Multidimensional DFT IP Generators for FPGA Platforms

by

Chi-Li Yu

A Dissertation Presented in Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy

Approved November 2011 by the
Graduate Supervisory Committee:

Chaitali Chakrabarti, Chair
Antonia Papandreou-Suppappola
Lina Karam
Yu Cao

ARIZONA STATE UNIVERSITY

May 2012

ABSTRACT

Multidimensional (MD) discrete Fourier transform (DFT) is a key kernel algorithm in many signal processing applications, such as radar imaging and medical imaging. Traditionally, a two-dimensional (2-D) DFT is computed using Row-Column (RC) decomposition, where one-dimensional (1-D) DFTs are computed along the rows followed by 1-D DFTs along the columns. However, architectures based on RC decomposition are not efficient for large input size data which have to be stored in external memories based Synchronous Dynamic RAM (SDRAM).

In this dissertation, first an efficient architecture to implement 2-D DFT for large-sized input data is proposed. This architecture achieves very high throughput by exploiting the inherent parallelism due to a novel 2-D decomposition and by utilizing the row-wise burst access pattern of the SDRAM external memory. In addition, an automatic IP generator is provided for mapping this architecture onto a reconfigurable platform of Xilinx Virtex-5 devices. For a 2048×2048 input size, the proposed architecture is 1.96 times faster than RC decomposition based implementation under the same memory constraints, and also outperforms other existing implementations.

While the proposed 2-D DFT IP can achieve high performance, its output is bit-reversed. For systems where the output is required to be in natural order, use of this DFT IP would result in timing overhead. To solve this problem, a new bandwidth-efficient MD DFT IP that is transpose-free and produces outputs in natural order is proposed. It is based on a novel decomposition algorithm that takes into account the output order, FPGA resources, and the characteristics of off-chip memory access. An IP generator is designed and integrated into an in-house FPGA development platform, AlgoFLEX, for easy verification and fast integration. The corresponding 2-D and 3-D DFT architectures are ported onto the BEE3 board and their performance measured and analyzed. The results shows that the architecture can maintain the maximum memory bandwidth throughout the whole procedure while avoiding matrix transpose operations used in most other MD DFT implementations. The proposed architecture has also been ported onto the Xilinx ML605 board. When clocked at 100 MHz, 2048×2048 images with complex single-precision can be processed in less than 27 ms.

Finally, transpose-free imaging flows for range-Doppler algorithm (RDA) and chirp-scaling algorithm (CSA) in SAR imaging are proposed. The corresponding implementations take advantage of the memory access patterns designed for the MD DFT IP and have superior timing performance. The RDA and CSA flows are mapped onto a unified architecture which is implemented on an FPGA platform. When clocked at 100MHz, the RDA and CSA computations with data size 4096×4096 can be completed in 323ms and 162ms, respectively. This implementation outperforms existing SAR image accelerators based on FPGA and GPU.

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my advisor, Dr. Chaitali Chakrabarti for her guidance, support, and encouragement. She gave me the freedom to explore and provided advice when my steps faltered. Her logical way of thinking and wide knowledge have been invaluable to my academic studies. Her technical and editorial advice on writing was essential for the completion of the published papers as well as this dissertation. I have been feeling amazingly fortunate that I could work with her in the past years. This experience has been one that I will always cherish.

My sincere thanks also go to Dr. Vijaykrishnan Narayanan of Pennsylvania State University, Dr. Xiaobai Sun and Dr. Nikos Pitsianis of Duke University, and my committee, Dr. Antonia Papandreou-Suppappola, Dr. Lina Karam, and Dr. Yu Cao, for their insightful comments and suggestions on my research.

I am indebted to my research colleagues, Lanping Deng of Arizona State University, Jungsub Kim, Srinidhi Kestur, Kevin Irick, Sungho Park, Ahmed Al Maashri, and Michael DeBole of Pennsylvania State University, for their stimulating discussions and continuous assistance. They made all difficult implementations and demonstrations possible with their tremendous effort. I am really thankful that I could work with these talented researchers. I also wish to thank my fellow members of Low Power Systems Lab, Qi Qi, Tao Liu, Lifeng Miao, Qian Xu, Chengen Yang, Ming Yang, Samatha Gummalla, and Rupa Mahadevan, for providing a fun environment in which to discuss, learn, and grow.

Most importantly, none of this would have been achieved without support from my family. I am grateful to my dear parents. It is thanks to my father, Ke-Chin Yu, that he was the one who encouraged me to pursue my PhD degree. For my mother, Kuang-Hui Chiang, I deeply appreciate her constant support, emotionally, morally, and of course, financially. My wife, Chiu-Hui Fan, has always been my joy and my pillar. I would like to thank her for her company, understanding, and encouragement during the past years. To my beloved family, I dedicate this dissertation.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Figure Page

Chapter 1

Introduction

Multidimensional digital signal processing (MD DSP) is an important research area with applications spanning image coding, medical imaging, remote sensing, seismology, surveillance, computer vision, video compression, and many more. Compared to one-dimensional (1-D) DSP systems, an MD DSP system needs to handle a much larger amount of data. For example, radar images could be as large as $4,096 \times 4,096$, which is equivalent to 16 mega pixels or 64 Mbytes of data in single-precision format. Many of the MD DSP applications also demand real-time computation. For instance, real time computation of $4,096 \times 4,096$ 2-D DFT in radar imaging requires about 6×10^{10} floating-point operations per second, and is quite a challenge.

Fortunately, the steadily increasing density of transistors on a chip has resulted in dramatic improvements in the performance of the computing devices. As shown in Table 1.1, the Intel i7-975 CPU has a peak performance of 55.36 GFLOPs with its 6 cores running at 3.46 GHz. This is 49x faster than the Intel Pentium III processor from 10 years ago. Thus today's processors have the capability to implement many real-time MD DSP systems, provided the I/O bandwidth is not a constraint.

MD DSP systems also operate on huge amounts of data, which have to be stored into external memories. Memory capacity has also significantly increased over the past decade, as shown in Table 1.1. Today there are 8GBytes per DIMM (double in-line memory module), which is sufficient

Table 1.1: Progress of CPU and memory in the past decade.

	2000	2010	Improvement
CPU performance(1)	1.13 GFLOPS (Intel Pentium-III 1.13 GHz)	55.36 GFLOPS (Intel i7-975)	49x faster
Memory capacity(2)	128 MB (DDR-SDRAM pc133)	8 GB (DDR3-SDRAM pc3-17000)	64x larger
Memory performance(2)	1.066 GB/s (DDR-SDRAM pc133)	17.066 GB/s (DDR3-SDRAM pc3-17000)	16x faster

(1): Theoretical peak performance without I/O constraints.
(2): Per dual in-line memory module (DIMM).

to store multiple high-resolution images or video clips. In short, with contemporary technologies, there is sufficient computing power and memory capacity to handle MD DSP computations.

However, memory performance, defined as data transfer rate, has not improved at the same rate as memory capacity or CPU performance. As shown in Table 1.1, the memory speed has only improved by 16 times, while the CPU's peak performance has improved 49 times and memory capacity has increased 64 times. Thus, the data transfer between a computing device and the external memory is still the bottleneck of a system. This was predicted in [1], and has only gotten worse. While memory bottleneck is true for general systems, it is exacerbated in MD DSP systems since huge amounts of data need to be accessed to and from the external memory multiple times.

In this research, we focus on a typical MD DSP algorithm, namely, multidimensional Discrete Fourier Transform (MD DFT). Its implementation is very challenging, because it is not only computation-demanding but also memory bandwidth-demanding. Not only do we have to access large amounts of data, they have to be accessed along different dimensions! In this dissertation, we concentrate on efficient implementations of MD DFT and study how to overcome the memory bandwidth constraints.

1.1 MD DFT Implementations

MD DFT is widely used in signal processing and scientific computing applications, more specifically, it is used in imaging applications which need operations in frequency domain, such as image watermarking, finger print recognition, synthetic aperture radar (SAR) processing and medical imaging. The image sizes in many of these applications have become larger over the years. In SAR imaging, for instance, the image size could be as large as 4,096×4,096 [2], and in medical imaging, the data size could be $512 \times 512 \times 384$ [3].

Existing MD DFT implementations include software which are optimized for general-purpose CPU, such as FFTW [4, 5], Spiral [6, 7], Intel MKL [8] and IPP [9], or even cluster computers [10, 11]. Software solutions are very flexible and can be ported to users' applications easily and quickly. However, these platforms usually consume power that is too high for embedded applications.

ASIC implementations for 1-D DFT [12, 13, 14, 15, 16, 17], and MD DFT [18, 19] have been proposed over the years. They exploit the high regularity and parallelism of the DFT algorithm and are quite efficient. While DFT ASICs offer high performance while consuming minimal power, the manufacturing cost of these chips is quite high, and once a chip is manufactured, its functionality and performance cannot be changed anymore.

1.2 FPGA-based MD DFT Accelerators

Field programmable gate array (FPGA) has become an attractive alternative to ASIC, because it offers high flexibility. Large amount of on-chip resources, such as logic slices, multipliers, and RAMs, make contemporary FPGAs a great candidate for DSP accelerator. Besides, FPGA vendors also provide versatile intellectual properties (IPs) and comprehesive hardware/software support, which help the user to shorten design cycle significantly. This is why many MD DFT implementations based on FPGA has been proposed [20, 21, 22, 23, 24, 25]. In this dissertation, we concentrate on FPGA-based architectures.

Many of the existing FPGA solutions for MD DFT are based on Row-Column (RC) decomposition [19, 21, 22], where the 2-D DFT is computed by successively applying 1-D DFT along rows and then along columns. This works fine for Static RAM (SRAM) based designs [21, 22], where data access along rows and along columns have the same cost. However, for systems with dynamic memory, e.g. synchronous dynamic RAM (SDRAM) which is typically adopted for its large storage density, RC decomposition has low performance. This is because SDRAM only favors burst access, and thus while data stored in consecutive locations in memory can be retrieved very efficiently, accessing data along columns is a lot more expensive. To avoid memory access with large strides, the transpose operation is used in [19] to re-align the column data into contiguous addresses. While this enables a long burst size to be maintained, the transpose operation take additional time. When the dimension is higher, e.g. in 3-D DFT, the transpose operation becomes harder to implement because of the limited local memory on the FPGA.

Finally, most hardware solutions are not as versatile as software solutions. They cannot support 2-D and 3-D DFTs in one package, and some [19, 21] are even fixed to some specific image sizes.

1.3 Contributions

In this research, our goal is to (1) optimize the performance of the MD DFT processors under the constraints of memory bandwidth and hardware resources; (2) provide flexible MD DFT IPs that are easy to use in other applications and can support various image sizes. Our study focuses on the following three tasks.

- **I. FPGA Architecture for 2-D Discrete Fourier Transform Based on 2-D Decomposition for Large-sized Data** (Chapter 3)

  We propose an efficient architecture to implement 2-D DFT for large-sized input data based on a novel 2-D decomposition algorithm. This architecture achieves very high throughput by exploiting the inherent parallelism due to the algorithm decomposition and by utilizing the row-wise burst access pattern of the external memory. A high throughput memory interface has been designed to enable maximum utilization of the memory bandwidth. In addition, an automatic system generator is provided for mapping this architecture onto a reconfigurable platform of Xilinx Virtex-5 devices. For a 2K$\times$2K input size, the proposed architecture is 1.96 times faster than RC decomposition based implementation under the same memory constraints, and also outperforms other existing implementations.

  e

- **II. Multidimensional DFT IP Generators for FPGA Platforms** (Chapter 4)

  We propose an MD-DFT intellectual property (IP) generator and a bandwidth-efficient MD DFT IP for high performance implementations of 2-D and 3-D DFT on FPGA platforms. The proposed architecture is generated automatically and is based on a decomposition algorithm that takes into account FPGA resources and the characteristics of off-chip memory access, namely, the burst access pattern of the Synchronous Dynamic RAM (SDRAM). The IP generator has been integrated into an in-house FPGA development platform, AlgoFLEX,

for easy verification and fast integration. The corresponding 2-D and 3-D DFT architectures have been ported onto the BEE3 board and their performance measured and analyzed. The results shows that the architecture can maintain the maximum memory bandwidth throughout the whole procedure while avoiding matrix transpose operations used in most other MD DFT implementations. The proposed architecture is also ported onto the high-end ML605 board. The simulation results show that 2K×2K 2-D images can be processed in less than 27ms, and 128x128x128 3-D DFT can be finished within 22ms.

- **III. Transpose-free SAR Imaging on FPGA Platform** (Chapter 5)

  We propose transpose-free implementations of Synthetic Aperture Radar (SAR) image algorithms, such as range-Doppler algorithm (RDA) and chirp scaling algorithm (CSA). In both algorithms, data needs to be transformed between space and frequency domains back and forth, and so in traditional implementations, multiple transpose operations were required thereby degrading timing performance. The proposed implementation utilizes the memory access patterns derived from the MD DFT IP and avoid the transpose operations. The proposed transpose-free flows for RDA and CSA are mapped to a unified architecture that supports both RDA and CSA and ported onto the ML605 FPGA board. For 4K×4K data size, the RDA computation can be completed in 323ms, which is 1.2x faster than the most advanced GPU-based solution. For CSA with the same data size, the computation can be finished in 162ms, which is 6.6x faster than another FPGA-based accelerator.

