MASTER

Mapping large 2D FFTs onto FPGAs using high level synthesis

Liu, T.

Award date:
2016

Link to publication

Disclaimer
This document contains a student thesis (bachelor's or master's), as authored by a student at Eindhoven University of Technology. Student theses are made available in the TU/e repository upon obtaining the required degree. The grade received is not published on the document as presented in the repository. The required complexity or quality of research of student theses may vary by program, and the required minimum study period may vary in duration.

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
Mapping Large 2D FFTs onto FPGAs using High Level Synthesis

Master Thesis

Ting Liu
0926814
t.liu@student.tue.nl

Supervisors:
prof.dr.ir. C.H. (Kees) van Berkel
dr. R.H. (Rudolf) Mak
prof.dr. H. (Henk) Corporaal

Eindhoven, August 2016
Abstract

Astronomical image processing plays an important role in astronomical observation. Central computing engines process signals received by radio telescopes to generate astronomical images, which is achieved by doing 2D Fast Fourier Transform (FFT). System executes around 1 exaflops per second according to the Square Kilometer Array project. This requirement is hard to be achieved by using CPU as the computing engine, because the operations are sequentially executed in CPU. FPGA, which supports parallel processing, is a better substitute.

The size of astronomical images are large such as 16384*16384. On-chip memory of FPGA is expensive and usually unable to store an entire astronomical image. Therefore, the external memory which has high capacity, high bandwidth and low cost is used to store images. Compared to SRAM which has high cost and DRAM which does not support burst mode, SDRAM could be a better choice.

2D FFT is basically carried out by processing images along rows and columns. Although SDRAM has high bandwidth, it has penalty to be accessed along columns. Therefore the 2D FFT algorithm should try to prevent column-wise accessing. Traditional Row-Column algorithm has poor performance because it has to access all rows and columns. Although transpose operation eliminates the column-wise accessing penalty, it requires one more time to transfer data to and from memory, which is inefficient for large images processing.

In this report, Multi-Dimension (MD) decomposition algorithm and 2D decomposition algorithm based on the paper from Arizona State University will be discussed. These two algorithms are bandwidth intensive that mitigate the data access problem of RC decomposition based algorithms without extra memory accessing. Their performance and memory requirements will be compared, followed by implementation of 2D decomposition algorithm as it needs less on-chip memory.

The design tool is Vivado High Level Synthesis (HLS). Vivado HLS transforms a C specification into a Register Transfer Level (RTL) implementation. Thus 2D decomposition algorithm can be implemented at the C level with HLS library. The data type of the design is fixed-point. The FFT IP core which supports pipeline streaming architecture helps to improve performance. Vivado HLS also provides many directives to optimize throughput, area and latency of the design. By using directives such as pipeline and dataflow, the latency of the design can be reduced.

The simulation result of the design proves that Vivado HLS has ability to implement 2D decomposition algorithm and the bandwidth of the design is close to the peak bandwidth of the SDRAM.
Preface

This thesis is a final work as partial fulfillment for the degree of Master of Embedded Systems. The thesis focuses on the implementation of high performance 2D FFT algorithm on FPGAs with SDRAM, which could be applied to astronomical images processing.

The master project including preparation starts at the middle of November, 2015 and lasts nearly 9 months.

First and foremost, I would like to thank Eindhoven University of Technology for offering me the opportunity to pursue my masters degree. It provides great study environment as well as careful assistance.

I also like to thank all people who were involved in this research. Prof.dr.ir. C.H. (Kees) van Berkel is the supervisor who teaches me how to be a superior embedded systems engineer. He is always patient with my questions and gives me many helpful suggestions. As presentation committee members, Dr. R.H. (Rudolf) Mak prof and dr. H. (Henk) Corporaal give advises to me during preparations, which is of great help for me to further improve my final report.

Furthermore, many people are appreciated for their assistance to this project. I would like to thank Pingping Lin, Mengqi Yang, Gonzalo Moro Perez, Zhaobin Zhou and Julien Grave for their help to solve some technical problems. Last but not least, I would like to thank my families for their love and support.

Ting Liu, 8,August, 2016
Contents

List of Figures ix
List of Tables xi

1 Introduction 1
  1.1 Background ................................................. 1
  1.1.1 Radio-astronomy ......................................... 1
  1.1.2 Digital Signal Processing ............................ 1
  1.1.3 FPGA+SDRAM ............................................ 2
  1.2 Image processing algorithm ................................... 5
  1.2.1 DFT and FFT ............................................ 5
  1.2.2 2D FFT: Row/Column and Transpose algorithm .......... 6
  1.3 Problem ................................................ 7

2 Preliminaries 9
  2.1 RC algorithm implementation .................................. 9
  2.2 Experiment results of RC algorithm ......................... 11
  2.3 Bandwidth-intensive 2D FFT algorithm .................... 13
    2.3.1 2D decomposition algorithm ......................... 13
    2.3.2 Multi-Dimensions decomposition algorithm .......... 14
    2.3.3 Comparison ........................................ 16

3 Method 19
  3.1 Analysis ................................................. 19
  3.2 Work flow ............................................... 21
  3.3 Proposed architecture .................................... 22
    3.3.1 Data type ........................................ 22
    3.3.2 Local memory ..................................... 23
    3.3.3 Process Elements .................................. 24
  3.4 Design tool ............................................. 25

4 Implementation 27
  4.1 2D decomposition implementation in C level .............. 27
  4.2 C++ programming in Vivado HLS .......................... 27
    4.2.1 HLS Interface protocols .......................... 28
    4.2.2 HLS data type library .......................... 29
    4.2.3 HLS FFT library ................................... 29
    4.2.4 Scaling ........................................... 31
  4.3 Optimization ............................................ 33
  4.4 Synthesis result ......................................... 37
  4.5 Simulation result ....................................... 39
CONTENTS

5 Conclusions ........................................ 43
Bibliography ........................................... 45
# List of Figures

1.1 2-element interferometer ........................................ 1
1.2 Imaging (W-projection/snapshot + CLEAN) .............................. 2
1.3 SDRAM timing .......................................................... 3
1.4 SDRAM timing of read/write cycle .................................... 4
1.5 Structure of butterfly ............................................... 5
1.6 Data access pattern for RC algorithm ............................... 6
2.1 Flow chart of reduced DFT ........................................... 10
2.2 Flow chart of reduced IDFT .......................................... 11
2.3 Lena after FFT-shift ............................................... 11
2.4 Lena after FFT .................................................. 12
2.5 Up-down symmetric image after FFT ................................. 12
2.6 Block after FFT .................................................. 12
2.7 Chessboard after FFT .................................................. 13
2.8 Circle after FFT .................................................. 13
2.9 Data access pattern for DX and local 2D DFT ....................... 15
2.10 Data access pattern for 2D DFT .................................... 16
2.11 Memory access time for different access patterns when on-chip memory size is 128*128 16
2.12 Roofline model of different size of input and different algorithms when size of input is 2048*2048 .................. 18
3.1 Amount of on-chip memory of different FFT size when B=8 .... 20
3.2 Amount of on-chip memory of different FFT size when B=32 .... 21
3.3 Work flow ............................................................ 22
3.4 The proposed FPGA architecture .................................... 22
3.5 Transfer timing for entire frames in Pipelined Streaming I/O with no Cyclic Prefix Insertion .............................. 24
3.6 Vivado HLS design flow ........................................... 26
4.1 Interface protocols of the design ................................... 29
4.2 Fixed-point data type .............................................. 29
4.3 Pipelined Streaming FFT ........................................... 31
4.4 Pipelined Streaming I/O ............................................ 32
4.5 DATAFLOW .................................................. 34
4.6 Loop unrolling .................................................. 35
4.7 PIPELINE .......................................................... 35
4.8 ARRAY PARTITION with factor 2 ................................ 36
4.9 BRAM access pattern of DX and L2DFT ............................ 36
4.10 Latency and Interval results ...................................... 37
4.11 Resource usage .................................................. 38
4.12 FFT in sub-function ............................................... 40
4.13 FFT in top-function ............................................... 40
LIST OF FIGURES

4.14 Loop without PIPELINE rewind ........................................... 41
List of Tables

1.1 Requirements specification. ......................................................... 7
2.1 Performance comparison between different algorithms when size of input is 2048*2048 ................................................................. 16
2.2 Performance comparison between different input size .......................... 16
2.3 Result of different algorithm when input size is 2048*2048 .................. 17
2.4 Result of 2D decomposition with different input size .......................... 17
3.1 Local memory requirement of MD/2D decomposition algorithm with different input size ................................................................. 19
3.2 Local memory requirement of 2D decomposition algorithm with different input size ................................................................. 19
3.3 Traffic light table of comparison among native fixed point, native floating point and floating point IP core ........................................ 23
3.4 SNR(dB) of 2D decomposition ......................................................... 23
4.1 Fixed-point data types ................................................................. 29
4.2 SNR(dB) results of different scaling schemes ..................................... 33
4.3 Synthesis result ........................................................................ 39
4.4 Simulation result ........................................................................ 41
4.5 Accuracy result ........................................................................ 41
5.1 Resources for different bandwidth .................................................. 44
Chapter 1

Introduction

1.1 Background

1.1.1 Radio-astronomy

Radio astronomy is a subfield of astronomy that studies celestial objects at radio frequencies[3].

To detect radio waves from astronomical objects, large radio antennas, namely radio telescopes are needed. Observation is limited without high resolution, so it requires radio telescopes to be extremely large. Astronomers found radio interferometry technique of great help to achieve high resolution. Instead of relying on large size of radio telescopes, this technique requires large number of different separations between different telescopes[3]. The achievable resolution is proportional to the observing frequency.

1.1.2 Digital Signal Processing

Radio interferometry works by superposing (“interfering”) the signal waves from different telescopes. The principle is waves that coincide with the same phase will add to each other while two waves that have opposite phases will cancel each other out[3].

Fig.1.1 [11] shows the basic structure of interferometer, the angle of radio antennas is solid and \( \tau_g/c \) (the speed of light) is the time delay between radio waves received by two elements.

\[
\Gamma_{12}(u, v, 0) = \int \int I(l, m)e^{-2\pi i (ul+vm)}dl dm[12]
\]

(1.1)

Figure 1.1: 2-element interferometer
The formula above is Van Cittert Zernike theorem, which plays an important role in radioastronomy digital signal processing (DSP). \( I(l,m) \) presents sky intensity, \((l,m)\) are coordinates of sky images and \(u,v\) are coordinates of the base-line vector.

Information that antennas provide are amplitude and phase of each frequencies for sky image. To process signals and know sky intensity, the best way is doing 2D Inverse Fourier Fast Transform (IFFT) to get sky image. From sky image, it is not difficult to know sky intensity.

However, Van Cittert Zernike theorem is not enough. Fig.1.2[11] shows the procedure of current radio-astronomy DSP. W-Projection is an algorithm to solve the problem arises when making wide-field images that have a non-zero projection toward the desired phase center. W-projection uses Fourier convolution theorem while CLEAN is an iterative deconvolution algorithm to clean noises. The structure helps to get more clear and complete images.

