtest=0.

[4]MathWorks. (n.d.). fft. MATLAB & Simulink. Retrieved May 16, 2024, from https://www.mathworks.com/help/matlab/ref/fft.html

[5]MathWorks. (n.d.). fft2. MATLAB & Simulink. Retrieved May 16, 2024, from https://www.mathworks.com/help/matlab/ref/fft2.html