The rest of this dissertation is organized as follows. In Chapter 2, we review state-of-the-art 1-D and MD DFT implementations. Then, the new high-performance 2-D DFT for large sized images is described in Chapter 3. Chapter 4 presents the flexible DFT IP generator for 2-D and 3-D DFT. In Chapter 5, we describe the in-pending work on design of a SAR imaging processor based on the MD DFT IP. The conclusion is given in Chapter 6.

Chapter 2

Background on DFT Algorithm and Its Implementations

In this chapter, we first review 1-D DFT algorithm and its hardware implementations. Next we present a survey on existing MD DFT implementations.

2.1 Fast Fourier Transform

Discrete Fourier Transform (DFT) of an N-point complex sequence $x(n)$ is defined as:

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{nk}, k = 0, 1, \ldots, N-1, \tag{2.1}$$

where $W_N = e^{-j2\pi/N}$ and $n = 0, 1, \ldots, N-1$. The computation complexity of an N-point DFT is $O(N^2)$. Fast Fourier Transform (FFT) algorithm reduces the computation complexity to $O(N \log_2 N)$ [26] by recursively decomposing the even and odd frequency components as follows:

$$X_{2k} = \sum_{n=0}^{N/2-1} (x_n + x_{n+N/2}) \cdot W_{N/2}^{nk}, \tag{2.2}$$

$$X_{2k+1} = \sum_{n=0}^{N/2-1} (x_n - x_{n+N/2}) \cdot W_{N/2}^{nk} \cdot W_{N/2}^{n}. \tag{2.3}$$

The data flow graph of an 8-point FFT is shown in Fig. 2.1.

Hardware implementation of FFT is a well-studied topic. A few of the commonly used architectures are described as follows.

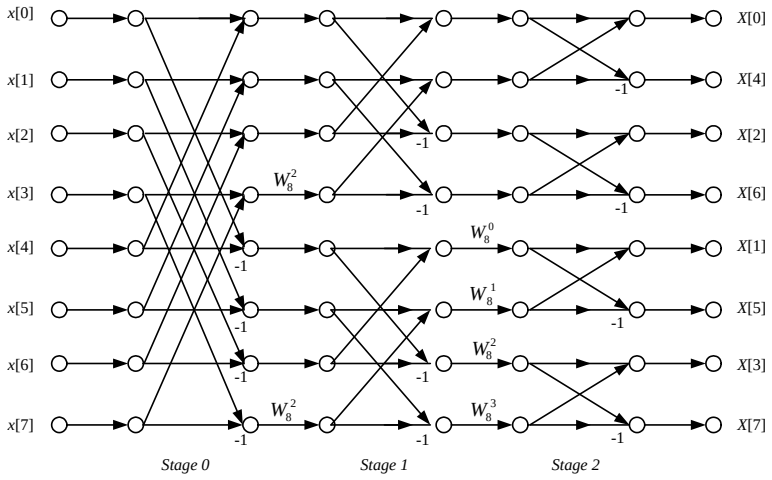


Figure 2.1: Data flow graph of 8-point FFT (decimation-in-frequency).

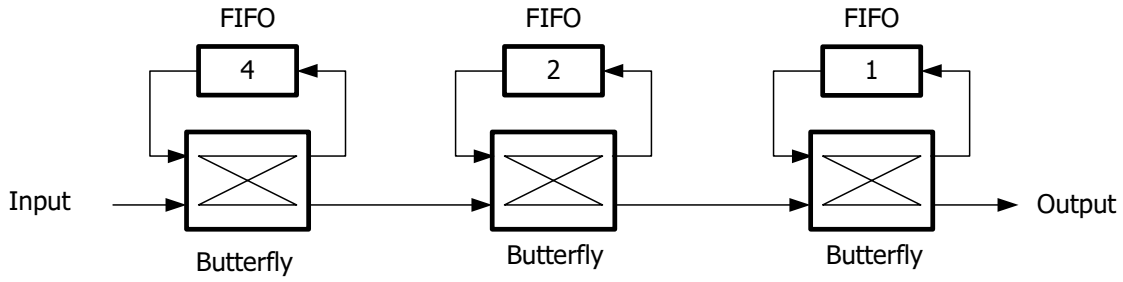


Figure 2.2: Architecture of Xilinx pipelined FFT IP core [27].

2.1.1 Pipelined Architectures

The pipelined FFT architecture [27] [28] [16] is a serial-in-serial-out(SISO) architecture. Fig. 2.2 shows a typical example of pipelined FFT. It is obtained by a straightforward mapping of the data flow graph onto $\log_2 N$ hardware units, where each hardware unit is assigned to one stage of the FFT flow graph. The main advantage of this architecture is its high throughput rate. Input sequences can be fed in continuously. Besides, its control mechanism is also very simple. The internal memories are simple FIFOs without any complex memory controllers. The drawback of the pipelined architecture is that it needs an extra bit-reverse operation, and an additional buffer is required for rearranging the data. Besides, the maximum FFT size it can compute is limited the number of butterfly units.

The pipelined FFT IP core [27] provided by Xilinx is based on this architecture. Because of its high performance, we adopt it in our MD DFT implementations.

2.1.2 Memory-based Architectures

Memory-based architecture [29] [30] [31] consists of a memory unit and a computational core, which contains one or multiple butterfly units, as illustrated in Fig. 2.3. The memory unit is used not only as a storage for intermediate results but also as an input/output buffer. By simply modifying the memory addressing, memory-based architectures can prevent the extra operation of bit-reversal.

When the number of butterfly units increases, however, the memory structure and addressing becomes very complex. This is because multiple butterfly units will need to access data at the same time, and conflicts need to be avoided. Xilinx [27] provides memory-based FFT with only

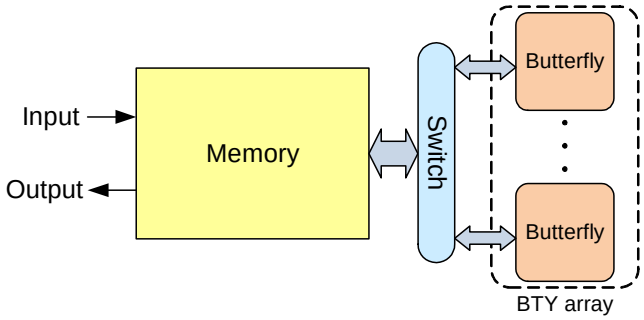


Figure 2.3: Architecture of Memory-based FFT.

one butterfly unit. It costs less in terms of hardware resources but it is much slower compared to the pipelined counterpart.

2.1.3 Automatic IP Generators

More recently, automatic tools to rapidly generate have been developed optimized FFT implementations for FPGAs. The method in [32] performs design space exploration on different number of butterfly units and sub-FFTs, and estimates the performance and hardware cost of the different configurations. The FFT compiler in [33] also provides a mechanism for folding FFT computations onto sub-FFT computation modules. Under user-defined performance requirements, these automatic tools are able to generate the most cost- or power-efficient FFT architectures. It is worthwhile to note that [32] [33] have been ported onto FPGAs and achieve high performance competitive to some commercial products such as [27].

2.2 Multidimensional DFT

A 2-D DFT of $N_1 \times N_2$ samples, $x(i_1, i_2)$, is defined by,

$$Y(k_1,k_2) = \sum_{i_1=0}^{N_1-1} \sum_{i_2=0}^{N_2-1} x(i_1,i_2) \cdot W_{N_1}^{k_1 i_1} \cdot W_{N_2}^{k_2 i_2}, \tag{2.4}$$

where $k_1 \in [0, N_1 - 1]$ and $k_2 \in [0, N_2 - 1]$.

The Row-Column (RC) decomposition algorithm decomposes a 2-D DFT into a series of multiple 1-D DFTs:

$$Row\ DFT: \quad \tilde{X}(i_1,k_2) = \sum_{i_2=0}^{N_2-1} x(i_1,i_2) \cdot W_{N_2}^{k_2 i_2}, \tag{2.5}$$

and

$$Column\ DFT: \quad Y(k_1,k_2) = \sum_{i_1=0}^{N_1-1} \tilde{X}(i_1,k_2) \cdot W_{N_1}^{k_1 i_1}. \tag{2.6}$$

With RC decomposition, a 2-D DFT on $N_1 \times N_2$ data can be computed by first performing N_2 row-wise 1-D DFTs, and then N_1 column-wise 1-D DFTs. It has a complexity of $O(N_1 N_2 (N_1 + N_2))$. If the 1-D DFTs can be replaced by FFTs, the complexity of 2-D FFT is $O(N_1 N_2 \log_2(N_1 N_2))$. For hardware implementation, one can simply adopt any 1-D FFT architectures mentioned in Section 2.1 to construct the 2-D DFT based on the RC decomposition. Furthermore, the decomposition of 2-D DFT can be easily extended to DFT with higher dimensionality, such as 3-D DFT.

2.3 Existing MD DFT Implementations

Several FPGA-based multidimensional DFT solutions have been proposed in the literature. In 2001, Dillon Engineering [21] delivered a high performance image processing system which includes two Xilinx Virtex-II FPGAs and can achieve 120 frames per second (fps) at a resolution of 2,048×2,048. However, the design is based on RC decomposition, which needs large-sized Static RAMs (SRAMs) as intermediate memory for matrix transpose operations. SRAM based designs were also used in [34] and [35]. In [34] the authors used two Xilinx Virtex-2000E FPGAs and four banks of on-board SRAMS to construct a 1,024×1,024 2-D FFT with a frame rate of 13 fps. Although its performance and size of 2-D FFT are lower than [21], the cost of the design is also much lower due to the cheaper FPGAs and smaller SRAMs. A 0.35um ASIC design with on-chip SRAM was presented in [35]. It operates at 133MHz and has a frame rate of 42 fps at a resolution of 512×512. Although the on-chip memory reduces the power consumption compared to external memory, its smaller capacity also limits the size of 2-D DFT.

To deal with large data sizes, Lenart et al. [19] chose SDRAM instead of SRAM. However, the three-dimensional organization of SDRAM devices (banks, rows, and columns) results in non-uniform access. For instance, data inside a row can be transferred in a burst manner, while accessing data in different rows can cause a much longer latency. This characteristic of SDRAM causes significant performance loss while doing column-wise FFT operations. To avoid column memory access, a transpose operation is required to realign the column data into the rowsTo implementation the transpose, the 2-D data need to be partitioned into small sub-blocks. The sub-blocks are read

sequentially, transposed on the chip, and written back to the external memory in proper locations. The operation can be represented by the following equation:

$$\begin{pmatrix} A_{00} & A_{01} & A_{02} \\ A_{10} & A_{11} & A_{12} \end{pmatrix}^T = \begin{pmatrix} A_{00}^T & A_{10}^T \\ A_{01}^T & A_{11}^T \\ A_{02}^T & A_{12}^T \end{pmatrix}. \tag{2.7}$$

With the transpose unit, the overall latency can be reduce by 3 times. The 2,048×2,048 2-D DFT is ported onto a Virtex-1000E FPGA, which runs at 24MHz and achieves 1.5 fps frame rate. The design has also been synthesized in 0.13 μm CMOS process. It is able to operate at 250MHz and supports a frame rate of 20 fps. In short, this design successfully solves the non-uniform access time of SDRAM. However, the transpose is an extra operation which is not overlapped with the DFT computations and the overall performance is weaker than that of [21].

There are other hardware solutions such as [20, 24, 36] that are not based on RC decomposition. Here the multi-stage DFT flow graph is folded into one or more butterfly units. These approaches are efficient when the data size is small and can fit in the on-chip memory. For large data sizes, these architectures will have to optimize data access from external memory to sustain high performance.

In Table 2.1, we list the existing 2-D DFT implementations. First, we see that ASIC/FPGA solutions can easily outperform the pure software solutions with standard CPUs. For instance, a 3GHz dual-core Pentium4 can only deliver $2,048 \times 2,048$ images at 0.8 fps [19]. Since 2-D DFT is an algorithm with high parallelism, hardware with multiple customized processing elements, such as ASIC or FPGA, is a more suitable architectural platform. Secondly, FPGA is a better candidate than ASIC, due to lower development cost and competitive performance. In fact, the best performer [21] is an FPGA-based solution as shown in Table 2.1.

To sum up, most 2-D DFT architectures are based on RC decomposition. They still require either an extra transpose unit or expensive SRAMs to overcome the problem caused by column-wise memory access. In the rest of this dissertation, we propose new 2-D decomposition algorithms which can reduce the size of both row- and column-wise 1-D FFTs and reduce the penalty of column-wise memory access. The matrix transpose is no longer required, and the proposed SDRAM-based architecture can compete with the more expensive SRAM-based solutions.

Table 2.1: Comparison of the existing MD DFT processors

Technology	Memory	Clock Frequency	Frame Rate*(fps)	Year
Intel Pentium4 Dual Core [19]	Dual-Bank SDRAM	3 GHz	0.8	2008
ASIC, 0.35μm [35]	Single-Bank SRAM	133 MHz	2.6	2003
ASIC, Eonic PowerFFT [18]	Quad-Bank SDRAM/SRAM	128 MHz	11.9	2002
ASIC, 0.13μm [19]	Dual-Bank SDRAM	250 MHz	20.0	2008
FPGA, Virtex-1000E [19]	Dual-Bank SDRAM	24 MHz	1.5	2008
FPGA, Virtex-2000E [22]	Quad-Bank SRAM	35 MHz	2.0	2005
FPGA, Virtex-II-6000$\times$2 [21]	Dual-Bank SRAM	125 MHz	120.0	2001
FPGA, Virtex-4-LX40 [37]	Quad-Bank SRAM	N/A	30.0	2007

(*: Frame rate is normalized to $2,048 \times 2,048$ image size.)