### 1.1.3 FPGA+SDRAM

Square Kilometer Array (SKA) project is an international effort to build the world’s largest radio telescope, with eventually over a square kilometer (one million square meters) of collecting area[7]. It will be much faster and have higher image resolution quality than current systems. As antennas will be widely distributed around the world, huge area of sky can be observed at the same time.

The problem is how to collect and process signals from million antennas, which required the central computing engines to have very high performance and enough storage capacity. According to SKA1-mid, system executes around 1 exaflops per second. Assuming the clock frequency of system is 1GHz and there are 100000 independent channels that can parallel run, the performance should be 10000 ops/clock in each channel.

Field Programmable Gate Array (FPGA) is a better choice than CPUs, as CPUs can only process a bandwidth equal to their clock rate divided by the number of operations per sample[8].

FPGA is a large-scale configurable logic device with many resources of logic gates. Compared to Application Specific Integrated Circuit (ASIC), FPGA is more flexible. As operations parallelly run in FPGA, it has higher throughput than CPU.

In terms of FPGA development, Xilinx provides many tutorials and documentaries for Vivado design suite which benefits beginners. Vivado has a large core library to be chosen from considering throughput and energy consumption.
As FPGA has to process massive data, local memory is not so sufficient that external memory is needed to store data. Data is transferred from external memory to local memory. And after DSP, local memory transfers data back to external memory. For large size of data, the best external memory is Synchronous Dynamic Random Access Memory (SDRAM), which has high throughput and high capacity.

Compared to Dynamic Random Access Memory (DRAM), SDRAM has burst mode that can read/write data to consecutive memory address in same row. DRAM can only read/write to one address, which means it needs to fetch a new address every time. And DRAM also spends more time on refreshing. Static Random Access Memory (SRAM) does not need to refresh and has higher speed. However, the size of SRAM is bigger and it has high cost.

The memory bandwidth of SDRAM depends not only on the frequency of local memory but also on the operation frequency of SDRAM and data-bus width. DDR(Double Data Rate) SDRAM can achieve higher data rate as it sends two data per clock. DDR2 SDRAM and DDR3 SDRAM are standard components on most of FPGA boards.

The bottleneck of SDRAM is that it has non-uniform access time. It has low latency while accessing consecutive elements in a row but high latency while accessing different rows. So it is better to prevent column-wise data access.

SDRAM has some important parameters:

CAS Latency (CL): CAS Latency (Column Access Strobe Latency) is the most important memory parameter. It is the delay time between giving read instruction and getting data on the output bus.

RAS to CAS Delay (tRCD): tRCD is the time required between the memory controller asserting a row address strobe (RAS), and then asserting a column address strobe (CAS) during the subsequent read or write command. [4].

RAS Precharge (tRP): Whenever a new row is to be activated for the purpose of accessing a data bit, a command called Precharge needs to be issued to close the already activated row[4].

Active to Precharge Delay (tRAS): tRAS is the minimum number of clock cycles needed to access a certain row of data in the memory between the data request (Active) and the Precharge command. Basically, this parameter limits when the memory can start reading (or writing) a different row[4].

Fig.1.3[4] shows timing of SDRAM and Fig.1.4[2] is the timing of read/write cycle. Changing to different row needs many cycles which can be performance bottleneck.
CHAPTER 1. INTRODUCTION

Figure 1.4: SDRAM timing of read/write cycle

(a) Timing diagram of a read cycle

(b) Timing diagram of a write cycle

Figure 1.4: SDRAM timing of read/write cycle
CHAPTER 1. INTRODUCTION

1.2 Image processing algorithm

1.2.1 DFT and FFT

Discrete Fourier Transform (DFT) converts signal from its time domain to frequency domain. The input and output of DFT are complex numbers consisting of real part and imaginary part.

It represents that a finite signal in time domain is composed of a finite combination of complex sinusoids with different frequencies. DFT plays an important role in DSP as the signal is easier to be analysed and processed from frequency domain. Moreover, DFT processes finite discrete data which can be implemented in computers or other hardware.

\[
DFT : X[k] = \sum_{n=0}^{N-1} x[n] \cdot W_N^{nk}\]

\[
IDFT : x[n] = \frac{1}{N} \sum_{k=0}^{N-1} X[k] \cdot W_N^{-nk}\]

According to the definition, DFT is invertible and periodic. Formulas above show DFT and IDFT that \(x[n]\) indicates input, \(X[k]\) indicates output and \(N\) indicates the number of sample points. The period of \(X[k]\) and \(x[n]\) is also \(N\).

DFT has circular shift, circular convolution, time-reversal and complex conjugation properties. Deducing from these properties, when \(x[n]\) are real numbers, \(X[k] = X^*[⟨−k⟩]\), \(*\) means conjugate \(X[k]\). Because image inputs are all real values, this property helps to prove whether the results of FFT are right. Also from this property, it shows that the output of FFT has Hermitian redundancy[10]. If the redundancy can be removed, the speed will improve and memory usage will decrease.

Parseval’s theorem means the energy of signal remains the same after DFT.

\[
\sum_{n=0}^{N-1} |x(n)|^2 = \frac{1}{N} \sum_{k=0}^{N-1} |X(k)|^2
\]

However, the complexity of computing DFT is \(O(n^2)\). Computation time will grow fast when \(n\) (data size) increases.

CooleyTukey FFT is an algorithm to compute DFT. It reduces the complexity to \(O(n \log n)\). A radix-2 FFT is the simplest one. It divides a DFT of size \(N\) into two interleaved DFTs of size \(N/2\) with each recursive stage. Noted that the number of sample points should be power of 2.

At each stage, inputs of even and odd index are divided. If inputs are reversal order, outputs are correct order. Therefore, bit-reversal permutation is needed to get correct ordered outputs.

\[
\begin{align*}
X_m(k) &= X_{m-1}(k) + X_{m-1}(k + 2^{m-1})W_N^r \\
X_m(k + 2^{m-1}) &= X_{m-1}(k) - X_{m-1}(k + 2^{m-1})W_N^r
\end{align*}
\]

Equations above show computation for stage \(m\). Assuming there are \(N\) inputs, each stage has to do \(N/2\) times “butterfly” operations. Fig.1.5 shows the structure of ”butterfly” operation. As \(N = 2^L\), there are \(L\) stages in total. The distance between A and B in stage \(m\) is \(2^{m-1}\).

\(r = (k)_{2} \cdot 2^{L-m}\).

Figure 1.5: Structure of butterfly
1.2.2 2D FFT: Row/Column and Transpose algorithm

DSP of radio-astronomy is mainly about 2D FFT image processing. Each pixel represents a sample point. All pixels have to be processed by row-wise and column-wise in order to get horizontal frequencies and vertical frequencies.

Traditional 2D FFT uses Row/Column (RC) algorithm which has two steps. The first step is doing row-wise 1D FFT to all rows. The second step is doing column-wise 1D FFT to all columns. The intermediate results of row-wise FFT are stored for column-wise FFT.

\[
2DFFT : F(u, v) = \sum_{x=0}^{N} \sum_{y=0}^{N} f(x, y) \cdot W_N^{ux} W_N^{vy} \quad (1.6)
\]

\[
2DIFT : f(x, y) = \frac{1}{N^2} \sum_{u=0}^{N} \sum_{v=0}^{N} F(u, v) \cdot W_N^{-ux} W_N^{-vy} \quad (1.7)
\]

Fig.1.6 shows Row-wise and Column-wise data access pattern.

Another traditional method is transpose[5]. Because column-wise operations in SDRAM are slower than row-wise operations, it is better to do row-wise operations only to maximize bandwidth. Transpose after row-wise FFT changes column-wise data to row-wise which ensures maximal bandwidth.

It is meaningless to store imaginary part and real part of outputs when doing image processing. Instead it is their magnitude and phase values that need to be computed. Magnitude means amplitude of sinusoid with corresponding frequency and phase represents position of the sinusoid. As for IFFT, real part and imaginary part of outputs can only be gotten back from magnitude and phase values.
CHAPTER 1. INTRODUCTION

Requirements

<table>
<thead>
<tr>
<th>Requirements</th>
<th>For master project</th>
<th>For ASU’s paper</th>
</tr>
</thead>
<tbody>
<tr>
<td>FFT size</td>
<td>16384 × 16384</td>
<td>4096 × 4096</td>
</tr>
<tr>
<td>Precision</td>
<td>SNR &gt; 60dB</td>
<td>SNR &gt; 67.72dB</td>
</tr>
<tr>
<td>Input data type</td>
<td>Real</td>
<td>Real or imaginary</td>
</tr>
<tr>
<td>IFFT</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Reduce Hermitian redundancy</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Fixed or floating point</td>
<td>Fixed</td>
<td>Fixed</td>
</tr>
<tr>
<td>Permutation</td>
<td>Yes</td>
<td>No</td>
</tr>
<tr>
<td>Peak bandwidth</td>
<td>3200MB/s</td>
<td>3200MB/s</td>
</tr>
<tr>
<td>Attainable performance</td>
<td>Near peak performance</td>
<td>22.27 Gops/s</td>
</tr>
<tr>
<td>Using Vivado HLS</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>

Table 1.1: Requirements specification.

1.3 Problem

According to the background, our project aims to map radio-astronomy DSP algorithm onto FPGA+SDRAM to process large images and minimize computation time.

SDRAM access time needs to be optimized. A bandwidth intensive algorithm will be chosen as the DSP algorithm which has better performance than RC and Transpose algorithms.

Both 2D FFT and 2D IFFT should be implemented and the input size should be as large as possible, it is better to be bigger than 16K by 16K.

On the other hand, FPGAs do not have native floating-point processor. Floating-point operations have worse performance than fixed-point operations on FPGAs. And floating-point arithmetic is not applicable on all FPGAs. So it is better to use fixed-point data type.

Because the dynamic range of fixed-point data is smaller than floating-point data, it will produce errors during computation. In order to get accurate results, the SNR (Signal Noise Ratio) needs to be greater than 60dB.

As the input is image that only has real values, the output of 2D FFT will have symmetry. Removing symmetric part of outputs improves speed and memory utilization. So the problem is how to remove redundancy. It will help a lot to get maximal throughput.

Instead of programming in hardware description language (HDL), it is preferable to program in C level, which is then transformed to HDL program with Vivado high level synthesis (HLS).

Final question is whether the feasibility can be demonstrated on a specific Xilinx FPGA with DSP. This research also tries to prove that the method proposed is generic for all FPGAs with DSP in case that the demonstration cannot achieve 1 exaops/sec, it is still possible for high end FPGAs.

Table.1.1 shows requirements according to problems.
Chapter 2

Preliminaries

2.1 RC algorithm implementation

To start with the project, 1D FFT in C language and RC algorithm are implemented. It consists of a few functions: weight(W) value computation, complex multiplication, complex addition, complex subtraction, FFT and IFFT.

In FFT function, the first step is doing bit-reversal. Then, calculating the number of stages, the number of points each group contains in m stage and distance between two points in a butterfly operation. The final step is doing butterfly operation. To verify output, IFFT is carried out to check whether output is the same as input.

2D FFT is also implemented. It is nearly the same as 1D FFT except that it should store intermediate values and do 1D FFT \( N \times N \) times. Another difference is the output of 2D FFT consists of phase part and magnitude part. In order to do IFFT, phase part and magnitude part should be transformed back to real part and imaginary part. After IFFT, it is possible to check if the output image is the same as input image according to the value of each pixel.

\[
\text{magnitude}(F) = \sqrt{\text{real}(F)^2 + \text{imaginary}(F)^2}
\]

\[
\text{phase}(F) = \text{atan2}(\text{imaginary}(F)/\text{real}(F))
\]

\[
\text{real}(F) = \text{magnitude}(F) \times \cos(\text{phase}(F))
\]

\[
\text{imaginary}(F) = \text{magnitude}(F) \times \sin(\text{phase}(F))
\]

BitMaP (BMP) images are used as input and only gray scale images are processed. Range of each pixel is from 0 to 255 (gray level).

2D FFT function between read BMP and write BMP steps is applied. The problem is that outputs are outside dynamic range. The big floating point value cannot fit in gray level, leading to a wrong output image. Therefore logarithmic transformation is used on magnitude part to scale it in order to fit in gray level. Also the phase part is scaled to fit in range from 0 to 180.

\[
\text{outputmagnitude} = \log_{10}(\text{magnitude} + 1) \times 255/\log_{10}(\text{height} \times \text{width} \times 255)
\]

\[
\text{outputphase} = \text{fabs}(\text{phase} \times 180/\pi)
\]

As the 0 frequency part is at the top left corner of the output image, it is hard to observe frequency distribution. The FFT-shift function moves corner part to the centre. Fig.2.3 shows frequency distribution before and after FFT-shift. Moreover, energy of image before FFT and after FFT is computed to verify Parseval’s theorem.
According to properties of DFT, outputs of 2D FFT must have Hermitian redundancy. Doing FFT to 2 rows at the same time helps to remove Hermitian redundancy. The method is: put values of even rows to real part and put values of odd rows to imaginary part. Time of row-wise FFT reduces half and only results of 0-N/2th columns need to be stored as 0th and N/2th elements are DC components. Afterwards, column-wise FFT is carried out to these N/2+1 columns. Except first row and N/2th row are left right symmetric, values of other rows is according to F(x,y)=F*(height-x,width-y).

The redundancy of IFFT is also removed. The first step is doing column-wise FFT to N/2+1 columns. Then combine values of 2 rows to one complex array and do row-wise FFT. The final results of 2D FFT are N/2 complex numbers. Real parts are values of even rows and imaginary parts are values of odd rows. An image as same as original one is obtained which proves the program runs correctly. Moreover, the number of butterfly operations is counted. The number reduces nearly half compared to the program with redundancy. Fig.2.1 and Fig.2.2[10] show flow charts of reduced 1D DFT and reduced 1D IDFT. The operational intensity will decrease a bit as the number of operations reduces nearly half but total transferred Bytes reduce less than half.

![Flow chart of reduced DFT](image.png)
2.2 Experiment results of RC algorithm

Fig. 2.4 shows results of Lena image. From magnitude part, it shows that most of energy gathered at low frequency part. Pixel value from 0 to 255 represents color from black to white, so white color means high magnitude. The reason for diagonal white line is that color changes sharply at lena’s hat. Difference between energy per pixel before and after FFT is not obvious (smaller than 0.00031).

Fig. 2.5 has white at the top and black at the bottom. The image consists of square waves. Because color only changes at vertical direction, there is just one vertical white dot line in magnitude image. The black gap between white dots means even frequency components with 0 amplitude.
CHAPTER 2. PRELIMINARIES

Color in Fig.2.6 changes not only at vertical but also at horizontal direction. So there is a white cross in magnitude part. The shape of frequencies distribution seems like numerous blocks because magnitude results also depend on the shape of original image.

The magnitude part of chessboard shows white dots distribute evenly as chessboard is also even distributed. Moreover, distance between dots depends on the size of block.

The magnitude part of circle image also consists of many circles. Central circles, also low frequency part, are brightest.

Figure 2.4: Lena after FFT

Figure 2.5: Up-down symmetric image after FFT

Figure 2.6: Block after FFT
2.3 Bandwidth-intensive 2D FFT algorithm

2.3.1 2D decomposition algorithm

The conventional 2D FFT algorithm is RC decomposition algorithm which has been implemented by C language. When mapping algorithm to the FPGA, local memory reads image from external memory and do row-wise 1D FFT. The intermediate results are stored back to external memory and then steps above are repeated to do column-wise 1D FFT. For small input size, local memory is enough to load the whole image from external memory. It only requires one reading/writing transaction between local memory and external memory. So the column-wise access bottleneck of SDRAM can be ignored because column-wise operations can be done inside local memory.

However, in general, local memory is not big enough for image processing. Therefore, the bottleneck of column-wise memory access should be the focus of this research.

Although transpose algorithm eliminates the column-wise data access bottleneck, it requires reading from and writing to external memory 3 times. It spends more time on communication which is not efficient.

Group from Arizona State University came up with a new 2D decomposition DFT algorithm which mitigates the data access problem of RC decomposition algorithms without transpose operation. It can be efficiently mapped onto FPGA based architectures[17].

Equations[17] below show how 1D DFT works. \( F_N \) is the twiddle factor matrix, \( N \) is sample size and \( \otimes \) is tensor product. \( F_p \otimes I_{N/p} \) does 2-butterfly operation to pairs of inputs. \( D_N \) is a diagonal matrix of twiddle factors. \( P_{N,p} \) denotes permutation with stride \( p \). As for radix-2 FFT, \( p = 2 \). In a word, \( F_N \) represents a N-point 1D DFT.
Images are divided into many small sub-blocks that can fit in local memory. Each sub-block can be processed individually. The data from the external memory is accessed only along rows. According to equation (11), input $X$ of size $M \times N$ is partitioned into $p \times q$ sub-blocks, each block size is $\frac{M}{p} \times \frac{N}{q}$.

\[
[X_0 X_1 ... X_{N-1}]^T = F_N \cdot [x_0 x_1 ... x_{N-1}]^T
\]  
(2.1)

\[
F_N = P_{N,p}(I_p \otimes F_{N/p}) \tilde{D}_N(F_p \otimes I_{N/p})
\]  
(2.2)

\[
\tilde{D}_N(j, j) = W_N^{j(mod \frac{N}{p})} [j/m]
\]  
(2.3)

\[
Y = P \left\{ (I_p \otimes F_{M/p}) \tilde{D}_M(F_p \otimes I_{M/p}) \cdot X \cdot (F_q \otimes I_{N/q}) \tilde{D}_N(I_q \otimes F_{N/q}) \right\}
\]  
(2.4)

There are 4 steps to do 2D DFT.

**Step 1**: Also called Data exchange (DX) operations. Doing q-point 1D FFT to all row-wise sub-blocks. It exchanges data in one block with other q-1 blocks. So every row has to do $N/q$ times q-point 1D FFT. All FFTs can be done in parallel.

**Step 2**: Doing p-point 1D FFT to all column-wise blocks which is the same as step 1.

**Step 3**: Also called local 2D DFT. Doing 2D DFT to each sub-block, namely doing $p \cdot q$ times $M/p \times N/q$-point 2D DFTs. All DFTs can be done in parallel.

**Step 4**: Also called permutation. The implementation in paper[17] does not achieve it during experiments. Although they do not need output in natural order, natural order output is needed to get correct image in this research. As a result, more permutation steps are added and consequently the computation time will increase.

In the DX stage, samples from the same position in each sub-block are loaded into local memory. This data is then used for row-wise and column-wise DX. The operations are repeated until the entire data is traversed, as shown in Fig.2.9[17].

As DX and 2D local DFT can run in parallel, capacity of local memory decides how many blocks can be processed in parallel. Parallelism helps a lot to reduce computation time.

### 2.3.2 Multi-Dimensions decomposition algorithm

Group from Arizona State University also came up with a Multi-Dimensions (MD) decomposition algorithm. MD decomposition algorithm pays attention to the burst mode of SDRAM which is read/write consecutive elements in the same row continuously. Burst size decides how many elements can be processed continuously.

Equation below shows its general form.

\[
V_2 = F_{N_2} \cdot U \cdot F_N
\]  
(2.5)

Here input $U$ and output $V_2$ are of size $N_2 \times N_1$; $F_{N_1}$ and $F_{N_2}$ are twiddle factor matrices for row and column DFT computations; $V_1 = U \cdot F_{N_1}$ can be done by applying 1D FFT along the rows of $U[16]$.

To maintain row-wise burst for column DFT computations, when doing column-wise FFT, they store burst data to different rows in local memory. Let $S$ be the size of the local memory, $S/N_2$ is the number of columns they can store in local memory. If $S/N_2$ is less than the SDRAMs
burst size $B$, then the SDRAM bandwidth is not utilized completely. To solve this problem, they decompose the column computations of size $N_2$ into 1D computation of size $m$ followed by 1D computation of size $p$, where $N_2 = m \cdot p$. Let $L = S/B$, then $p \leq L$. The procedure can be represented mathematically as follows:[16]

$$V_2 = P_{N_2,p} \cdot (I_m \otimes F_p) \cdot \hat{D}_{N_2} \cdot (F_m \otimes I_p) \cdot U \cdot F_N$$ (2.6)

The algorithm consists of 2 steps:

**Step 1. Row operations**: Doing row-wise 1D FFT to all rows of $U$. Then doing column-wise FFT of size $m$ followed by twiddle multiplications.

**Step 2. Column local FFT**: Doing column-wise FFT of size $p$ followed by column-wise permutation.

The data access pattern for the 2D DFT is illustrated in Fig.2.10. In Fig.2.10(a), $m$ rows spaced $p$ rows apart are selected from SDRAM and $N_1 \times m$ data is sent to the local memory. FFT of $N_1$ points is executed along $m$ rows. Then, column-wise $m$-point FFT followed by twiddle factor multiplication is applied as shown in Fig.2.10 (b). The result is stored back to the SDRAM. After $p$ iterations of operations above, step 1 is done. Step 2 is shown in Fig.2.10(c). $L$ rows with $B$ elements per row are stored in the local memory. $p$-point column-wise FFT is computed on $B$ columns.

If $N_2 \leq L$, the decomposition is not required.
2.3.3 Comparison

Compared to RC, 2D decomposition does not spend extra time on accessing columns. Compared to transpose operation, 2D decomposition does not need extra external memory access.

Table 2.1 and Table 2.2 are experiment results from paper[17]. They are implemented on Virtex-5FX with DDR2-400 SDRAM and the data width of complex data is 16*2.

The computation time of MD algorithm is from another paper[16]. It is running on Virtex-6 LX240T with DDR3-800 SDRAM and the data width of complex data is 32*2.