Chapter 3

FPGA Architecture for 2-D Discrete Fourier Transform based on 2-D Decomposition for Large-sized Data

3.1 Introduction

In this chapter, we describe FPGA architectures for 2-D DFT that are targeted for large data sizes. The proposed algorithm partitions the original data into a mesh of sub-blocks, performs butterfly type operations between sub-blocks and then computes local 2-D DFT on each of the sub-blocks. The size of the sub-blocks is a function of the available FPGA resources and is determined automatically. The experimental results demonstrate that our architecture based on the 2-D decomposition algorithm achieves better performance than optimized architectures based on Row-Column (RC) decomposition.

The rest of the chapter is organized as follows. Section 3.2 briefly introduces 1-D and 2-D DFT and derives the proposed 2-D decomposition algorithm. Section 3.3 describes in detail our novel FPGA architecture for 2-D DFT. Section 3.4 describes the automatic system generator. Various configurations of our architecture are evaluated in Section 3.5, and concluding remarks are given in Section 3.6.

3.2 Decomposition Algorithms for DFT

In this section, the decomposition of 1-D DFT is described in Section 3.2.1, followed by the 2-D decomposition algorithm for 2-D DFT in Section 3.2.2 and the functional components of the 2-D DFT in Section 3.2.3.

3.2.1 Decomposition of 1-D DFT

A 1-D DFT of length N can be decomposed and computed by a series of smaller transforms and permutations. We first represent DFT in the matrix-vector multiplication form as

$$[X_0\, X_1 \ldots\, X_{N-1}]^T = F_N \cdot [\, x_0\, x_1 \ldots\, x_{N-1}]^T, \tag{3.1}$$

where F_N is the twiddle factor matrix.

The decomposition of 1-D DFT is essentially representation of F_N as a product of sparse matrices and is described as follows [38], [39], [40].

$$F_N = P_{N,p}(I_p \otimes F_m)\tilde{D}_N(F_p \otimes I_m), \tag{3.2}$$

where $N = p \cdot m$, where p and m are both integers. I_m is an $m \times m$ identity matrix, $\tilde{D}_N$ is a diagonal matrix of twiddle factors, and $\otimes$ is the Kronecker or tensor product and can be expressed as

$$\tilde{D}_N(j,j) = W_N^{(j \bmod m)\cdot \lfloor j/m \rfloor} \ \ for\ j = 0,1 \ldots N-1, \tag{3.3}$$

$$A_n \otimes B_m = [a_{k,l}B_m]_{0 \le k,l < n} \ \ for\ \ A_n = [a_{k,l}]_{0 \le k,l < n}. \tag{3.4}$$

Finally, $P_{N,p}$ denotes permutation with stride p.

The traditional radix-2 FFT can be considered a specific case of recursive decomposition with factor $p = 2$.

3.2.2 2-D decomposition algorithm

The general form of 2-D DFT is described in matrix form as follows:

$$Y = F_M \cdot X \cdot F_N^T = F_M \cdot X \cdot F_N, \tag{3.5}$$

where input X and output Y are of size $M \times N$; F_M and F_N are DFT matrices which are symmetric.

The expression $(F_M \cdot X)$ is traditionally calculated by applying an M-point DFT for each column of X. As described in Section 3.2.1, an M-point DFT can be replaced by the sparse matrix product form as depicted in Eq. (3.2). Hence, by partitioning a column of size M into p sub-blocks, the expression $(F_M \cdot X)$ can be written as follows:

$$F_M \cdot X = P_{M,p} \cdot (I_p \otimes F_{M/p})\tilde{D}_M(F_p \otimes I_{M/p}) \cdot X, \tag{3.6}$$

where the permutation $P_{M,p}$ term in Eq. (3.2) is taken into account in the final stage.

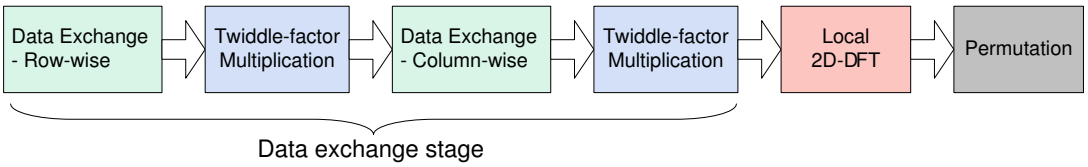


Figure 3.1: The functional flow graph of 2-D DFT.

Similarly, the expression $(X \cdot F_N)$ in Eq. (4.1) can be written as follows:

$$X \cdot F_N = X \cdot (F_q \otimes I_{N/q}) \tilde{D}_N (I_q \otimes F_{N/q}) \cdot P_{N,q}. \tag{3.7}$$

Extending such a partitioning to both row-wise and column-wise elements, the 2-D decomposition of Eq. (4.1) on a $p \times q$ mesh is written as follows:

$$\tilde{Y} = (I_p \otimes F_{M/p}) \tilde{D}_M (F_p \otimes I_{M/p}) \cdot X \cdot (F_q \otimes I_{N/q}) \tilde{D}_N (I_q \otimes F_{N/q}). \tag{3.8}$$

Y is obtained by applying a bit-reverse permutation (P) on $\tilde{Y}$ of Eq. (3.8).

3.2.3 Functional components of the 2-D decomposition algorithm

Eq. (3.9) pictorially demonstrates the sequence of operations involved in the computation of the decomposed 2-D DFT.

$$Y = P((I_p \otimes F_{M/p}) \tilde{D}_M (F_p \otimes I_{M/p}) \cdot \underbrace{X \cdot (F_q \otimes I_{N/q}) \tilde{D}_N}_{\text{Step 1}} (I_q \otimes F_{N/q})). \tag{3.9}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{Step 2}}$$
$$\underbrace{\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}_{\text{Step 3}}$$

Input X of size $M \times N$, is partitioned into a $p \times q$ mesh where each sub-block in the mesh is of size $M/p \times N/q$. The four main steps are described below Fig. 3.1 presents the functional flow of the proposed 2-D decomposition algorithm.

Step 1. Row-wise data exchange with twiddle-factor multiplication:

There are q sub-blocks in each row, and each element inside one sub-block has to do data exchange with the corresponding elements in the other $(q-1)$ sub-blocks. This data exchange (DX) operation can be implemented as a q-point 1-D FFT followed by a twiddle-factor multiplication (with $\tilde{D}_N$). Since there are $M/p \cdot N/q$ elements in each sub-block, there are $(MN/pq)(q \cdot \log_2 q)$

arithmetic operations with q-point 1-D FFT for each row, and $(MN/pq)(q \cdot \log_2 q)p$ for all rows. Including MN operations for twiddle-factor multiplication ($\tilde{D}_N$), it takes a total of $(MN/pq)(q \cdot \log_2 q)p + MN \approx O[MN(1+\log_2 q)]$ arithmetic operations. Note that all q-point 1-D FFTs can be computed in parallel.

Step 2. Column-wise data exchange with twiddle-factor multiplication:

$Y_2 = \tilde{D}_M(F_p \otimes I_{M/p}) \cdot Y_1$

Similar to Step 1, the DX is repeated for the p sub-blocks in each column. This step has a complexity of $O[MN(1+\log_2 p)]$ operations.

Step 3. Local 2-D DFT on each sub-block:

$\tilde{Y} = (I_p \otimes F_{M/p})Y_2(I_q \otimes F_{N/q})$

After the row-wise and column-wise data exchange with twiddle-factor multiplications, 2-D DFT computation is performed on each sub-block of size $M/p \cdot N/q$. This operation is fully parallel for all sub-blocks, and only limited by resource constraints in the underlying architecture. No data communication is required between any sub-blocks.

Step 4. Output permutation:

$Y = P(\tilde{Y})$

A bit-reverse permutation is required before generating the output. This is done by the host computer before display.

The pseudo code of the 2-D decomposition algorithm for the case when the image is of size $N \times N$ is presented in Fig. 3.2. The sub-blocks are of size $K \times K$, the mesh is of size $(N/K) \times (N/K)$, and the local memory is of size $S \times S$. Note that 1-D DFT is computed along rows and columns in each stage. Also notice that there are other 2-D decomposition methods, such as Vector Radix FFT [41], which recursively decomposes the 2-D DFT into small-sized ones, like 2×2 or 4×4 2-D DFT. While such methods can effectively reduce the number of multiplications, the recursive decomposition makes hardware implementation and memory addressing much more complicated.

INPUT:
Input size: $N \times N$
Local 2-D DFT size: $K \times K$
Mesh size: $(N/K) \times (N/K)$
Local memory size: $S \times S$

1. Data Exchange (DX)
   for $i = 1 : (N/S) \times (N/S)$
      Load DX data into local memory
      for $j = 1 : 2(S/(N/K))$
         Compute (N/K)-point 1-D FFT with twiddle-factor multiplication
      Store the result back to external memory
2. Local 2-D DFT (LDFT)
   for $i = 1 : (N/S) \times (N/S)$
      Load LDFT data into local memory
      for $j = 1 : 2(S/K)$
         Compute K-point 1-D FFT
      Store the result back to external memory

Figure 3.2: Pseudo code for 2-D decomposition algorithm for 2-D DFT

3.3 Proposed 2-D DFT Architecture

Fig. 3.3 gives a bird's eye view of the FPGA architecture for implementing the 2-D decomposition based 2-D DFT. It is composed of several components that can be classified broadly into *Domain-specific components* and *Infrastructure components*. Components such as processing elements (PEs) are designed specially for the 2-D DFT application and hence are domain-specific. Infrastructure components, including PowerPC, memory interface, host connection and UART, provide high-level control and support for the domain-specific components.

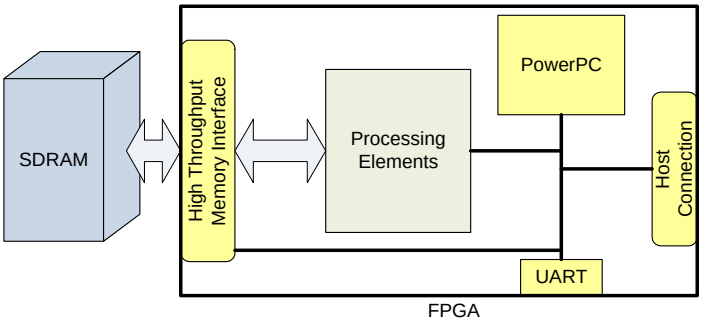


Figure 3.3: Block diagram of the FPGA architecture for 2-D decomposition based 2-D DFT.

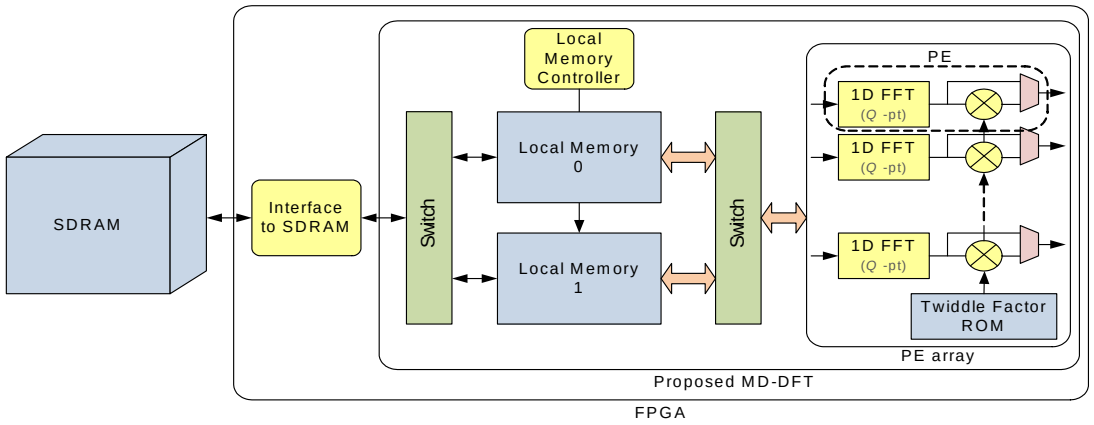


Figure 3.4: The proposed 2-D DFT architecture. (The size of the 1D FFT IP, Q, is described in Section 3.4.1)

3.3.1 Domain-specific Components

As shown in Fig. 3.4, the PE array consists of multiple PEs, where each PE is formed by a primitive 1-D FFT IP core and a complex multiplier. The primitive FFT is used for both data exchange and local 2-D DFT. In our design, we adopt Xilinx pipelined FFT IP [27]. The complex multiplier is used for twiddle-factor multiplication during data exchange. All PEs operate in parallel and access data from/to multi-banked local memory simultaneously without conflicts. The number of PEs and size of the primitive FFTs is determined by the memory bandwidth and available FPGA resources as explained in Section 3.4. Note that for ease of implementation, an input size of $N \times N$ is always assumed, where N is a power of 2, otherwise zero-padding is applied.

The input data is stored in the external memory (SDRAM), and is logically partitioned into a mesh of sub-blocks. During each step mentioned in Section 3.2, portions of data are loaded into the FPGA local memory, processed, and then stored back to the external memory. There are two identical local memories that serve as ping pong buffers. These local memories are implemented with dual-port Block RAM (BRAM) on the FPGA. The operations are described in details below.