The input size of all algorithms is 2048*2048 and the frequency is 100MHz. It shows 2D decomposition spends computation time. In terms of 2D decomposition, computation time nearly linearly increases with input size.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>RC</th>
<th>Transpose</th>
<th>2D-DEC</th>
<th>MD-DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation time (ms)</td>
<td>44.0</td>
<td>30.4</td>
<td>22.4</td>
<td>26.2</td>
</tr>
</tbody>
</table>

Table 2.1: Performance comparison between different algorithms when size of input is 2048*2048

<table>
<thead>
<tr>
<th>Input size</th>
<th>1024*1024</th>
<th>2048*2048</th>
<th>4096*4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Computation time (ms)</td>
<td>5.4</td>
<td>22.4</td>
<td>90.4</td>
</tr>
</tbody>
</table>

Table 2.2: Performance comparison between different input size


CHAPTER 2. PRELIMINARIES

<table>
<thead>
<tr>
<th>Algorithms</th>
<th>RC</th>
<th>Transpose</th>
<th>2D-DEC</th>
<th>MD-DEC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attainable Gops/s</td>
<td>10.486</td>
<td>15.177</td>
<td>20.597</td>
<td>17.61</td>
</tr>
<tr>
<td>Operational intensity ops/Byte</td>
<td>7.857</td>
<td>5</td>
<td>7.857</td>
<td>7.857</td>
</tr>
</tbody>
</table>

Table 2.3: Result of different algorithm when input size is 2048*2048

<table>
<thead>
<tr>
<th>Size</th>
<th>1024*1024</th>
<th>2048*2048</th>
<th>4096*4096</th>
</tr>
</thead>
<tbody>
<tr>
<td>Attainable Gops/s</td>
<td>19.418</td>
<td>20.597</td>
<td>22.271</td>
</tr>
<tr>
<td>Operational intensity ops/Byte</td>
<td>7.143</td>
<td>7.857</td>
<td>8.571</td>
</tr>
</tbody>
</table>

Table 2.4: Result of 2D decomposition with different input size

Fig.2.11 shows memory access time for different access patterns[17]. Cycles of column-wise access is nearly 3 times more than row-wise access. It seems DX and local 2D DFT almost eliminate this bottleneck.

The most obvious difference between MD decomposition algorithm and 2D decomposition algorithm is that MD decomposition only conducts 1D decomposition.

Moreover, MD decomposition pays attention to the burst size of SDRAM. The size of FFT in MD decomposition is decided by image size, local memory size and burst size while the size of FFT in 2D decomposition is decided by image size. MD decomposition mentions that if \( \text{burstsize} \times \text{thenumberofcolumns} \leq \text{sizeoflocalmemory} \), no further implementation is needed. However, 2D decomposition does not mention this.

To make a clear comparison, a roofline model is created[13].

A roofline model is a performance model relates processor performance to off-chip memory traffic and device limitation. It is easy to find bound by this method. Peak performance shows compute bound and peak memory bandwidth shows memory bound. If the performance of an algorithm hits peak performance, it means the algorithm is efficient enough. Performance can only be improved by changing a better memory.

When \( N=2048 \), there are \( \log_2(N) = 11 \) stages, each stage has to do \( N/2 \) butterfly operations. Each butterfly operation has 10 operations: 6 ops for complex multiplication, 4 ops for complex addition and complex subtraction. So 1D FFT needs \( 11 \times 1024 \times 10 = 112640 \) ops. 2D FFT needs \( 112640 \times 2048 \times 2 = 461373440 \) ops in total. Total Bytes transferred between local memory and external memory also needs to be counted. RC and 2D decomposition require 2 times W (writing) and R (reading) operations while transpose operation needs 3 times W and R. As the paper mentions that input is image which means values are all real, only real part (2 Bytes) needs to be transferred from external memory to local memory at the beginning. Note that complex data width is 4 Bytes. Equations below show how to compute total ops, total Bytes, attainable performance and operational intensity. Table.2.3 and Table.2.4 are roofline results.

\[
\text{totalOps} = \frac{N}{2} \times \log_2(N) \times 10 \times N \times 2
\]
\[
\text{totalBytes} = \sum N \times N \times \text{dataWidthOfInputs} \times \text{accessTimes}
\]
\[
\text{attainableperformance} = \frac{\text{totalOps}}{\text{computationtime}}
\]
\[
\text{operationalintensity} = \frac{\text{totalOps}}{\text{totalBytes}}
\]

The paper [17] implements algorithms on device XC5FX200T, which has 384 DSP slices and each DSP can do 2 ops per cycle. As the working frequency is 100MHz, the peak performance is \( 384 \times 2 \times 0.1 = 76.8 \text{Gops/sec} \). MD algorithm is implemented on Virtex-6 LX240T which has 768 DSP slices. Fig.2.12 shows RC algorithm has lowest bandwidth while Transpose algorithm has highest bandwidth. The memory bandwidth of 2D decomposition and MD decomposition are near peak memory bandwidth which means performance is limited by memory bandwidth.
Figure 2.12: Roofline model of different size of input and different algorithms when size of input is 2048*2048.
Chapter 3

Method

3.1 Analysis

According to the comparison, 2D decomposition and MD decomposition algorithms utilize memory bandwidth better than RC and Transpose algorithms. As the aim of the research is to maximize the bandwidth, choice between 2D decomposition and 2D MD decomposition should be made.

Size of local memory is also an important reference. On-chip memory is expensive and more local memories need more area. It is better to choose an algorithm that requires least local memory.

According to 2D decomposition algorithm, image is divided into many sub-blocks. When processing an \( N \times N \) image, let \( N = K \times L \), then sub-block size is \( L \times L \) and the quantity of sub-blocks is \( K \times K \). If the size of local memory is \( S \times S \), then \( K \leq S \) and \( L \leq S \).

According to MD decomposition algorithm, when processing an \( N \times N \) image, \( m \times N \) elements in local memory are stored, \( N = m \times q \). After processing these elements, they are stored back to off-chip memory and store \( L \times B \) elements in local memory. If the size of local memory is \( S \times S \), then \( q \leq L \) and \( S \geq B \times L \), \( B \) is the burst size of external memory. That is to say, \( S \times S = \max \{ B \times L, m \times N \} \). When \( B \times L = m \times N \), the smallest \( S \times S \) is achieved.

Assume a \( 1024 \times 1024 \) image is going to be processed, the minimum capacity of local memory for 2D decomposition algorithm is \( S \times S = 32 \times 32 = 1024 \). In terms of MD decomposition algorithm, only when \( B = 1 \) can the same amount of local memory size as 2D decomposition be obtained. Otherwise, MD decomposition requires larger local memory size.

<table>
<thead>
<tr>
<th>MD DEC</th>
<th>( 1024^*1024 )</th>
<th>( 2048^*2048 )</th>
<th>( 4096^*4096 )</th>
<th>( 16384^*16384 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>local memory size, ( B=1 )</td>
<td>1024</td>
<td>2048</td>
<td>4096</td>
<td>16384</td>
</tr>
<tr>
<td>local memory size, ( B=2 )</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>32768</td>
</tr>
<tr>
<td>local memory size, ( B=4 )</td>
<td>2048</td>
<td>4096</td>
<td>8192</td>
<td>32768</td>
</tr>
<tr>
<td>local memory size, ( B=16 )</td>
<td>4096</td>
<td>8192</td>
<td>16384</td>
<td>65536</td>
</tr>
</tbody>
</table>

Table 3.1: Local memory requirement of MD/2D decomposition algorithm with different input size

<table>
<thead>
<tr>
<th>2D DEC</th>
<th>( 1024^*1024 )</th>
<th>( 2048^*2048 )</th>
<th>( 4096^*4096 )</th>
<th>( 16384^*16384 )</th>
</tr>
</thead>
<tbody>
<tr>
<td>local memory size</td>
<td>1024</td>
<td>4096</td>
<td>4096</td>
<td>16384</td>
</tr>
</tbody>
</table>

Table 3.2: Local memory requirement of 2D decomposition algorithm with different input size

Table.3.1 and Table.3.2 compare different local memory requirements with different input size. MD decomposition always require bigger local memory than 2D decomposition except that when \( B = 1 \) and \( \sqrt{N} \) is not integer.
CHAPTER 3. METHOD

Figure 3.1: Amount of on-chip memory of different FFT size when $B=8$

On the other hand, different algorithms require different amount of on-chip memory to utilize bandwidth. For example, the bandwidth is 3200MB/s, operating frequency is 100MHz and each pixel is 4Bytes. That is to say, 8 pixels should be transferred in one cycle. As SDRAM prefer to be accessed along rows, 8 pixels are in the same row.

To do column-wise FFTs, on-chip memory for RC algorithm has to store at least 8 columns in order not to throw the other 7 elements out when reading columns. So the memory size is at least $8 \times N$.

The local memory requirement of MD decomposition algorithm is $\max\{8 \times p, N/p \times N\}$. For 2D decomposition, the size of local memory is at least $8 \times 8$. If the size of one sub-block is bigger than $8 \times 8$, then local memory size is equal to the size of one sub-block.

Fig.3.1 shows the amount of local memory each algorithm needs to utilize bandwidth. 2D decomposition needs least on-chip memory and RC needs most. When the bandwidth increases, for example, 32 pixels should be transferred per cycle, the gap becomes wider between each algorithm. Fig.3.2 illustrates the on-chip memory requirement when $B=32$. From tables and figures, it comes to the conclusion that 2D decomposition is a better choice to achieve high bandwidth with small on-chip memory.
Figure 3.2: Amount of on-chip memory of different FFT size when B=32

3.2 Work flow

The goal of the project is minimizing computation time of radio-astronomy DSP and try to find threshold of computation time. The DSP algorithm is based on 2D decomposition algorithm and should be made applicable on FPGA. Hence the architecture of the design is based on ASU’s paper[17]. Moreover, the image processing includes permutation. Permutation without extra external memory accessing is better to be done. Hermitian redundancy of FFT will be removed to maximize throughput.

FFT consists of quantities of floating-point operations that are not supported by all FPGAs. Although some FPGAs has floating-point operator IP core, it normally has high overhead. To find the threshold of performance, it is better to convert floating-point data to fixed-point data.

The design will be implemented by C program, which is easy to code and verify. It is also a good way to understand how 2D decomposition algorithm works.

Afterwards, Vivado HLS is applied to implementing the design based on the FPGA architecture. It compiles C level languages to HDLs and produces an IP core which can be instantiated and simulated in Vivado. This is an important step to ensure the program can efficiently and correctly running on FPGAs. Fig.3.3 is the work flow of the project.
3.3 Proposed architecture

The FPGA architecture of the project is composed of several components: SDRAM, memory interface, dual local memories, memory switches and process elements (PEs). Fig.3.4 is a bird’s eye view of proposed architecture.

3.3.1 Data type

As mentioned above, the data type of this design is fixed-point because it is feasible for all FPGAs. Table.3.4 compares fixed-point with floating-point.

The following step is to decide the data width. As the floating-point 2D FFT is changed to the fixed-point 2D FFT, dynamic range of result decreases. The intermediate result and final result
### CHAPTER 3. METHOD

<table>
<thead>
<tr>
<th>R=red G=green Y=yellow</th>
<th>Native fixed point (Xilinx)</th>
<th>Native floating point (Altera)</th>
<th>Floating point IP core (Xilinx)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Easy to get high precision</td>
<td>R</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>Computation time</td>
<td>G</td>
<td>Y</td>
<td>R</td>
</tr>
<tr>
<td>Energy consumption</td>
<td>G</td>
<td>Y</td>
<td>R</td>
</tr>
<tr>
<td>Hardware cost</td>
<td>G</td>
<td>R</td>
<td>Y</td>
</tr>
<tr>
<td>Dynamic range</td>
<td>Y</td>
<td>G</td>
<td>G</td>
</tr>
<tr>
<td>Program development</td>
<td>Y</td>
<td>G</td>
<td>Y</td>
</tr>
</tbody>
</table>

Table 3.4: Traffic light table of comparison among native fixed point, native floating point and floating point IP core

Without scaling always cause overflow. The result after scaling is going to have noises. Precision loss is inevitable when doing fixed-point 2D FFT.

Table 3.3 is the SNR of different data width and input size[17]. SNR increases about 6dB when 1 additional bit is added to the data width. 2D DFTs have lower SNRs than 1D DFTs, because 2D DFTs have more stages of radix-2 butterflies, which means 2D DFTs need to be scaled more. Here comes to the conclusion that SNR is proportional to data width and inversely proportional to input size.

Equation below shows how to calculate SNR. \( X_{\text{float}}(n) \) and \( X_{\text{fix}}(n) \) are outputs after floating-point 2D FFT and fixed-point 2D FFT respectively.

\[
\text{SNR}(dB) = 10\log_{10} \left( \frac{\sum_{n=0}^{N^2} X_{\text{float}}(n)^2}{\sum_{n=0}^{N^2} (X_{\text{float}}(n) - X_{\text{fix}}(n))^2} \right)
\]  

<table>
<thead>
<tr>
<th>DFT size</th>
<th>256*256</th>
<th>512*512</th>
<th>1024*1024</th>
<th>2048*2048</th>
</tr>
</thead>
<tbody>
<tr>
<td>16 bit</td>
<td>69.67</td>
<td>69.36</td>
<td>68.72</td>
<td>67.72</td>
</tr>
<tr>
<td>20 bit</td>
<td>93.92</td>
<td>92.97</td>
<td>92.16</td>
<td>91.59</td>
</tr>
<tr>
<td>24 bit</td>
<td>117.52</td>
<td>117.15</td>
<td>116.81</td>
<td>116.83</td>
</tr>
</tbody>
</table>

Table 3.3: SNR(dB) of 2D decomposition

When data width increases, the resource usage also increases. As most of DSP slices on FPGAs includes 16 bit multiplier, it requires more DSP slices for multiplication of data that wider than 16 bit. Typically, a 16-bit Xilinx FFT core occupies only 3040% of resources compared with a 24-bit design[17]. With saved resources, more FFTs can be parallelized on the FPGA to accelerate the 2D FFT computation. Thus increasing data width has a bad effect on performance.

Although wider data width ensures higher precision, it is a trade off between accuracy and performance. Since the goal of this project is to reduce computation time, some precision would be sacrificed to reduce computational cost.

In order to maximize the performance and prevent too much precision loss, the data width of this research is 16 bit.

#### 3.3.2 Local memory

To do 2D decomposition, image in SDRAM should be partitioned to sub-blocks. Then during each step, sub-blocks are transferred to local memory and store back to the external memory after processing.

There are two identical local memories that serve as ping pong buffers. Ping pong buffers can consume and produce data at the same time which helps to utilize bandwidth. It overlaps communication with computation and thus communication cost is eliminated. Local memories
are implemented with dual-port Block RAM (BRAM) so that they can do two writing or reading operations in one clock.

A read-only memory (ROM) is also in need to store twiddle factors, which have been calculated before and are for reading only.

### 3.3.3 Process Elements

The process element consists of FFT and twiddle factor multiplication.

Xilinx provides FFT IP core that can be simply applied to PEs. Xilinx FFT IP core has 4 architecture options: Pipelined Streaming I/O, Radix-4 Burst I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O.

Pipelined Streaming I/O requires most resources but it has highest throughput. This architecture pipelines all stages of FFT to offer continuous data processing. Each stage has memory banks to store the input and intermediate data.

Assuming there are \( m \) stages in FFT, \( m \) stages can run at the same time, processing \( m \) different frames of data. Consequently memory banks need to store \( m \) rows of data and their intermediate results. It consumes lots of local memories.

The core has the ability to simultaneously perform transform calculations on the current frame of data, load input data for the next frame of data, and unload the results of the previous frame of data. As a result, it is possible to continuously stream in data and unload the results after the calculation latency\([6]\). Hence the ideal throughput is 1 datum/cycle.

Fig. 3.5 shows the general timing for back-to-back frames in the Pipelined Streaming architecture\([6]\). When frame A is being processed at stage \( X \), frame B and frame C are also being processed at different stages.

After choosing the architecture of FFT IP core, the size of FFT which depends on the size of the input has to be decided. As mentioned above, in 2D decomposition algorithm, sub-block size is \( L \times L \) and the quantity of sub-blocks is \( K \times K \). Thus K-point FFT has to be done during DX step and do L-point FFT during local 2D DFT step.

To use same FFT IP core during DX and local 2D DFT steps, only Q-point FFT will be implemented to accomplish both K-point and L-point FFTs, where \( Q = \max \{ K, L \} \).

When FFT size increases, the number of DSPs, BRAMs and other resources all increase subsequently. Because resources are limited on FPGA, the smallest value of Q needs to be found. That is to say, Q should be power of 2 that close to \( \sqrt{N} \). For example, if \( N = 2048 \), \( N = 32 \times 64 \),
then $Q = 64$.

According to 2D decomposition algorithm, all FFTs can run in parallel, enabling several PEs to be executed simultaneously.

The number of PEs can be decided after choosing the size of FFT. The number of PEs depends on the resources available on devices ($P_{res}$) and memory bandwidth ($P_{bandwidth}$). As implementing FFT IP requires many resources, the quantity of PEs is limited by the resources of devices. On the other hand, if FFT IP is running at 200MHz and the data width of complex is $16 \times 2$, the data rate of FFT IP is 800MB/s. If the bandwidth of SDRAM is 3200MB/s, then 4 PEs are enough. More PEs cannot do any favour to improve performance.

$$PE = \min\{P_{res}, P_{bandwidth}\}$$

(3.2)

3.4 Design tool

The development tool chosen is Vivado HLS. Vivado HLS enables C, C++ and System C specifications to be directly targeted into Xilinx All Programmable devices without the need to manually create RTL[14]. Designing and verifying in C instead of RTL make development faster. It is also friendly to software designers. People without hardware knowledge and HDL programming experience can also quickly handle Vivado HLS.

Vivado HLS supports data type specification (integer, fixed-point or floating-point) and interfaces (FIFO, AXI4, AXI4-Lite, AXI4-Stream). It also has extensive libraries for arbitrary precision data types, video, DSP and so on.

On the other hand, Vivado HLS provides synthesis report after successful synthesis that includes useful information of the whole design and each component. It helps us to optimize the design according to the latency and resource usage of each part of the program. At the same time, HDL files are produced. The report shows[15]:

- **Area**: Amount of hardware resources required to implement the design based on the resources available in the FPGA, including look-up tables (LUT), registers, block RAMs, and DSP48s.
- **Latency**: Number of clock cycles required for the function to compute all output values.
- **Initiation interval (II)**: Number of clock cycles before the function can accept new input data.
- **Loop iteration latency**: Number of clock cycles it takes to complete one iteration of the loop.
- **Loop initiation interval**: Number of clock cycle before the next iteration of the loop starts to process data.
- **Loop latency**: Number of cycles to execute all iterations of the loop.

When synthesis is completed, in the Vivado HLS GUI, the Analysis Perspective includes the Performance tab, which allows us to interactively analyze the results in detail.

The function correctness of the design is based on both C program and compiler directives. The design can be verified by using C/C++ test bench simulation and automatic VHDL or Verilog simulation. After RTL verification, it is possible to check simulation waveform. If it is correct, it means C program is successfully translated to VHDL or Verilog program. Then, after exporting RTL, an IP core which can be instantiated in Vivado design tool, System Generator or other Xilinx platform studio is achieved.

Fig. 3.6 is the design flow of Vivado HLS.

However, it is difficult to understand how directive works. The main difference between Vivado HLS coding and normal C level coding is that Vivado HLS provides many directives. It also
supports configurations that are only available for hardware, for example, clock frequency and reset.

Using directives in C code can optimize throughput, latency and area. For example:

- Instruct a task to execute in a pipeline.
- Specify a latency for the completion of functions, loops and regions.
- Specify a limit on the number of resources used.
- Set dependencies in the code and permit specified operations.
- Select the I/O protocol to ensure that the final design can be connected to other hardware blocks with the same I/O protocol.

Although Vivado HLS supports C/C++/system C programming, it does not support some C constructs. The following constructs are not synthesizable, or can result in errors.

- **System calls**: System calls such as `printf()` and `fprintf()` cannot be synthesized because they are actions that relate to performing some tasks upon the operating system in which the C program is running.

- **Dynamic memory usage**: Any system calls that manage memory allocation within the system cannot be synthesized, taking `malloc()` and `free()` with function of creating and releasing memory space during runtime for examples. But a hardware implementation must specify all resources requirements before synthesis, memory cannot be released during run time.

- **Pointer limitations**: Vivado HLS does not support general pointer casting, but supports pointer casting between native C types. Vivado HLS supports pointer arrays. But Arrays of pointers cannot point to additional pointers.

- **Recursive functions**: Recursive functions cannot be synthesized.
Chapter 4

Implementation

4.1 2D decomposition implementation in C level

In order to be understood comprehensively by practice, 2D decomposition algorithm will firstly be implemented in C level. It imitates the FPGA architecture. Different from RC algorithm, 2D decomposition needs to divide image into sub-blocks, perform twiddle factor multiplication and permutation.

The most difficult part is transaction between external memory and local memory. A big 2D array is regarded as the image in SDRAM and two small 2D arrays are used as local ping pong memory. When assign values, indexing is really important. Although reading and writing operations between SDRAM and local memory are along rows, addresses are not sequentially increasing.

In terms of twiddle factor multiplication, an array is used to store twiddle factor. Index of array is decided by the position of pixel when doing multiplication. Because image has been divided into many sub-blocks, the coordinate of each value in original image need to be traced.

Because ASU’s paper does not explain bit-reversal part clearly, the output of each FFT is assume to be in bit-reversed order. It leads to a wrong result and nearly two weeks are spent on debugging and searching for reasons. Then it is proved that the output of each FFT should be in natural order, which means either the order of input (decimation in time) or the order of output after FFT (decimation in frequency) has to be changed.

When all FFTs are done, permutation needs to be carried out with stride \( p \) \( p \) which depends on the size of sub-blocks.