In the data exchange stage which consists of the first four blocks in Fig. 3.1, first, equal number of samples from the same position in each sub-block is loaded into local memory. This data is then used for computing both row-wise and column-wise data exchanges. Note that the data from the external memory is accessed only along the row direction as depicted in Fig. 3.5. This

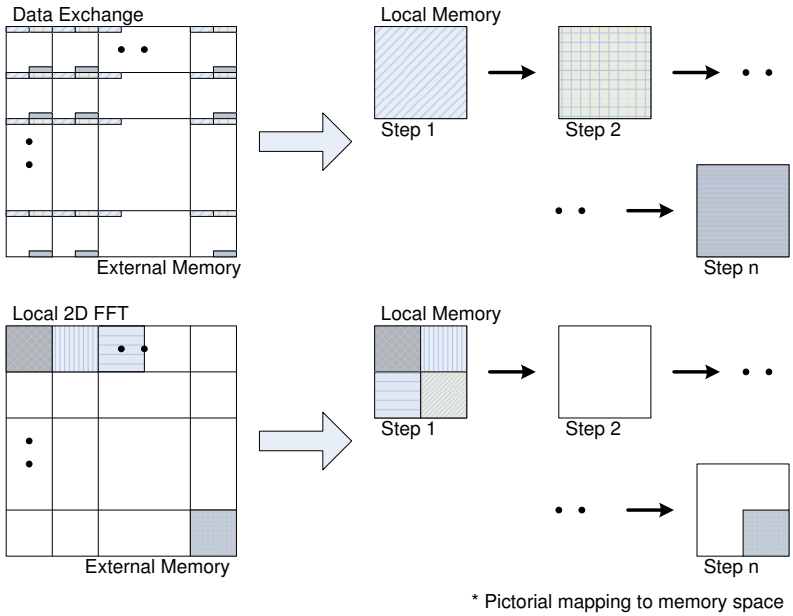


Figure 3.5: Data access pattern from external memory for data exchange and local 2-D DFT stages.

pattern is especially advantageous for accessing a dynamic memory, such as DDR2 and DDR3 SDRAM, which only favors row-wise burst access. The operations are repeated until the entire data is traversed, as shown in Fig. 3.5.

In the local 2-D DFT stage (corresponding to the fifth block in Fig. 3.1), fixed number of contiguous sub-blocks of the 2-D data are loaded into FPGA local memory and the PE array computes 1-D transforms along rows and then along columns. This operation is repeated for all the blocks.

3.3.2 Infrastructure Components

PowerPC is utilized for loading the input data into external memory, UART debugging and ethernet TCP/IP connections, and also run-time configurations for the domain-specific components, such as processing elements and data path controller. It is implemented as a Hard IP core in Xilinx FPGA, particularly the FX series in Virtex-4 and Virtex-5 device. If PowerPC can not be supported in case of LX, SX series of Virtex FPGA series, other soft-core processors like Microblaze can be utilized. UART block is used to provide a basic terminal to show status and debugging information. For host connection, currently Ethernet interface is preferred because Xilinx supports Hard tri-mode MAC

(TEMAC) for Virtex-4 and Virtex-5 [42]. A light-weight implementation of the TCP/IP protocol library, lwIP [43], is loaded to support ethernet communication.

High Throughput Memory Interface: While the PE array can finish computations very fast, the bottleneck is the interface to external memory. Xilinx's Multi-Port Memory Controller (MPMC), for instance, is a very versatile controller supporting SDRAM/DDR/DDR2 memory. However, its peak throughput is only 50% of a DDR2-SDRAM DIMM's peak transfer rate. To alleviate this bottleneck, we design a customized high throughput memory interface.

The customized memory interface, as shown in Fig. 3.6, has has a 128-bit wide internal data-bus. Since the DDR2 SDRAM device has an operating frequency of 200MHz and an user application in FPGA runs at 100MHz, the memory interface operates at 200MHz and has a 256-bit wide data bus between the interface and the application. This enables data transfer rates of up to 256 bits at 100 MHz or 3200 MBytes per sec. Together with double buffering technique on local memory, the customized memory interface enables us to completely overlap communication with computation and avoid any loss of performance due to communication bottle-neck. This memory interface can be ported onto any FPGA board with SDRAM DIMMs.

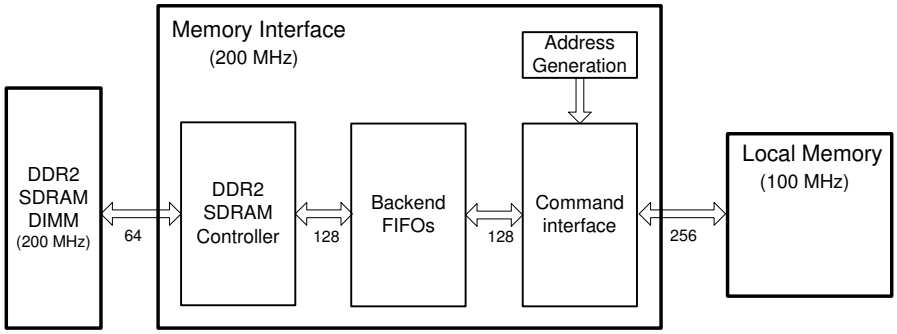


Figure 3.6: High throughput memory interface.

3.4 Automatic 2-D DFT System Generator

Given the specifications of input data and hardware platform, we propose a 2-D DFT system generator to automatically generate a 2-D DFT implementation based on the proposed 2-D decomposition algorithm. The automation flow is shown in Fig. 3.7. A design optimizer first determines the best decomposition based on input specifications to obtain the size of the primitive FFT. Then,

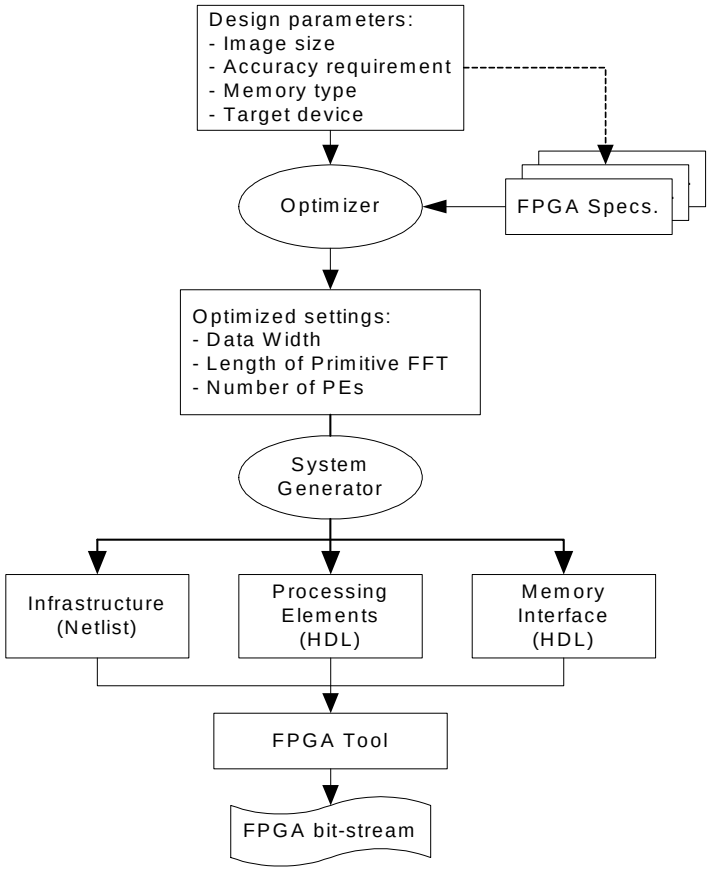


Figure 3.7: Automation flow of generating architecture for 2-D Decomposition based 2-D DFT.

according to the available FPGA resources and memory bandwidth, the optimizer calculates the number of PEs. The information is passed to the system generator, which generates hardware module in Verilog or VHDL. Then, the HDL files are fed into the FPGA tool to produce final configuration bit-files. Note that no user intervention is required for the entire process.

3.4.1 Choosing size of the FFT IP core

The input specifications include data size, target device, memory type, and accuracy requirement. The data size is a power of two, typically from 512×512 to 4096×4096. The target FPGA platform is Virtex-5 from Xilinx. For memory type, only DDR2 SDRAM is currently supported to provide large memory bandwidth. However, the memory interface can be easily extended to other memory types by replacing SDRAM controller block. The user can choose either 16-bit implementation for resource constrained designs, or 24-bit implementation for high accuracy designs.

Let $N = K \cdot L$. Then the sub-block size is $L \times L$ for a mesh of size $K \times K$. The design optimizer is used to find the best decomposition, i.e. L and K for a given N. Since our architecture uses high throughput memory interface (Section 3.3.2) and double-buffering technique to overlap communication with computation, the communication cost is not be considered in the modeling. Moreover, if the local memory has a fixed size $S \times S$, then there is an implicit constraint on the possible values for K and L namely, $K \leq S$ and $L \leq S$.

Assume that it takes T_{dx} to complete row-wise and column-wise data exchange operations, and T_{ldft} to complete local 2-D DFTs, respectively. Then the total DFT computation time T_{total} for input data of size $N \times N$ can be calculated as

$$T_{total} = (N^2/S^2) \cdot (T_{dx} + T_{ldft}). \tag{3.10}$$

Since the local memory can hold up to $(S/K)^2$ blocks for data exchange and $(S/L)^2$ blocks for local 2-D DFTs each time, then if $T_{dft(n\times n)}$ is defined as the time required for a 2-D DFT computation of size $n \times n$,

$$T_{dx} = (S/K)^2 \cdot T_{dft(K\times K)}, \tag{3.11}$$

$$T_{ldft} = (S/L)^2 \cdot T_{dft(L\times L)}. \tag{3.12}$$

For pipelined implementations, $T_{dft(n\times n)}$ is proportional to the 2-D DFT size $n \times n$, and

$$T_{dx} = (S/K)^2 \cdot c \cdot K \cdot K = c \cdot S^2, \tag{3.13}$$

$$T_{ldft} = (S/L)^2 \cdot c \cdot L \cdot L = c \cdot S^2. \tag{3.14}$$

where c is a constant. It can be seen that both data exchange time T_{dx} and local 2-D DFT time T_{ldft} depend only on the local memory size.

To fully utilize standard IP cores, only Q-point DFT will be implemented to accomplish both K-point and L-point DFTs, where $Q = max\{K, L\}$. So if the FPGA can support P Q-point processing elements, then

$$T_{dx} = T_{ldft} = c \cdot S^2/P. \tag{3.15}$$

Combining Eq. (3.15) with Eq. (3.10), we have

$$T_{total} = 2 \cdot c \cdot N^2/P. \tag{3.16}$$

Table 3.1: Resources required for Xilinx pipelined FFT IP.

		FFT point	32	64	128	256
Virtex-4	24 bit	DSP48	16	16	24	24
		BRAM	0	2	6	8
	16 bit	DSP48	6	6	9	9
		BRAM	0	1	3	4
Virtex-5	24 bit	DSP48E	24	24	36	36
		BRAM	0	2	5	7
	16 bit	DSP48E	8	8	12	12
		BRAM	0	1	3	4

This implies that the time to complete DFT depends on data size $N \times N$, and the number of processing elements, P, which is, in turn, determined by the resources available on the FPGA.

As primitive FFT size increases, the number of DSPs and BRAMs both increase, as shown in Table 3.1. For an FPGA with limited DSP and BRAM resources, we need to keep the value of Q as small as possible, so that more PEs can be accommodated. Since $Q = \max(K,L)$, and $K \cdot L = N$, the smallest value of Q is $\sqrt{N}$. However, if $\sqrt{N}$ is not an integer, we need to find a factorization of N so that K and L are close to each other. Once the optimal factorization is found, the length of the primitive FFT cores, Q, can be determined. For instance, if $N = 1024$, $Q = 32$, and if N increases to 4096, $Q = 64$.

3.4.2 Choosing the number of PEs

After choosing the size of the FFT IP core, the number of PEs is determined. Based on the hardware resources of the target FPGA and resources required by the FFT IP core given in Table 3.1, the 2-D DFT generator calculates the maximum number of PEs, P_{RES}, that can fit onto the FPGA. The external memory type/interface plays an equally important role in determining the number of PEs, since the pipelined Xilinx FFT IP [27] can input 1 datum/cycle. For instance, suppose that the data width of a complex datum is 32 (16×2) bits, and if FFT IP is running at 100 MHz, its data rate is 400 MBytes/s. Now, let P_{BW} be the maximum number of PEs that can be supported by the available memory bandwidth. If the external memory is one DIMM of DDR2-400 SDRAM, which can offer 3200 MBytes/s data rate, up to 8 PEs can be supported, and P_{BW} would be set to "8". In this case, the performance of the 2-D DFT cannot be improved with more PEs. Eventually, the number of

PEs is decided by the minimum of P_{RES} and P_{BW}, i.e.

$$P = min\{P_{RES}, P_{BW}\}. \tag{3.17}$$

Once the design optimizer decides the number and type of the primitive FFT IPs and memory partitioning, the system generator generates all the required hardware modules, namely, infrastructure block, FFT core block and memory interface block. The infrastructure blocks such as UART and host connection are fixed, whereas other blocks are user specified and provided in a template format, with several parameters that are set based on decisions of the design optimizer. After the generation of these modules, scripts for Xilinx flow are produced to run Xilinx tool automatically. Finally, the configuration bit file consisting of both hardware bit file and software binaries are generated from FPGA tool.