A stride permutation can be defined using matrix representation[1]. Given an \( N \)-element data vector \( x \) and a stride \( p \) \( 0 \leq p \leq (N - 1) \), and the data vector \( y \) produced by the stride-by-p permutation over \( x \), then

\[
y = P_{N,p}x
\]  

where \( P_{N,p} \) is called the permutation matrix. \( P_{N,p} \) is an invertible \( N \times N \) bit matrix such that

\[
P_{N,p} = \begin{cases} 
1, & \text{if } j = (p \times i) \mod N + (\lfloor p \times i/N \rfloor); \\
0, & \text{otherwise.}
\end{cases}
\]  

4.2 C++ programming in Vivado HLS

After verifying the C program for the algorithm, it should be feasible on FPGAs. Hence the C program is modified in Vivado HLS. The data type of the FFT in C program is floating-point. As mentioned above, floating-point operations are not efficient on FPGAs, that should be transformed to fixed-point. Vivado HLS supports fixed point data type in C++ language, thus
the programming language is changed to C++. Implementation in Vivado HLS will be introduced later.

4.2.1 HLS Interface protocols

In an RTL design, input and output operations in C based design must be performed through a port in the design interface with an I/O (input-output) protocol.

Vivado HLS supports interface synthesis and manual interface specification for specifying the type of I/O protocol used. Manual interfaces specification allows any arbitrary I/O protocol but it is only available for System C designs. Thus the project only focuses on interface synthesis which creates the port interface based on industry standard interfaces.

Interface synthesis synthesized the arguments of top-level function into RTL ports. Vivado HLS creates three types of ports on the RTL design:

- **Clock and Reset ports**: `ap_clk` and `ap_rst`.

- **Block-Level interface protocol**: This protocol is added to the design by default. The ports are `ap_start`, `ap_done`, `ap_ready`, and `ap_idle`. `ap_start` controls when the block can start processing data. `ap_ready` indicates when it is ready to accept new inputs. `ap_idle` indicates if the design is idle. `ap_done` indicates the operation is completed.

- **Port Level interface protocols**: Port level interface protocols are created for each argument in the top-level function and the function return. The port-level IO protocols are used to sequence data into and out of the block after the block starting processing data. Function arguments which are both read from and write to are split into separate input and output ports. If the function has a return value, an output port `ap_return` is implemented to provide the return value.

The port level interface protocols used in the design includes `ap_memory` and `ap_fifo`. The `ap_memory` interface is used to implement array arguments. It can communicate with memory elements (for example, RAMs and ROMs) when the implementation requires random accesses to the memory address locations. The array targets are specified using the RESOURCE directive to determine whether to use a single or dual-port RAM interface. This protocol is used to specify the ports of local memory and external memory in the design.

If there only needs sequential access to the memory element, use the `ap_fifo` interface is a better choice. The `ap_fifo` interface reduces the hardware overhead, because it does not need address generation. Note that the `ap_fifo` interface can not be specified on an argument that is both read from and write to. This protocol is used to specify the ports of FFT core.

Fig.4.1 indicates the interface protocols of our design.
4.2.2 HLS data type library

Vivado HLS provides many libraries that include fixed-point data type, arbitrary precision data types and FFT IP core, which makes the implementation of the design more simple.

Vivado HLS offers arbitrary precision fixed-point data types for use with C++ and SystemC functions as shown in the following table[17].

<table>
<thead>
<tr>
<th>Language</th>
<th>Fixed-point data type</th>
<th>Required header</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>Not applicable</td>
<td>Not applicable</td>
</tr>
<tr>
<td>C++</td>
<td>ap::ufixed(W, I, Q, O, N)</td>
<td>#include “ap_fixed.h”</td>
</tr>
</tbody>
</table>

Table 4.1: Fixed-point data types

W is the Word length in bits. I is the number of bits used to represent the integer value (the number of bits above the decimal point). Q is the Quantization mode. O is the Overflow mode which dictates the behavior when the result is overflow. N defines the number of saturation bits in overflow wrap modes. W is set as 16 and I as 1. Other parameters keep default value.

Arbitrary precision data types transform the value of floating point numbers to let them fit in the boundaries of a specified total width and integer width, Fig.4.2 shows how it works.

4.2.3 HLS FFT library

In terms of FFT, there are a few steps to use Xilinx FFT ip core in C++ code:
1. Include the `hls_fft.h` library in the code.

2. Define the static parameters of the FFT, such as input width, number of channels and type of architecture, which do not change dynamically. They can be defined by using pre-defined struct `hls::ip_fft::params_t`. The width of configuration and status should be changed according to the size of FFT, otherwise it will cause error when doing simulation. Note that the HLS FFT namespace should be used to specify parameters which are not integer or boolean.

Ordering option will be set as natural order and `has_nfft` as true in order to run time configure the size of FFT. Note that the run time size of FFT cannot exceed the size defined by `max_nfft`. The default value of architecture option is pipelined streaming I/O, and we will use default scaling option, they do not need to be changed. The following example shows how to define, where `param1` is the name of user defined structure.

```c
struct param1 : hls::ip_fft::params_t {
  static const unsigned ordering_opt = hls::ip_fft::natural_order;
  static const unsigned config_width = FFT_CONFIG_WIDTH;
  static const unsigned status_width = FFT_STATUS_WIDTH;
  static const bool has_nfft = true;
};
```

3. Define types and variables for both the run time configuration and run time status. These values can be dynamic and are therefore defined as variables in the C code which can change and are accessed through APIs.

```c
typedef hls::ip_fft::config_t<param1> config_t;
typedef hls::ip_fft::status_t<param1> status_t;
cfg_t fft_config1;
status_t fft_status1;
```

4. Set the run time configuration. This example sets the direction of FFT and scaling scheme of FFT.

```c
fft_config1.setNfft(9);  // if run time configuration is available
fft_config1.setDir(direction);
fft_config1.setSch(0x2AB);
```

5. Call the FFT function using hls name space. The function parameters consist of input sample, output result, output of status and input of configuration.

```c
hls::fft<param1>(input, output, &status, &config);
```

6. Check the output status. The example shows how to check whether FFT is overflow.

```c
*ovflo = fft_status1->getOvflo();
```

The data types of input and output are defined by user. They should be arrays and both arrays should be streaming because the ports on the FFT RTL block will be implemented as AXI4-Stream ports. If the interface protocol of arrays is not set as `ap_fifo`, it will cause error.

Fig.4.3 is the simulation waveform of a 64-point FFT. It indicates the streaming input `xn,dout` and streaming output `xk,din`. The gap between input and output is the fixed latency of 64-point FFT. After the latency, output data is produced each cycle. It proves that the throughput of pipelined streaming FFT is 1 output/cycle.
4.2.4 Scaling

According to Xilinx FFT IP, pipelined streaming I/O FFT includes radix-2 FFTs at each stage. Each butterfly in radix-2 FFT picks up two complex numbers, respectively, and two complex numbers to the same memory. The numbers returned to memory by the core are potentially larger than the numbers picked up from memory. A strategy must be employed to accommodate this dynamic range expansion.

As fixed-point data type is used and the data width is 16 bit, the intermediate and final results of FFTs should be scaled down to fit in 16 bit.

For Radix-2 FFT, the bit growth is by a factor of up to \(1 + \sqrt{2} \approx 2.414\), that is 2 bit. Xilinx FFT IP core supports three scaling schedules:

1. Performing the calculations with no scaling and carrying all significant integer bits to the end of the computation
2. Scaling at each stage using a fixed-scaling schedule
3. Scaling automatically using block floating-point

When using full-precision unscaled arithmetic, the width of the datapath increases to accommodate the bit growth. The growth of the fractional bits created from the multiplication are truncated (or rounded). The width of the output is \((\text{inputwidth} + \log_2(\text{transformlength}) + 1)\). This accommodates the worst case scenario for bit growth[6].

The complex multiplier preserves the magnitude of an input, which may produce bit-growth when the magnitude of the input is greater than 1 (for example, 1+j has a magnitude of 1.414). So there is a +1 in the output width in case of complex multiplier bit growth. For example, a 1024-point transform with an input of 16 bits consisting of 1 integer bit and 15 fractional bits has an output of 27 bits with 12 integer bits and 15 fractional bits.

In the scaled fixed-point mode, a scaling schedule is used to divide by a factor of 1, 2, 4, or 8. The scale factor \(S\) is defined in equation below, where \(b_i\) is the scaling (specified in bits).

\[
S = 2^b, b = \sum_{i=0}^{\lfloor \log(N)/2 \rfloor} b_i
\]  

The data is scaled after every pair of Radix-2 stages for the Pipelined Streaming I/O architecture. Assuming 2 stages as a group, it results in group 0, group 1 . . . . Groups are computed starting with group 0 as the two LSBs. In each group, the data can be shifted by 0, 1, 2, or 3 bits which corresponds to SCALE_SCH values of 00, 01, 10, and 11. When the point size is not a power of 4, the last group only contains one stage, and the maximum bit growth for the last group is one bit. For example, when \(N= 512, [01 10 00 01 11]\) means group 0 (stage 0 and 1) right shift by 3, group 1 (stage 2 and 3) right shift by 1, group 2 (stage 4 and 5) no shift, a shift of 2 in group 3 (stage 6 and 7) and group 4 (stage 8) shift by 1.
CHAPTER 4. IMPLEMENTATION

With block floating-point, each stage applies sufficient scaling to keep numbers in range, and the scaling is tracked by a block exponent\[6\].

Fig.4.4 is the architecture of pipelined streaming I/O\[6\].

According to the data from Xilinx FFT IP core guidance\[6\], unscaled schedule has highest dynamic range when scaled schedule has lowest dynamic range. Hence, unscaled schedule has highest SNR when scaled schedule has lowest SNR.

According to Equation (1.2) and (1.3), when doing IDFT, the output is divided by N, which is the size of FFT. If we do division during DFTs, then IDFTs do not need to do division. N can be the scaling factor, which means there is no need to do scaling during IFFTs. As the design consists of 2D FFTs and 2D IFFTs are also implemented to verify the design, the scaling routes can be:

- **FFT**: $X(k) \times N \rightarrow IFFT$: $x(n)$
- **FFT**: $X(k) \rightarrow IFFT$: $x(n)$
- **FFT**: $X(k) \rightarrow IFFT$: $x(n) \times N$

Compared to other routes, first scaling route has least probability to cause overflow.

A proper scaling schedule should be chosen for the design. Although unscaled schedule can achieve highest SNR, it is more applicable for one time FFT. The design has to do multiple FFTs to the same image and the output of each FFT will be the input of the later FFT. It is not practical to change the width of datapath during run time.

The block floating-point mode might use more BRAM and DSP slices than the scaled mode. Using block floating-point mode may limit the parallelization of FFTs. This is a trade off but scaled schedule is chosen as resources and throughput are more important in this project than precision.

The parameters used to verify the output of IFFT are SNR and Root Mean Square (RMS). SNR and RMS are applied to compare original image to processed image. Moreover, SNR and RMS of floating-point processing and fixed-point processing are compared as well. Equation below shows how to get RMS result.

$$RMS = \sqrt{\frac{\sum_{n=0}^{N^2}(x_{original}(n) - x_{processed}(n))^2}{N \times N}}$$ (4.4)
The way to calculate SNR after IFFTs is different from what is mentioned above, that is:

$$SNR(dB) = 10 \log_{10} \left( \frac{\sum_{n=0}^{N^2} x_{original}(n)^2}{\sum_{n=0}^{N^2} (x_{original}(n) - x_{processed}(n))^2} \right)$$ (4.5)

After selecting the scaling schedule, the scaling scheme has to be decided. ASU’s paper adopts half-a-bit-per-stage scheme [17] for scaling. They first scale the input array so that $|x(i_1, i_2)| < 1/2.4142$ to prevent overflow in the first stage. Thereafter they right-shift 1 bit for every pair of stages.