3.5 Evaluation

In this section, the 2-D DFT architecture generated using the DFT system generator is evaluated. First, the evaluation of high throughput memory interface is presented in Section 3.5.1, and the resource utilization is analyzed in Section 3.5.2. Then, the evaluation of performance for various input sizes is presented and compared with existing solutions in Section 3.5.3. Finally, the numerical accuracy of the 2-D DFT is analyzed in Section 3.5.4. For the evaluations, Xilinx 10.1 tool set and Modelsim 6.4 are used, and Virtex-5FX device is considered as the candidate FPGA device.

3.5.1 Memory throughput for RC- and 2-D decomposition- based architectures

The performance of the memory interface is measured in terms of the number of cycles taken to read/write data from/to DDR2 SDRAM to/from local memory. The performance for read, write and read+write operations for various memory access patterns for an input size of 2048×2048 and local memory size of 128×128 is shown in Fig. 3.8. It can be observed that column access is much slower compared to other access patterns, and hence it would be the bottle-neck for direct RC method. On the contrary, data exchange (DX) and local DFT (LDFT) are both row-wise accesses. Therefore, the proposed 2-D DFT avoids the performance penalty caused by column access and hence can achieve a higher performance.

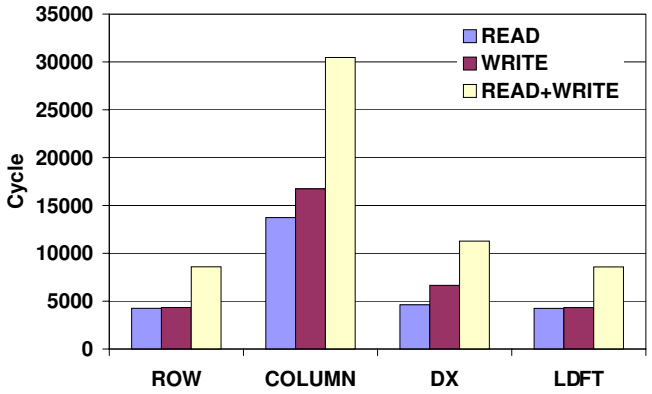


Figure 3.8: Memory access time for different access patterns when on-chip memory size is 128×128.

3.5.2 Resource utilization

The resource utilization of the PEs (Xilinx FFT IPs, complex multipliers, and the control units) and the memory requirement in terms of bandwidth, are summarized in Table 3.2. This evaluation is based on the device XC5FX200T, which has 384 DSP slices, 456 BRAMs (16 Mbs), 30,720 logic slices, and 122,880 slice LUTs. For large image sizes from 1024×1024 through 4096×4096, the size of the FFT IP can be either 32-point or 64-point. The data representation can be either 16-bit or 24-bit. The infra-structure block uses fixed number of resources of 8,149 logic slices and 47 BRAMs including TEMAC for host interface. We assume the operating frequency of the IPs is 100 MHz and the external memory device to be one DDR2-400 SDRAM DIMM operating at 200 MHz.

From Table 3.2, we see that the number of BRAMs and Slice LUTs utilized by 64-point FFT are slightly higher than 32-point FFT, though the number of DSP resources utilized is the same for both the cases. The number of DSP resources and the memory bandwidth required increase almost linearly with the number of PEs. Also, there is a large increase in DSP resources and memory bandwidth, when the data-width is changed from 16-bit to 24-bit.

From Table 3.2 we also see that the target FPGA can fit either 16 32-point FFT cores with 16-bit data width, or 8 32-point FFT cores whith 24-bit data width, or 8 64-point FFT core with 16-bit, or 8 64-point FFT cores with 32-bit data width. The DSP resource requirement for the 16

Table 3.2: Resource utilization and memory requirement for different configurations on Xilinx XC5FX200T FPGA.

Primitive FFT	Bit Width	# of PEs	DSP Slices	BRAM (36 Kbits)	Slice LUTS LUTs	Required Memory Bandwidth(MB/s)
32-point FFT	16bits	4PEs	36	137	25536	1600
		8PEs	72	137	35238	3200
		16PEs	144	145	54442	6400
	24bits	4PEs	112	145	33294	3200
		8PEs	224	145	46326	6400
64-point FFT	16bits	4PEs	36	139	26882	1600
		8PEs	72	141	37958	3200
	24bits	4PEs	112	149	34991	3200
		8PEs	224	153	50132	6400

FFT cores (32-point and 64-point) with 24-bit data representation exceeds that supported by the Virtex5 FPGA and hence cannot fit in the device. Also, the configuration with 16 64-point FFT cores with 16-bit data width cannot be supported due to local memory constraints. Thus, for both cases, the 2-D DFT system generator (see Section 3.4) would set P_{RES} as "8". Note that P_{RES} could be higher for other FPGAs with larger capacities.

Table 3.2 also list the required bandwidth for each configuration, and based on the table, the 2-D DFT system generator is able to determine P_{BW}. For example, if we have 1 DIMM of DDR2-400 SDRAM with a bandwidth of 3200 MB/s, then for 64-point FFT IP with 24-bit representation, $P_{BW} = 4$. Although P_{RES} in this case is "8", the final number of PEs is "4", based on Eq. (3.17). However if we have 2 DIMMs of DDR2-400 SDRAM, then the available memory bandwidth is 6400 MB/s, and the final number of PEs is 8. Note that P_{BW} could be higher if more DIMMS or a faster memory device, such as DDR2-800 SDRAM, is used.

3.5.3 Comparison of timing performance

Table 3.3 shows the performance of the 2-D DFT architecture for different input sizes and different architecture configurations. The performance is specified in terms of frame rate, which refers to the number of frames of input data that the 2-D DFT architecture can process in a second. The size of the FFT primitive is chosen by the design optimizer (see Section 3.4). It can be seen that the performance is inversely proportional to the input data size, but directly proportional to the number of PEs.

Table 3.3: Performance for different input sizes.

Input size	Primitive	# of PE	Computation time(ms)	Frame rates
1024×1024	32-point FFT	8	5.4	182
		4	10.7	92
2048×2048	64-point FFT	8	22.4	44
		4	43.4	23
4096×4096	64-point FFT	8	90.4	11
		4	174.3	5

Table 3.4: Performance comparison for different architectures for input size 2048×2048

	RC	Transpose	2D-DEC
Computation time (ms)	44.0	30.4	22.4
Frame rates	23	33	44

Table 3.4 compares the performance in terms of computation time and frame rate for FPGA implementations corresponding to (i) RC decomposition, (ii) Transpose (row-DFT - transpose - row-DFT) and (iii) the proposed 2-D decomposition algorithm. This comparison is done for identical architecture constraints such as one DDR2-400 SDRAM DIMM, 128×128 local memory size, 16-bit implementation and for an input data size of 2048×2048. The evaluation of the candidate architectures is performed using the proposed high throughput memory interface for the case when there are 8 PEs. The table shows that under identical conditions, the 2-D decomposition architecture provides a 96.4% improvement compared to the RC decomposition architecture and a 35.7% improvement compared to the transpose-based architecture. The improvement can be maintained across different image sizes, as shown in Fig. 3.9.

Further, Table 3.5 presents a comparison of other existing 2-D DFT implementations with the proposed 2-D decomposition based DFT (2D-DEC) architecture that utilizes maximum memory bandwidth on Virtex-5 FPGA. Uzun et al. [34] have used SRAM as the external memory with Virtex-E FPGA. Our 2D-DEC architecture provides significant performance improvement over [34] for the same input data size of 1024×1024. Dillon [21] has used two cascaded Virtex-II FPGAs for row-DFTs and column-DFTs and large SRAMs for intermediate data storage. In contrast, our 2D-DEC architecture uses a single FPGA with cheaper DDR2 SDRAM as external memory. Furthermore, our datawidth is 32 bits compared to 16 bits in [21]. Lenart et al. [19] have

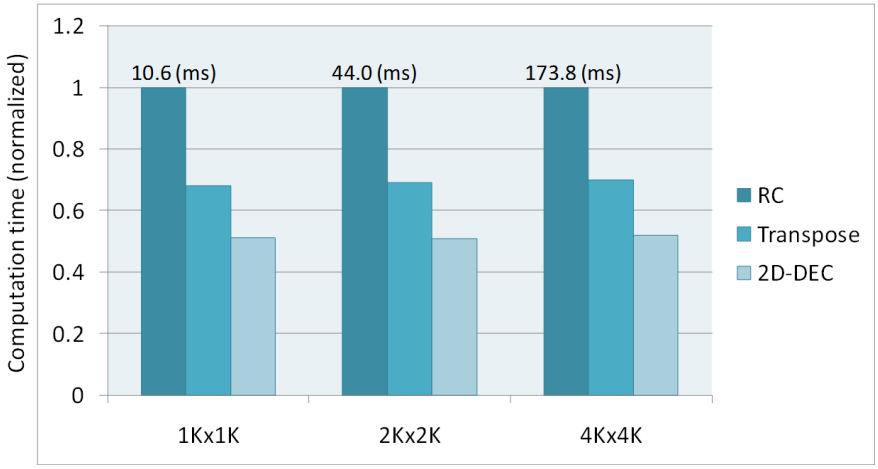


Figure 3.9: Comparison of the computation times (normalized to RC-based method) for different image sizes.

Table 3.5: Performance comparison with existing works.

Input size	Method	Technology	External memory	Complex data (bits)	Frame rate (fps)
1024×1024	Uzun [34]	Virtex-E, 180 nm, 27 MHz	Quad SRAM	2×16	13
	2D-DEC	Virtex-5FX, 65 nm, 100 MHz	Single SDRAM	2×16	182
2048×2048	Dillon [21]	Virtex-II$\times$2, 130 nm, 125 MHz	Dual SRAM	2×8	120
	Lenart[19]	ASIC, 130 nm, 250 MHz	Dual SDRAM	2×16	20
	2D-DEC	Virtex-5FX, 65 nm,100 MHz	Single SDRAM	2×16	44

implemented a transpose based architecture with DDR SDRAM memory. Our 2D-DEC architecture operates at 100 MHz compared to 250 MHz and still provides performance improvement by avoiding transpose operation.

3.5.4 Accuracy evaluation

As mentioned earlier, the proposed 2-D DFT is implemented using Xilinx pipelined FFT IPs, which are fixed-point cores. Due to finite wordlength effects, accuracy is a major design issue that needs to be analyzed.

An N-point Xilinx pipelined FFT core consists of $\log_2(N)$ stages of radix-2 butterflies. For an $N \times N$ 2-D DFT, therefore, every input element needs to be operated by $2\log_2(N)$ stages of radix-2 butterflies, irrespective of the decomposition algorithm used. To prevent overflow, Xilinx provides corresponding function (right-shifting 0-3 bits) on each pair of two stages in the FFT IP. The corresponding control bits can be programmed by the user. In this work, we first scale the input

Table 3.6: SNR(dB) of proposed 2-D DFT, where the input set is drawn from uniformly distributed random number and $|x(i_1,i_2)| < 1/2.4142$.

DFT Size	64×64	128×128	256×256	512×512	1024×1024	2048×2048
16 bit	70.56	69.94	69.67	69.36	68.72	67.72
20 bit	94.50	94.50	93.92	92.97	92.16	91.59
24 bit	118.77	118.17	117.52	117.15	116.81	116.03

array so that $|x(i_1,i_2)| < 1/2.4142$ [44] to prevent overflow in the first stage. Then we right-shift 1 bit for every pair of stages to prevent overflow in the FFT computation. This half-a-bit-per-stage scheme works for a broad class of signals [45].

To quantify the output accuracy, we feed uniformly distributed random numbers into the 2-D DFT. We adopt a commonly used criterion, Signal-to-Noise-Ratio (SNR), which is defined as:

$$SNR(dB) = 10\log_{10}\frac{P_{output}}{P_{quantnoise}},$$

where P_{output} is average output power, and $P_{quantnoise}$ is acquired by comparing the hardware output with Matlab floating point results. The SNR results for randomly generated input data are listed in Table 3.6.

From Table 3.6, three observations can be made. First, SNR of the 2-D DFT is mainly dominated by Xilinx FFT IP. For example, a 16-bit 1024-point Xilinx FFT has an SNR around 73dB [27]. The 16-bit 2-D DFTs have lower SNRs than 73dB, because the 2-D DFTs have more stages of radix-2 butterflies. Second, we can gain about 6 dB of SNR when 1 additional bit is added to the data width. This observation is consistent with the results in [27]. Based on this, the user can decide the data width. Thirdly, the accuracy is insensitive to the 2-D DFT problem size (as shown in this case). To explain this, one should notice that the denominator of the SNR equation is dominated by data width, which is fixed, and the numerator is dominated by the signal power (or amplitude). In Fig. 3.10, we record the internal maximum amplitude of every stage in the 2-D DFT. The values are maintained around a certain value (0.75 in this case), irrespective of the number of butterfly stages. That means the signal power is almost fixed throughout the whole computation. As the denominator and numerator are both fixed, the SNR value can be maintained.