However, half-a-bit-per-stage scheme is not proper for the design of this project, where 3 images of Lena, Random and Big are processed. Size of Lena and Random is $512 \times 512$ while size of Big is $1024 \times 1024$. Pixels in Random are produced randomly from 0 to 255. Both half-a-bit-per-stage scheme and one bit per stage scheme are applied to FFTs.

The experiment results (Table 4.2) turn out that overflow happens and even sometimes, one bit per stage scheme achieves higher SNR.

Based on the analysis above, scaled schedule with one bit per stage scheme will be applied to the design of this project in order to save resources, prevent overflow and maximize throughput.

### 4.3 Optimization

Vivado HLS provides many directives to optimize throughput, latency and area. The main feature of Vivado HLS is directives. Using directives in C level code can achieve Hardware behaviors.

As 2D FFT is done at the C-level in this project, it requires amount of loop operations to pass by values and do multiple 1D FFTs. Loops in C program are executed iteratively, which cause high overhead when implemented on FPGAs. FPGA has advantage that operations can run in parallel. In order to benefit from this feature, this research focuses on directives DATAFLOW, UNROLL, PIPELINE and ARRAY PARTITION.

- **DATAFLOW**: Loops execute in order without optimization, but DATAFLOW optimization allows data from one loop flow to next loop in sequential order. If later loop consumes data that is produced in the previous loop, it does not need to wait until previous loop finished. Once the data is produced, it will immediately flow to the next loop. DATAFLOW optimization is a powerful method for improving design throughput. In the design of this project, the default port type of FFT core is FIFO while two arrays are set as FIFO arrays for input and output of FFT core. A value needs to be passed from BRAM to FIFO and after FFT, and loaded from FIFO to BRAM, it requires 2 sequential loops. When using DATAFLOW directive, each cycle there can be one input and one output. Fig.4.5 shows how DATAFLOW optimization allows the execution of tasks to overlap, overall throughput increases.

However, there are some strict rules to use DATAFLOW directive. To realize the DATAFLOW optimization, the data must flow through the design from one task to the next, resulting in that array can only be produced and consumed one time. Reading before writing is not allowed. There should be no feedback between tasks. Conditional execution is better inside loop, otherwise, it would not benefit from DATAFLOW.
CHAPTER 4. IMPLEMENTATION

Figure 4.5: DATAFLOW

- **UNROLL**: UNROLL directive has the ability to complete unroll or partially unroll for loops which allows all iterations to occur in parallel. When the loop is rolled, each iteration is performed in a separate clock cycle. When loop is complete unrolled, each iteration is performed in one clock cycle. However, it requires more logic copies to unroll loops. Fig.4.6 shows how UNROLL directive works. It is applied to make multiple FFT IP cores run in parallel. The number of FFT cores is 8 as 8 pixels should be processed per cycle to meet the bandwidth which is 32Bytes/cycle. After unrolling, the resource of FFT core is 8 times more than before.

![Diagram of loop unrolling]

Figure 4.6: Loop unrolling

- **PIPELINE**: PIPELINE directive allows operations to happen concurrently: the task does not have to complete all operations before it begin the next operation. If a loop is pipelined, any loop in the hierarchy would be unrolled. PIPELINE directives are used to pass value from BRAM to FIFO of each FFT IP core, so each cycle pass 8 values to 8 FFT IP core. Top-level loop in a function or in the DATAFLOW region can continuously execute with PIPELINE rewind. That is at the end of the loop iteration count, next loop immediately starts to execute. Fig.4.7 shows how PIPELINE works.

![Diagram of pipeline operation]

Figure 4.7: PIPELINE operation

34 Mapping Large 2D FFTs onto FPGAs using High Level Synthesis
As DATAFLOW, UNROLL and PIPELINE do not support loop scopes with variable bounds, the FFT size, sub-block size and other parameters that loop bound involves are calculated and stored as constant data in program.

On the other hand, if a loop is completely unrolled, all operations will be performed in parallel. If the data in one iteration of the loop depends on the result from a previous iteration, they cannot execute in parallel because of data dependency.

Although UNROLL and PIPELINE can make operations perform in parallel, limited resources may block this. In the design of this project, 8 pixels should be transferred each cycle. Arrays are implemented as block RAM which only has a maximum of two data ports. As a result, one BRAM needs at least 4 cycles to transfer 8 pixels. ARRAY PARTITION directive can be used to solve this problem.

The bandwidth can be improved by splitting the array into multiple smaller arrays (multiple block RAMs), effectively increasing the number of ports.

ARRAY PARTITION directives split the array into multiple smaller arrays to increases the number of data ports so that the bandwidth can be improved. Vivado HLS provides three types of array partitioning: block, cyclic and complete.

- **block**: If partition factor is 8, array is split into 8 smaller arrays. Size of each block is equal and elements in each block are consecutive.

- **cyclic**: The original array is split into equally sized blocks interleaving the elements of the original array. [15]

- **complete**: Array is split into its individual elements. Each element will be stored in a register instead of BRAM.

As multi-dimension arrays need to be split, it is essential to specify which dimension would be partitioned. Fig.4.8 is the detail of different array partition types, first dimension is split into 2 parts or completely. This directive helps partition BRAMs and FIFOs, so that it is possible to pass 8 pixels from BRAM to FIFO within one cycle.
CHAPTER 4. IMPLEMENTATION

Figure 4.8: ARRAY PARTITION with factor 2

After array partition, it may still have problem due to limited data port. In the design, it should be ensured that 8 pixels are read from BRAM at the same time. That is to say, they should be stored in 8 different BRAM. In data exchange step, the block size in local memory is different from the block size in SDRAM. Consecutive elements in each local memory block are not consecutive in SDRAM. They are from different blocks that have different row and column addresses. In local 2DFT step, the block in local memory has same structure as the block in SDRAM. Consecutive elements in local memory are also consecutive in SDRAM.

Therefore, a general interface structure is required to store or load 8 pixels per cycle. The local memory size of $8 \times p \times q$ is able to do this, where $p \times q$ is the size of the sub-block in data exchange step. Because the size of the sub-block in data exchange step is larger than in local 2DFT step, local memory can at least store 8 sub-blocks. After array partition with block style, BRAM is split into 8 blocks.

During data exchange, SDRAM transfers 8 consecutive elements in same row and then they are sent to 8 different BRAM respectively. During local 2DFT, SDRAM transfers 8 elements at the same position to 8 sub-blocks. Although burst mode can not be applied, there is no delay between loading 8 elements. The disadvantage is that the resource usage would be higher than burst mode. Fig.4.9 is the BRAM access pattern of the design. This interface also ensures bandwidth when doing column-wise operations on chip.

Figure 4.9: BRAM access pattern of DX and L2DFT
4.4 Synthesis result

In this section, synthesis results are all based on the 2D-decomposition of a 64*64 image. The device is Zedboard Zynq which has 280 BRAM18k and 220 DSP slices. The working frequency is 100Mhz. Local pingpong memory and memory interface are not included in the design. The design includes interface of local memory, ROM for twiddle factors and PEs. PE includes FFT IP cores, input FIFOs and output FIFOs for FFT and multiplier units.

The reason why local memory has not been implemented is that DATAFLOW only allows same array to be read from and written to per time. Reading before writing is also forbidden. Thus when doing row-column FFT during DX and L2DFT, it is not possible to read from same array and write to same array. Neither is writing to different array when doing row-wise FFT nor reading it when doing column-wise FFT feasible.

Fig.4.10 is the synthesis report from HLS. Latency is the total clock cycles for top-function. If top-function has been called multiple times, interval is the time between the start point of the previous one and the start point of the next one. Latency and interval here represent clock cycles program takes to do 8 times 8-point FFT. Min and max values are different because of the conditional statement inside top-function. Min is the minimum number of cycles when all conditions are false. Max is the maximum number of cycles when all conditions are true. Loop1 in report is for loading data from BRAM to FIFOs of FFTs. Loop1 has 8*8 iterations and there are conditional statements inside loop as entries of DX-row FFT, DX-col FFT, L2DFT-row FFT and L2DFT-col FFT steps. Loop2 is for reading data from FIFOs to BRAM and doing twiddle factor multiplications.

UNROLL directive is applied to FFT cores while DATAFLOW is applied to whole program. Because of UNROLL directive, HLS instantiates 8 FFT cores that can run in parallel. Fig.4.11 is the resource usage detail from HLS, BRAM is used as ROM for twiddle factors while DSP is used for FFT cores and twiddle factors multiplications. Each FFT cores requires 3 DSP slices, thus 8 FFT cores hold 24 DSP slices. The other 12 DSP slices are for multiplications.

<table>
<thead>
<tr>
<th>Instance</th>
<th>Module</th>
<th>Latency</th>
<th>Interval</th>
</tr>
</thead>
<tbody>
<tr>
<td>grp_ProcE_loop1_procl_fu434</td>
<td>ProcE_loop1_procl1</td>
<td>25 345</td>
<td>25 345</td>
</tr>
<tr>
<td>grp_ff_param1_5_fu490</td>
<td>fff_param1_5</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_1_fu498</td>
<td>fff_param1_1</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_2_fu506</td>
<td>fff_param1_2</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_3_fu514</td>
<td>fff_param1_3</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_4_fu522</td>
<td>fff_param1_4</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_5_fu530</td>
<td>fff_param1_5</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_6_fu538</td>
<td>fff_param1_6</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ff_param1_7_fu546</td>
<td>fff_param1_7</td>
<td>103 103</td>
<td>103 103</td>
</tr>
<tr>
<td>grp_ProcE_loop2_procl_fu408</td>
<td>ProcE_loop2_procl1</td>
<td>25 660</td>
<td>25 660</td>
</tr>
</tbody>
</table>

**Figure 4.10: Latency and Interval results**
CHAPTER 4. IMPLEMENTATION

Figure 4.11: Resource usage

The detail of report shows that one transaction between array of BRAM and array of FIFO requires more than one cycle. After adding PIPELINE directives to inner most loops, the latency decreases around 2/3 while resource usage is nearly the same as before. One transaction can be done in one cycle after inner most pipeline. Then outer loops are also pipelined. Total latency decreases to 113 cycles while latency for loop1 and loop2 decrease to 66 and 111 respectively. Because of pipeline, resources usage becomes higher than before and DPS slices for multiplications increases to 68. However, only two data can be processed each cycle because of limited memory ports.

To make 8 transactions done in one cycle, ARRAY PARTITION directive is applied. But both latency and interval are prolonged a lot. The reason is that as the index of array is given via interface, HLS cannot decide the range of index. It will consider all possible situations and leads to conflict reading. The extreme example is when 8 data from same BRAM need to be sent to 8 FIFOs, it still takes at least 4 cycles to finish. This problem can be solved by adding conditional statement for the index. The report shows the latency for loop1 and loop2 reduce a lot after adding constraints for index.