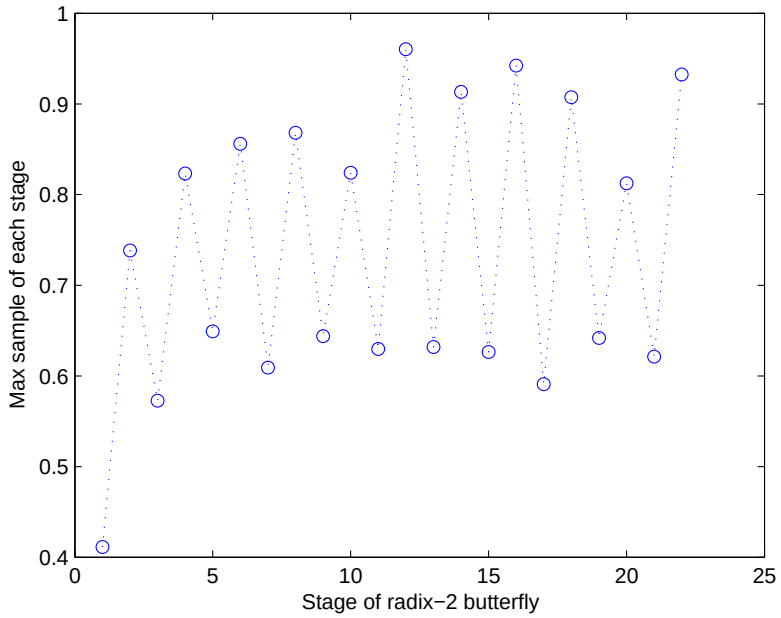


Figure 3.10: Maximum amplitude of every stage in a 2048×2048 2-D DFT, where the input is uniformly distributed random numbers.

Besides, Fig. 3.10 indicates that the internal data utilizes full data-width but never lets overflow occur. It also proves that the scaling scheme works well.

If accuracy is not critical, it is recommended to use a smaller data-width to minimize the hardware resources occupied by the primitive FFTs. Typically, a 16-bit Xilinx FFT core occupies only 30-40% of resources compared with a 24-bit design. With saved resources, we can put more primitive FFTs into the FPGA to further accelerate the 2-D DFT computation.

3.6 Summary

In this chapter, we have proposed an efficient architecture to implement the 2-D DFT for large input sizes based on a novel 2-D decomposition algorithm. This architecture provides high performance by leveraging the inherent parallelism of the 2-D decomposition and by scheduling data communication to overlap with computation. The memory bandwidth problem is alleviated by employing a custom-designed high throughput memory interface. In addition, a system generator is provided, which can automate the generation of an optimized version of the 2-D DFT architecture for various

input sizes. The evaluation of this architecture shows significant performance enhancements over existing 2-D DFT implementations.

One drawback of this implementation is that the outputs are generated in bit-reversed order. This could be a problem if the 2-D DFT module is fed to another module that need its input in natural order. In the next chapter, we describe another implementation of MD DFT where the outputs are generated in natural order.

Chapter 4

Multi-Dimensional DFT IP Generators for FPGA Platforms

4.1 Introduction

In this chapter, we propose an automated MD DFT hardware IP generator for implementing 2-D and 3-D DFT on FPGA platforms. The optimized MD DFT architecture maximizes the external memory bandwidth. It achieves this by accessing the memory data in a way that avoids transpose operations and utilizes the burst access pattern of the SDRAM memory. The proposed MD DFT architecture is automatically generated by an in-house IP generator. Based on the image size, dimensionality, and FPGA hardware constraints, the IP generator produces optimized HDL code that utilizes the on-chip FPGA resources efficiently. The DFT IPs have been ported onto the BEE3 board. The results prove that our implementation is transpose-free and that it can maintain the maximum memory throughput rate for the entire computation.

The rest of this chapter is organized as follows. In Section 4.2, the proposed MD DFT algorithm is derived. Section 4.3 describes in detail the proposed DFT architecture. A newly developed FPGA framework is described in Section 4.4. The implementation details and evaluation results are discussed in Section 4.5, and concluding remarks are given in Section 4.6.

4.2 MD DFT Algorithm

The general form of 2-D DFT can be described in matrix form as follows. 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:

$$V_2 = F_{N_2} \cdot U \cdot F_{N_1}. \tag{4.1}$$

In Eq. (4.1), $V_1 = U \cdot F_{N_1}$ can be done by applying 1-D DFT along the rows of U, and $(F_{N_2} \cdot V_1)$ by applying 1-D DFT along the columns. This is the traditional Row-Column (RC) decomposition. If the data along the rows of U are stored in a one-dimensional memory, say SDRAM, we can efficiently access data in consecutive location while executing row DFT. However, adjacent data along the columns are no longer in consecutive addresses, and the long strides that are necessary to gather data along columns could make the access latency about 10 times longer during column

DFT computation [19]. To reduce access latency of column data, in many implementations, the data is transposed after row operation [19] so that the column-wise data can be re-aligned into adjacent addresses in the memory and can still be accessed in a burst manner when executing column DFT. However, transpose operations cost extra time, and during matrix transpose operation, the computation unit remains idle. Furthermore, when computing higher dimensional DFT, such as 3-D DFT, much larger on-chip local memory is required for the transpose operation, which cannot be supported on some smaller FPGAs.

In the rest of this section, we first describe the memory-aware 2-D DFT algorithm, which is transpose-free. Then, we also show how the proposed operations in 2-D DFT can be reused in 3-D DFT.

4.2.1 Proposed 2-D DFT Algorithm

To maintain row-wise burst for column DFT computations, we utilize the local memory on FPGA and store multiple columns of data. Let S be the size of the local memory, then we can store S/N_2 columns. If S/N_2 is less than the SDRAM's burst size B, then the SDRAM bandwidth is not utilized completely. For instance, if $B = 32$, $S = 16,384$ and $N_2 = 1024$, then $S/N_2 = 16 < 32$, the burst size. To utilize burst size, B, we decompose the column computations of size N_2 into 1-D computation of size m followed by 1-D 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:

$$V_2 = \underbrace{P_{N_2,m} \cdot (I_m \otimes F_p) \cdot \underbrace{\tilde{D}_{N_2} \cdot (F_m \otimes I_p) \cdot \underbrace{U \cdot F_{N_1}}_{\text{Step 1-a}}}_{\text{Step 1-b}}}_{\text{Step 2}}. \tag{4.2}$$

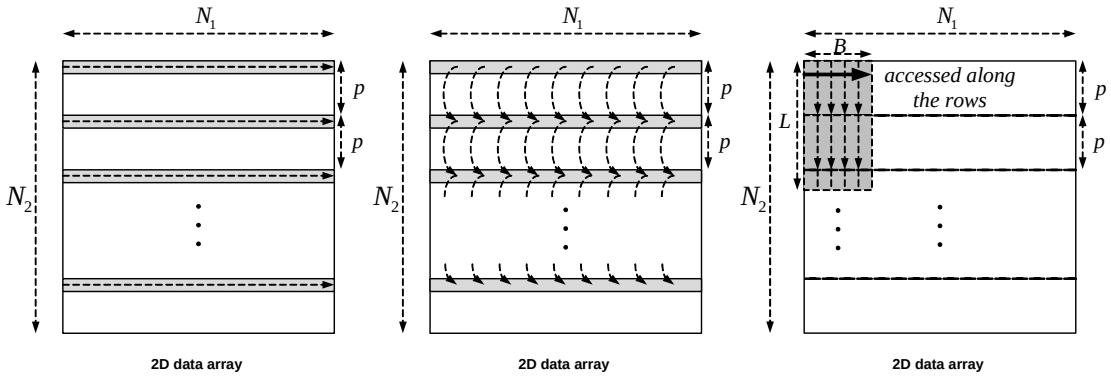
where I_m and I_p are the identity matrices of sizes $m \times m$ and $p \times p$, respectively, $\tilde{D}_{N_2}$ is a diagonal matrix of twiddle factors, $P_{N_2,m}$ is the column permutation, and $\otimes$ is the Kronecker product. Eq. (4.2) can be summarized as follows:

[2-D DFT Procedure]

- **Step 1. Row Operations**: This step includes two operations:
  - Step 1-a. Row DFT ($U \cdot F_{N_1}$): Compute the DFT along the rows of array U.
  - Step 1-b. Column stride DFT: Column DFT of size m followed by twiddle multiplications.
- **Step 2. Column Local DFT**: Column DFT of size p followed by column-wise permutation.

Note that Step 1-b & Step 2 form the column DFT. The data access pattern for the 2-D DFT is illustrated in Fig. 4.1. In Fig. 4.1a, m rows spaced p rows apart are selected from the data array and $N_1 \times m$ data is sent to the FPGA local memory. Since the local memory is of size S, $S \geq N_1 \times m$, DFT of N_1 points is executed along each of these rows. Then, column-wise m-point DFT followed by twiddle factor multiplication is applied as shown in Fig. 4.1b. The result is stored back in the same location of the data array in the SDRAM. After p iterations of Step 1-a & 1-b, all row DFTs and column stride DFTs are completed. Then Step 2, which consists of p-point local DFT, is computed. As shown in Fig. 4.1c, L rows with B elements per row are stored in the local memory. Thus $S \geq B \times L$, where $L \geq p$. This enables p-point local DFT to be computed on these sub-columns. Note that if $N_2 \leq L$, column stride DFT can be skipped, because the whole column can fit in the local memory and the decomposition is not required. Next, the data along the rows have to be stored back to the correct row locations (based on the column-wise permutation, $P_{N_2,m}$) in the SDRAM.

We compare the theoretical time consumptions of three different 2-D DFT solutions in Table 4.1. Direct RC implementation only needs to access the data array in the SDRAM two times. However, the column access is K times longer, where K could be as high as 10 [19]. The transpose-based solution needs to access the SDRAM four times, because of the two additional transpose operations. Note that during the transpose operation, the computation units are idle. In contrast, the proposed design only needs to access the SDRAM two times, for row operations (Step 1) and for column local DFT (Step 2). The row operations could take two times longer ($\alpha = 2$) than the column local DFT if column stride DFT is activated. However, if the row operations can be fully



(a) Step 1-a. Row DFT (b) Step 1-b. Column stride DFT (c) Step 2. Column local DFT

Figure 4.1: The proposed data access pattern for 2-D DFT .

Table 4.1: Comparison of computation times for 2-D DFT.

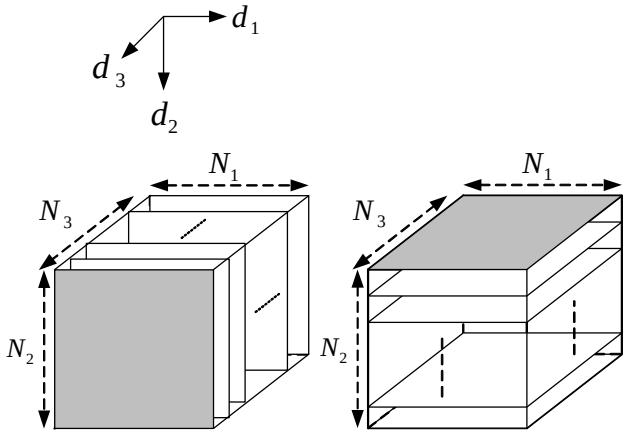
Direct RC		Transpose		Proposed	
Operations	Time	Operations	Time	Operations	Time
Row DFT	T	Row DFT	T	Row operations	αT
Column DFT	KT	Transpose	T	Column local DFT	T
		Row DFT	T		
		Transpose	T		
Total	(K+1)T	Total	4T	Total	$(\alpha+1)$T

overlapped with data accesses to the external memory, as is done in our proposed method, α can be reduced to 1. Thus, under the same hardware constraints, the proposed method can be up to two times faster than the transpose-based solution and much faster than the direct-RC implementation.

4.2.2 3-D DFT Algorithm

3-D DFT is a simple extension of 2-D DFT. As illustrated in Fig. 4.2a, the 2-D DFT algorithm (described in Section 4.2.1) is computed on each of the N_3 2-D slices parallel to the d_1-d_2 plane. Next, 1-D DFT of size N_3 is done along the d_3-axis. Since adjacent data along the d_3 dimension are not in consecutive addresses in the SDRAM, to utilize the burst along d_1 dimension, we access data on a 2-D slice parallel to d_1-d_3 plane and apply the decomposition along d_3 dimension, as shown in Fig. 4.2b. The mathematical description of this procedure is given by:

$$V_{1,3} = F_{N_3} \cdot U_{1,3} = \underbrace{P_{N_3,m'} \cdot \left(I_{m'} \otimes F_{p'}\right) \cdot \underbrace{\tilde{D}_{N_3} \cdot \left(F_{m'} \otimes I_{p'}\right)}_{\text{Step 2-a}} \cdot U_{1,3}}_{\text{Step 2-b}}, \tag{4.3}$$



(a) 2-D DFT on each slice (b) DFT along d_3 on each slice

Figure 4.2: The proposed data access pattern for 3-D DFT .

where $U_{1,3}$ represents an input 2-D array parallel to d_1-d_3 plane and $V_{1,3}$ is the output array; $N_3 = m' \times p'$. The whole 3-D DFT procedure can be summarized as follows:

[3-D DFT Procedure]

- **Step 1. Run [2-D DFT Procedure] on N_3 slices**: Compute 2-D DFT on every 2-D slice parallel to d_1-d_2 plane.
- **Step 2. DFT along d_3 dimension (Eq. (4.3))**:
  - Step 2-a. Stride DFT: Compute m'-point stride DFT followed by twiddle multiplications on the 2-D slice parallel to d_1-d_3 plane.
  - Step 2-b. Local DFT: Compute p'-point local DFT followed by the permutation on the 2-D slice parallel to d_1-d_3 plane.

Step 2 has to be iterated N_2 times. After Step 3, the final results have to be stored to the correct locations in a different part of the SDRAM. Similar to the 2-D DFT procedure, Step 2 can be skipped if $N_3 \leq L$.

Note that the purpose of the decompositions along d_2(column)- and d_3-dimension is to utilize the burst access along d_1-dimension(row) throughout the whole 3-D DFT computation. With

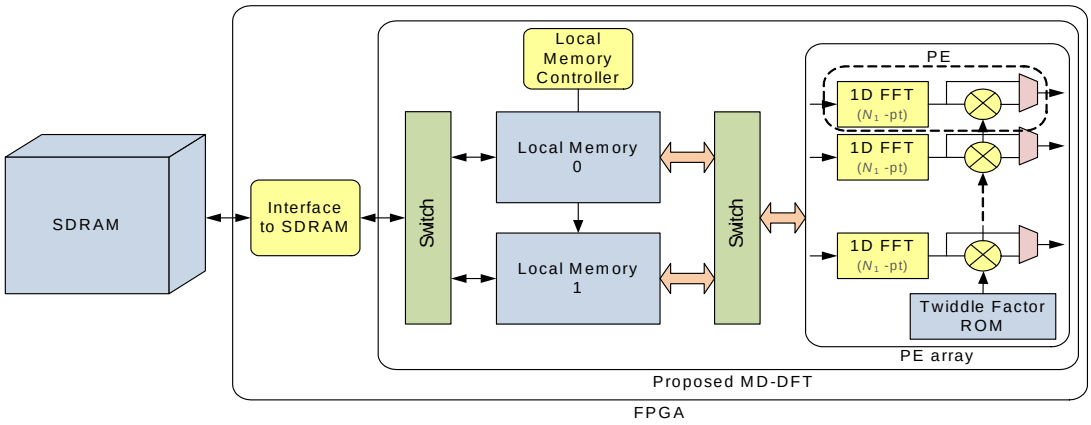


Figure 4.3: The proposed MD DFT architecture.

this decomposition technique, we can avoid the data transpose required in other 3-D DFT implementations [11][46] and achieve higher performance.

4.3 Proposed DFT Architecture

4.3.1 Architecture Overview

The architecture proposed for MD DFT is shown is Fig. 4.3. The main components are an FPGA to do the processing and an SDRAM to store the data. The SDRAM controller fetches the input data from SDRAM and sends it to the local memory on the FPGA. The processing elements (PE) read this data, process it, and store the results back to the local memory. The SDRAM controller then reads these results from the local memory and stores them back to the SDRAM. The main components of the architecture are described below:

Processing elements (PEs): A PE consists of a 1-D DFT module followed by a complex multiplier. The 1-D DFT module can support a maximum of N_1-point DFT for computing along rows, but it can also compute column stride/local DFTs of other lengths $L \leq N_1$. In our design, we adopt Xilinx's streaming Fast Fourier Transform (FFT) IP core [27]. The complex multiplier is used to compute the twiddle multiplication after column stride DFT. The number of PEs depends on the required data throughput as well as the hardware resources available on the FPGA.

SDRAM: SDRAM is the main memory used to store the multi-dimensional data. In our implementation, a 2GB DDR2-400 DIMM is adopted. For 2-D or 3-D images, consecutive data along

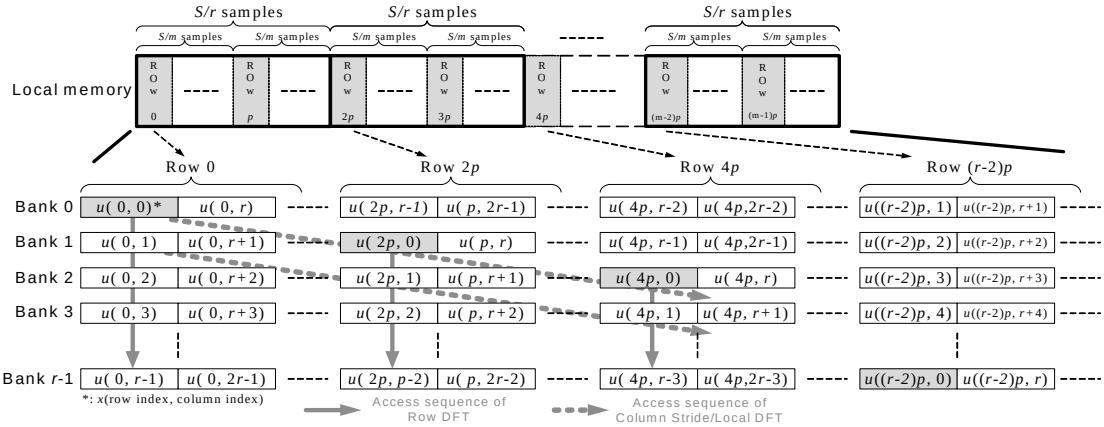


Figure 4.4: Data organization and the access patterns of the local memory.

d_1 dimension is stored in consecutive locations in the SDRAM. We use AlgoFLEX (described in Section 4.4) to transfer the data between SDRAM and FPGA. The data is always accessed along d_1 dimension, thereby fully exploiting SDRAM's bandwidth efficiency.

Dual local memory: There are two identical local memories of size S that serve as ping pong buffers. These local memories are implemented with dual-port Block RAMs on the FPGA. Unlike the SDRAM, non-consecutive addresses in the Block RAM can be accessed in contiguous clock cycles without performance penalty. Each local memory consists of multiple banks, so that multiple data can be received within one cycle from the SDRAM, and data in the local memory can be accessed by multiple 1-D FFT IP cores at the same time. To support simultaneous accesses from multiple (up to r) PEs, each local memory is divided into r banks, as illustrated in Fig. 4.4. If S is the size of the local memory, for the row operation, the SDRAM controller fetches the first S/m consecutive data (i.e. the first $S/(m \times N_1)$ rows) from the SDRAM and stores them into the banks starting from Bank 0. Then, the SDRAM controller fetches the next S/m consecutive data starting from the pth row and stores them starting from Bank 1, and so on. Such a storage scheme enables r PEs to access the data that were in the same bank. The arrows in Fig. 4.4 show how the conflict-free accesses work for row DFT and column stride DFT. The same addressing scheme also works for column local DFT. Note that the local memory can also receive multiple input data via a wider bus or SDRAM interface with the multiple-banked organization.

4.4 Test Platform and MD DFT IP Generator

To implement and evaluate the proposed MD DFT design on different FPGAs, we have developed an automated MATLAB-based MD DFT IP generator. The IP generator automatically calculates the size of the FFT IP in the PE, and the optimal number of PEs, N_{PE}, based on image size, FPGA resources and external memory bandwidth. Then, it generates the corresponding HDL (Verilog) files which can be fed into the FPGA tool to produce the final configuration bit-files. The automation flow of the MD DFT IP generator is shown in Fig. 4.5.

In order to integrate the MD DFT cores with other programmable and customized cores for large system design, we have also developed a framework called AlgoFLEX. This framework provides a backbone hardware platform for seamless integration of customized cores. Further, it provides a front-end GUI interface and back-end synthesis software chain for mapping the desired set of IP blocks onto a target FPGA. We only explain the portions of this framework essential to understanding the integration of MD DFT core on to this platform.

4.4.1 AlgoFLEX Platform

The AlgoFLEX framework has been designed for integrating multiple cores to aid fast development of complete applications on FPGA based hardware platforms. On the hardware side, it provides various standard interfaces, like system bus and SDRAM controller for plug-n-play user-defined modules. On the software side, the platform incorporates a unified graphical user interface (GUI) which allows the user to modify the configurations, execute the FPGA implementation flow and display the results.

The AlgoFLEX hardware infrastructure, shown in Fig. 4.6, provides a system designer with the capability to plug-n-play a collection of custom accelerator blocks, such as our MD DFT core, without requiring intimate knowledge of how those blocks will be composed. As shown, the platform consists of a hybrid communication network comprised of a System Local Bus (SLB) and a packet-based on-chip router along with memory controllers, accelerators, and other system components. Currently, we adopt Xilinx's 128-bit Processor Local Bus (PLB) [47] as SLB and 64-bit Multi-Port Memory Controller (MPMC) [48] as the memory controller. While the SLB serves

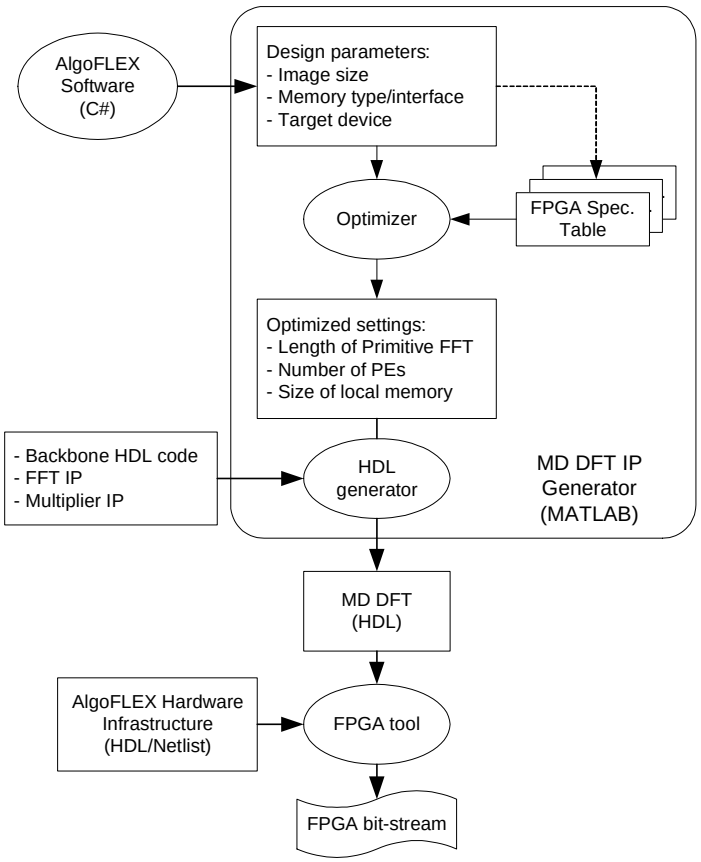


Figure 4.5: Automation flow of generating architecture for the proposed MD DFT.

as the communication backbone within a single FPGA, the router is used to coordinate a secondary channel for both intra- and inter-FPGA communication.

To integrate accelerators such as our MD DFT into the base hardware configuration of AlgoFLEX, we have designed a Universal Custom Accelerator Module (UCAM) wrapper. It defines a standard interface between the common system level facilities and an instance of an accelerator module. It takes advantage of the fact that many signal processing applications can be characterized by the following: 1) at the application level a sequence of subtasks are repeatedly performed on incoming data and 2) in each subtask, a static set of compute-intensive operations are performed whose sequence can vary across different invocations. For example, an optimized MD DFT implementation may selectively perform decomposition of data into 1-D DFT operations based on the dimensionality, but the sequence of these operations may be radically different across different problem specifications.

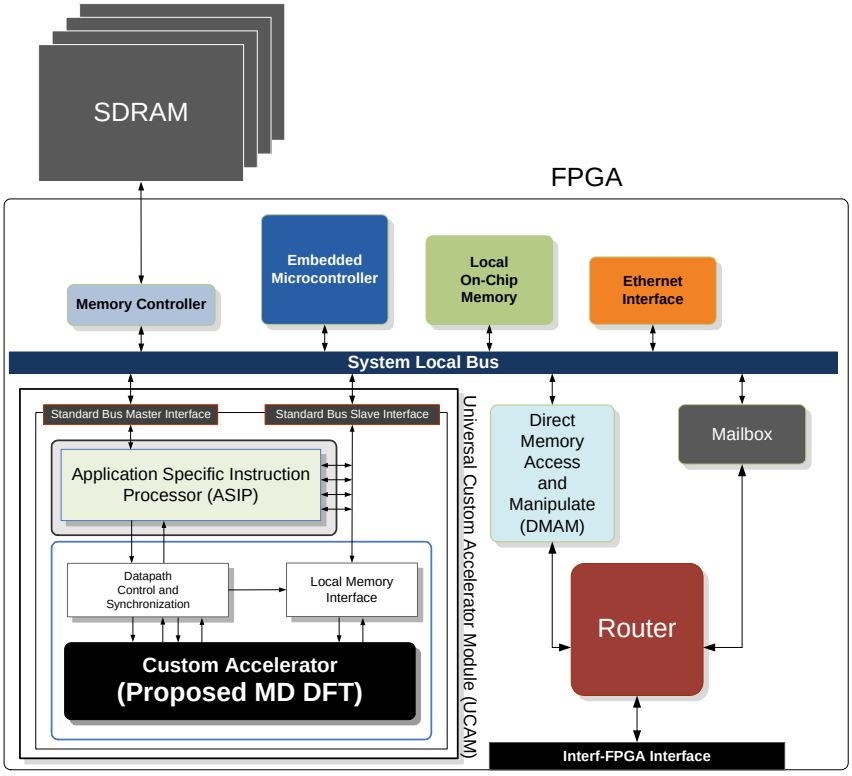


Figure 4.6: AlgoFLEX Hardware Infrastructure.

Furthermore, each UCAM is equipped with an Application Specific Instruction Processor, ASIP, that facilitates fetching, decoding, and preprocessing of accelerator specific instruction sequences. It also provides a set of built-in Command Handlers that are useful for many algorithm accelerators. For the case of 2-D and 3-D DFT, the window fetch handler supports optimized fetching and storing of n-dimensional data with programmable column size, number of rows and columns, row and column offset, inter-row stride, inter-column stride, and inter-dimension offset. The provisioning of such features within the AlgoFLEX framework facilitates reuse of design effort across modules as well as helps in quickly adapting the customized cores to a different target FPGA with changes only in a small set of base platform modules.