However, intervals for loop1 and loop2 are 18 and 28 while the theoretical value of intervals should be 8 to transfer 64 pixels. Adding pipeline rewind to outer loop reduces interval of loop1 and loop2. But it is still not enough. After many attempts, interval becomes 8 when the conditional statements inside loops change to “if” “else” style instead of using all “if”. It seems HLS regards all “if” conditions can run in parallel. When use “if” “else”, HLS can schedule them separately without overlap.

Table 4.3 shows synthesis result. From this table, BRAM and DSP for multiplications are needed when loops are pipelined. The interval is 8 which meets the requirement.
CHAPTER 4. IMPLEMENTATION

<table>
<thead>
<tr>
<th>Optimization</th>
<th>dsp</th>
<th>bram</th>
<th>mlu</th>
<th>latency</th>
<th>interval</th>
<th>latency loop1</th>
<th>interval loop1</th>
<th>latency loop2</th>
<th>interval loop2</th>
</tr>
</thead>
<tbody>
<tr>
<td>UNROLL DATAFLOW</td>
<td>36</td>
<td>2</td>
<td>12</td>
<td>103-971</td>
<td>104-970</td>
<td>25-345</td>
<td>25-345</td>
<td>25-969</td>
<td>25-969</td>
</tr>
<tr>
<td>PIPELINE outer loop</td>
<td>92</td>
<td>2</td>
<td>68</td>
<td>113</td>
<td>112</td>
<td>66</td>
<td>66</td>
<td>111</td>
<td>111</td>
</tr>
<tr>
<td>ARRAY PARTITION</td>
<td>92</td>
<td>2</td>
<td>68</td>
<td>260</td>
<td>260</td>
<td>259</td>
<td>259</td>
<td>83</td>
<td>83</td>
</tr>
<tr>
<td>Add constraint</td>
<td>92</td>
<td>2</td>
<td>68</td>
<td>103</td>
<td>103</td>
<td>18</td>
<td>18</td>
<td>28</td>
<td>28</td>
</tr>
<tr>
<td>PIPELINE REWIND</td>
<td>92</td>
<td>2</td>
<td>68</td>
<td>103</td>
<td>104</td>
<td>17-18</td>
<td>16</td>
<td>28-29</td>
<td>16</td>
</tr>
<tr>
<td>Use if,else</td>
<td>92</td>
<td>8</td>
<td>68</td>
<td>103</td>
<td>104</td>
<td>8-9</td>
<td>8</td>
<td>19-20</td>
<td>8</td>
</tr>
</tbody>
</table>

Table 4.3: Synthesis result

4.5 Simulation result

This section will show the simulation result of the design. Vivado HLS supports simulation and validation at C-level. A test bench program with return value can be used for C simulation and C/RTL co-simulation. C simulation is just for verifying at C-level, which helps developer to ensure the function correctness of C-level program. C/RTL co-simulation uses the same test bench as C simulation, but it can verifies the correctness of RTL. The proper steps to verify the design are doing C simulation, synthesis and C/RTL co-simulation.

The test bench will not produce RTL, it is just for simulation. The test bench first reads pixel values from two images. As we mentioned above, 2D FFT for all real values will have hermitian redundancy. We eliminate the redundancy by load first image into real part of complex number and load second image into image part of complex number. A two dimension complex array stores two images. The array can be regarded as SDRAM part.

There are another two smaller arrays working like pingpong local memories. Consequently, data in big array is passed to smaller arrays. Loops are setting to call top-function. The number of iterations is equal to the number of times to do 1D FFTs. This part is not contained in top-function because it will block streaming input. FFT cores in loops of top-function cannot be pipelined and next frame of data should wait for it. FIFO in dataflow region will send ap_ready signal once it is empty. But adding an outer loop for FFT core and FIFO would block ap_ready signal. Moreover, FIFO and FFT core in sub-function also have this issue. HLS instantiate loops and sub-function as a block that has its own “ready” signal.

Fig.4.12 is the simulation waveform of FFT in sub function. Fig.4.13 is the simulation waveform of FFT in top function. It shows that when FFT is in sub-function, ap_ready is active after FFT results are all output. Although input FIFO is empty and sends a ap_ready signal to outer block, no data can be in without ap_ready signal of top-function. Hence, when FIFO and FFT are at the top level, ap_ready signal can be sent out directly and next frame of data could be continuously input.
If the input is not streaming, another reason could be that loop is pipelined without rewind. Data should wait until loop interval finished, therefore there should be no idle time between reading operations.
After all FFTs are finished, IFFT operations follow subsequently. Test bench will calculate SNR and RMS of two images before and after FFT+IFFT. Table 4.4 is the simulation result of the design with different image sizes and optimizations. It shows that when processing 128*128 image, throughput decreases. The reason is that the minimum interval is equal to the size of the sub-block in data exchange step. If the size of the sub-block is different in local 2D DFT step, input is not streaming. If their sizes are the same, processing larger image can achieve higher throughput.

<table>
<thead>
<tr>
<th>size</th>
<th>cycle</th>
<th>stream</th>
<th>array</th>
<th>constraint</th>
<th>pipeline</th>
<th>ifft</th>
<th>throughput(Bytes/clk)</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>28434</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>no</td>
<td>2.3</td>
</tr>
<tr>
<td>64</td>
<td>8852</td>
<td>no</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>14.81</td>
</tr>
<tr>
<td>64</td>
<td>4713</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>27.81</td>
</tr>
<tr>
<td>128</td>
<td>26292</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>19.94</td>
</tr>
<tr>
<td>256</td>
<td>69749</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>yes</td>
<td>30.07</td>
</tr>
</tbody>
</table>

Table 4.4: Simulation result

Table 4.5 is the accurate result of different image sizes. The accuracy depends on not only the image size, but also the content of the image.

<table>
<thead>
<tr>
<th>size</th>
<th>SNR</th>
<th>RMS</th>
<th>overflow</th>
</tr>
</thead>
<tbody>
<tr>
<td>64</td>
<td>53.05</td>
<td>0.0022</td>
<td>no</td>
</tr>
<tr>
<td>128</td>
<td>43.55</td>
<td>0.014</td>
<td>yes</td>
</tr>
<tr>
<td>256</td>
<td>37.93</td>
<td>0.034</td>
<td>yes</td>
</tr>
</tbody>
</table>

Table 4.5: Accuracy result
Chapter 5

Conclusions

In this report, different 2D FFT algorithms are analysed and 2D decomposition algorithm is selected to be implemented as it is a bandwidth intensive algorithm. Compared to other algorithms, it requires less on-chip memory. Paper from Arizona State University [17] proves that this algorithm can achieve around 3000MB/s bandwidth when processing 2048 × 2048 random images. This is close to the peak bandwidth of the SDRAM they used. In our project, we assume the peak bandwidth of the SDRAM is also 3200MB/s and try to utilize bandwidth.

The algorithm is implemented by Vivado HLS and programmed by C language. HLS provides fixed point data type, which helps us to transform float point data to fixed point.

After optimization, the simulation result shows the maximum throughput of 2D decomposition algorithm can be around 30Bytes/clk when frequency is 100Mhz and the image size is 256×256. It proves that Vivado HLS is possible to make input streaming and achieve high throughput like behavior on hardware.

The Hermitian redundancy has been removed, so that two images can be processed at the same time. The computation time for each image reduces nearly half after removing Hermitian redundancy. Moreover, IFFT has also been achieved. The permutation function is achieved in test bench, namely Host PC.

Vivado HLS is a powerful tool to transform C-level language to HDL, enabling software developers to implement their algorithms on the FPGA. Simple design which does not require high throughput is not hard to be achieved in Vivado HLS. Even developers without any hardware knowledge can complete their design by learning tutorials.

However, according to the experiment results, developers should have basic hardware thinking if they want to implement complicated design. As this project aims to utilize bandwidth and improve throughput, problem such as how to configure memory interface, how FIFO and FFT IP core works, should be figured out. Each HLS project only contains one top-function. Functions inside top-function are sub-functions. FFTs in sub-function have poor performance as the input cannot be streaming. Thus there is no sub-function in our design. The experiment result shows that we cannot get ideal throughput if the entire project is implemented in one top-function.

Pingpong BRAM and memory interface have not been implemented successfully on the FPGA because they are banned by rules when using optimization directives. But it is feasible to combine multiple IP core that produced by Vivado HLS to complete the design. It can be done in Vivado which is also from Xilinx but only supports HDL.

On the other hand, the program has not been run on real FPGA. Vivado HLS can export IP core that is able to be used with the Vivado Design Suite, System Generator and other Xilinx ISE, but it cannot produce bit stream. The design can be verified by simulation but not on real FPGA. Therefore next step is to do combination in Vivado design suite and make bit stream file.

In terms of the image size, samples we used are small images because we aims to verify the function correctness and improve the throughput of the design. Simulation for big images will cost lots of running time. Big images will be processed once the entire design is completed in Vivado design suite. The generality of the code still needs to be verified.
CHAPTER 5. CONCLUSIONS

Size | Bandwidth(MB/s) | BRAM18k | BRAM18k | PE | DSP | Device
-----|----------------|---------|---------|----|-----|---------
1024 | 3200           | 32      | 40      | 8  | 116 | Zedboard
1024 | 6400           | 64      | 84      | 16 | 228 | other
2048 | 3200           | 128     | 64      | 8  | 116 | Zedboard
2048 | 6400           | 256     | 132     | 16 | 228 | other
4096 | 3200           | 128     | 80      | 8  | 116 | Zedboard
4096 | 6400           | 256     | 164     | 16 | 228 | other
8192 | 3200           | 512     | 112     | 8  | 116 | other
8192 | 6400           | 1024    | 228     | 16 | 228 | other
16384| 3200           | 512     | 176     | 8  | 116 | other
16384| 6400           | 1024    | 356     | 16 | 228 | other

Table 5.1: Resources for different bandwidth

On the other hand, the precision of image after 2D FFT and 2D IFFT doesn’t meet requirement of SNR, we need to find out the reason.

As only ZedBoard has been used for synthesis and simulation, more experiments need to be carried out to see whether the program can be demonstrated on a specific FPGA and SDRAM with higher bandwidth or not. If $16384 \times 16384$ images are going to be processed with higher bandwidth, the resource usage would significantly increase. Table 5.1 is the resource usage comparison between different FFT sizes and different bandwidths. When bandwidth is 6400MB/s, Zedboard is not applicable to the design. Because the number of PE increases to 16, local memory should be able to store $16 \times p \times q$ sub-blocks. When bandwidth is 3200MB/s, Zedboard is not applicable to images with size of $8192 \times 8192$ or bigger. Hence, a high end FPGA such as Virtex 7 VC707 that has thousands of BRAM blocks can be a substitute.
Bibliography

[1] Ren Chen and Viktor K Prasanna. Energy-efficient architecture for stride permutation on streaming data. In 2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig), pages 1–7. IEEE, 2013. 27


Mapping Large 2D FFTs onto FPGAs using High Level Synthesis 45