AlgoFLEX also provides a drag-and-drop graphical interface that allows the user to compose a system by diagramming the algorithmic specification onto a canvas using AlgoLets, as shown in Fig. 4.7a. An AlgoLet is an abstraction of an algorithmic entity that, when synthesized, is implemented in one or more associated UCAM modules. The dataflow between AlgoLets is

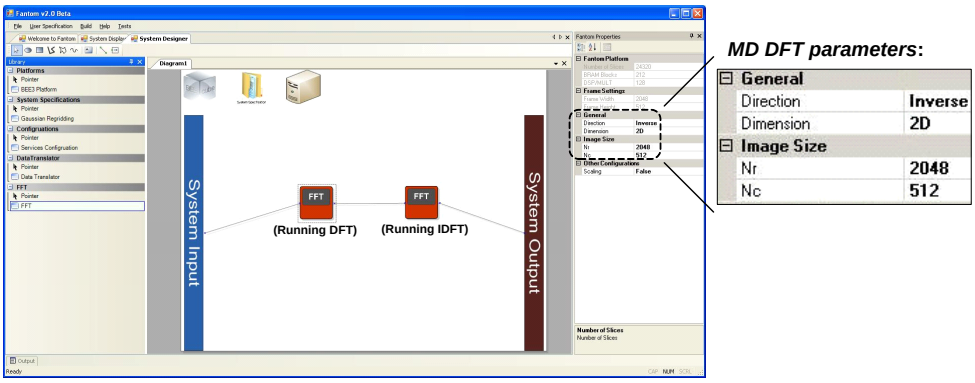
described by the user by drawing connections between several AlgoLets. AlgoFLEX automatically infers control flow, maps UCAMs to FPGA resources, instantiates Direct Memory Access and Manipulate (DMAM) module and Stream Operators, and executes synthesis scripts necessary to generate bitstreams for each FPGA in the platform. An example command set and execution sequence for 2-D DFT can be seen in Fig. 4.8. In addition to hardware, software is generated for the embedded microcontroller to perform initialization tasks such as configuring routing tables, performing memory allocation, and memory mapping for each UCAM. The executables for each microcontroller are loaded using a JTAG debug interface while the system initialization data is loaded through an Ethernet interface. For instance, in Fig. 4.7a, two MD DFT AlgoLets are dragged and linked on the canvas. The former one runs DFT, and the latter one runs IDFT. This setting is for the MD DFT function verification. In Fig. 4.7b, the GUI displays the input and output images, which if identical, implies that the MD DFT/IDFT works correctly.

4.4.2 Automated MD DFT IP Generator

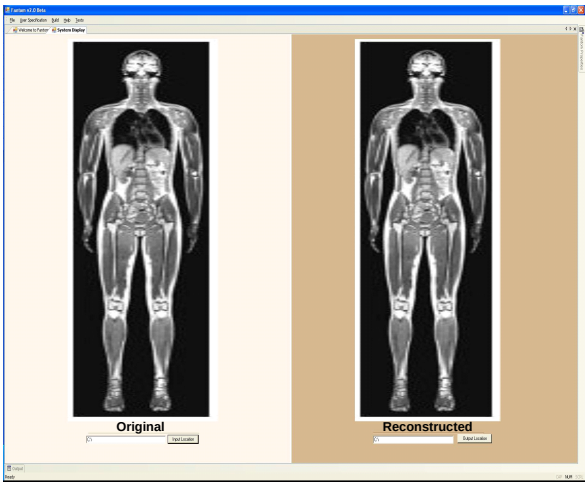
In this section, we describe the operations in the MD DFT IP generator shown in Fig. 4.5. It receives the user-selected device, memory type/interface, the required image size, and dimensionality from the AlgoFLEX GUI. Based on the image size, it chooses the size of the Xilinx 1-D FFT IP. Then, based on the hardware resources of the target FPGA, the IP generator picks the number of PEs, $N_{PE\_Resource}$. Note that the external memory type/interface also affects the number of PEs, since the pipelined Xilinx FFT IP [27] can input 1 datum/cycle. For instance, if the memory interface can transfer 128 bits/cycle and an image sample is 64-bit wide, only 2 samples from the external memory can be read in one clock cycle, and the performance of the MD DFT cannot be improved with more PEs. Let $N_{PE\_Bandwidth}$, be the number of PEs based on the memory bandwidth. Then, the number of PEs is the minimum of $N_{PE\_Resource}$ and $N_{PE\_Bandwidth}$, i.e.

$$N_{PE} = min\{N_{PE\_Resource}, N_{PE\_Bandwidth}\}. \tag{4.4}$$

The MD DFT IP generator also has the capability to determine if the image size can be supported by the FPGA platform or not. It computes the maximum image size based on the following equations: First, let the 1-D FFT IP's maximum size, N_{FFT_IP} be the maximum value of N_1,



(a) An FFT-IFFT bi-directional test for functionality verification.



(b) Compare the original and reconstructed images.

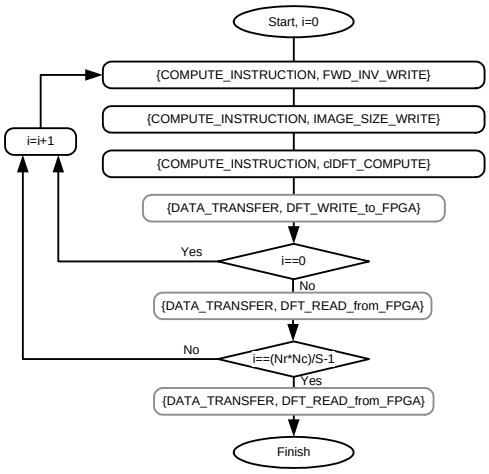
Figure 4.7: Graphical user interface of AlgoFLEX.

which is provided by the user through the GUI. So

$$N_{FFT\_IP} = N_{1\_\max}. \tag{4.5}$$

The local memory size, S, decides the number of rows, m, which can be loaded into the FPGA in row DFT/column stride DFT computations, where $m = S/N_{1\_\max}$. The burst size, B, which is an SLB parameter, affects the sizes of N_2 and N_3. For column local DFT, L rows with B samples per row are loaded onto the local memory, i.e. $L = S/B$. L is actually the maximum size of column local DFT, p, and $N_2 = p \times m$. Therefore, the maximum value of N_2 is

$$N_{2\_\max} = L \times m = \frac{S^2}{N_{1\_\max} \times B}. \tag{4.6}$$



Command Option	Parameter	Description
rDFT_COMPUTE = 2	--	Run row DFT
csDFT_COMPUTE = 3	--	Run column stride DFT
rDFT_csDFT_COMPUTE = 4	--	Run row operations
clDFT_COMPUTE = 5	--	Run column local DFT
NULL_COMPUTE = 6	--	No computation (For memory read/write tests)
csDFT_START_rADDR_WRITE = 7	Row address of the first sample	Specify starting address for twiddle factor generating in column stride DFT
IMAGE_SIZE_WRITE = 8	{log2(Nc), log2(Nr)}	Specify image size
FWD_INV_WRITE = 9	IDFT:0, DFT:1	Choose IDFT or DFT

Figure 4.8: (Left) A pseudo code of the algorithm specified using command sequence; (Right) Commands exposed by the accelerator to the algorithm designer.

In other words,

$$N_{1\_\max} \times N_{2\_\max} = \frac{S^2}{B}. \tag{4.7}$$

A similar analysis holds for N_3, and

$$N_{1\_\max} \times N_{3\_\max} = \frac{S^2}{B}. \tag{4.8}$$

For example, in our implementation, the local memory size S on a Virtex-5 LX155T FPGA is 16384, and the burst size B is 32 samples. Based on Eqs. (4.7)-(4.8), the MD DFT implementation can support up to 2048×4096 2-D DFT and $2048 \times 4096 \times 4096$ 3-D DFT. If the user inputs N_2 and N_3 larger than $N_{2\_\max}$ and $N_{3\_\max}$, the GUI would show a warning message and suggest the user to reduce the image size.

4.5 Evaluation

The proposed MD DFT architecture has been generated by the AlgoFLEX platform and its functionality has been verified on the BEE3 board [49], which is equipped with a Virtex-5 LX155T FPGA. We assume that $N_{1\_\max} = 2048$ in our experiments. For this configuration, the FPGA can only accommodate one PE, which consists of a 2K-point FFT IP and a complex multiplier, since this occupies more than half (53%) of DSP48Es. The local memory size, S, is 16,384 samples. The ping pong buffer and other memory in the 2K-point FFT IP consume 41% of Block RAMs, and the other Block RAMs are needed to support AlgoFLEX's infrastructure. The hardware resource

Table 4.2: Hardware resource utilization of the MD DFT IP on a Xilinx Virtex-5 XC5LX155T FPGA.

Slices	DSP48Es	Block RAMs
25%	53%	41%
(8,273/33,088)	(68/128)	(87/212)

utilization of the MD DFT IP is summarized in Table 5.4. Since there is only one PE, single-banked local memory would have been sufficient. However, we choose to divide it into 2 banks, since in each cycle the 128-bit SLB can transfer 2 complex samples with single precision and store them into the local memory. For maximum performance, SLB's burst size is set to the largest value: 16 cycles. Thus, the effective burst size, B, is $16 \times 2 = 32$ complex samples. In our implementation, m is set to 8, because 8 rows of length 2048 can fit in the local memory. The clock frequency is set to 100MHz. A timer on the FPGA is used to count the clock cycles elapsed during the computations.

4.5.1 2-D DFT

The computation times (measured) of 2-D DFT for square and rectangular images of different sizes are listed in Table 4.3. First, these results show that column local DFT takes almost the same time as the row operations. This means that the row-wise burst access mode for the column local DFT computations achieve the same bandwidth efficiency as the row operations. Secondly, the computation time is proportional to the image size, that is, if the computation time is T for an $N \times N$ image, it is $4T$ for a $2N \times 2N$ image. So a 512×512 2-D DFT takes about four times longer than 256×256 2-D DFT; a 1024×1024 2-D DFT takes almost the same time as 2048×512 2-D DFT, because their data sizes are the same.

Table 4.3: Measured computation time of 2-D DFT on BEE3.

Shape	Image size ($N_1 \times N_2$)	Row operations (ms)	Column local DFT (ms)	Total (ms)
Square	128×128	0.89	0.90	1.79
	256×256	3.01	3.04	6.05
	512×512	12.14	12.72	24.86
	1024×1024 *	50.21	52.42	102.63
	2048×2048 *	202.45	209.65	412.10
Rectangle	512×2048 *	50.34	52.37	102.71
	2048×512	50.11	52.47	102.58

(*: Column stride DFT is required.)

In the current implementation on the BEE3 board, the SDRAM's peak performance is not fully exploited due to the slow SLB (Xilinx's PLB) and the memory controller, MPMC. In the 2K×2K DFT, the average data transfer rate is only 325.69MB/s, while the peak transfer rate of the SDRAM is 3200MB/s. To achieve a much higher performance, the proposed architecture is currently being ported onto the Xilinx ML605 FPGA board [50], which is equipped with a Virtex-6 LX240T FPGA and a DDR3-800 SDRAM SO-DIMM. This FPGA can accommodate 8 PEs, which can easily deal with the full SDRAM bandwidth. In addition, if a dedicated SDRAM controller is designed that can utilize 80% of the SDRAM's bandwidth, the proposed design can complete 2K×2K DFT in 26.2ms.

With ML605, the proposed architecture can outperform other 2-D DFT solutions listed in Table 4.4. To make a fair comparison, we extrapolate the performances of the different architectures for the same data width, namely, 2×32 bits (single precision complex data). We assume that in all cases the performance is constrained by the data transfer between the external memories and FPGA/ASIC, and that the bandwidth of the external memory is the same as the original implementation. So the normalized time consumption is 64/(data width) of the reported computation time. Under this scenario, both Dillon's implementation [21] and ML605 (simulated) have very low computation times, around 30ms for 2K×2K data. While our design requires only one SDRAM, Dillon's solution [21] utilizes multiple SRAM modules and a memory controller that is optimized for an image size of 2K×2K. Uzun's 2-D DFT [22] supports multiple image sizes. However, since it requires transpose operations and runs at a lower frequency, its performance is

Table 4.4: Comparison of 2-D DFT implementations with respect to hardware configuration and performance.

		Eonic[18]	Lenart[19]	Uzun[22]	Dillon[21]	Proposed (BEE3$^{(1)}$)	Proposed (ML605$^{(2)}$)
	Technology	ASIC, 180nm	ASIC, 130nm	Virtex-E, 180nm	Virtex-II-Pro, 130nm	Virtex-5, 65nm	Virtex-6, 40nm
	External memory	Quad SDRAM	Dual SDRAM	Quad SRAM	Dual SRAM	Single SDRAM	Signal SDRAM
	Clock freq. (MHz)	128	250	26	120	100	100
	Data Width (bits)	2×32	2×16	2×16	2×8	2×32	2×32
Computation	128×128	–	–	2.4 (4.8)†	–	1.8	0.1
times for	256×256	–	–	10.1 (20.2)	–	6.1	0.4
different	512×512	–	–	28.6 (57.2)	–	24.9	1.6
Image sizes	1024×1024	21.0	–	76.9 (153.8)	–	102.6	6.5
(ms)	2048×2048	–	50.0 (100.0)	–	8.3 (33.3)	412.1	26.2

(1): The results are measured from BEE3 board; (2): The results are simulated based the hardware configuration of ML605.
(†: The computation times in the brackets are normalized for a data width of 2×32 bits.)

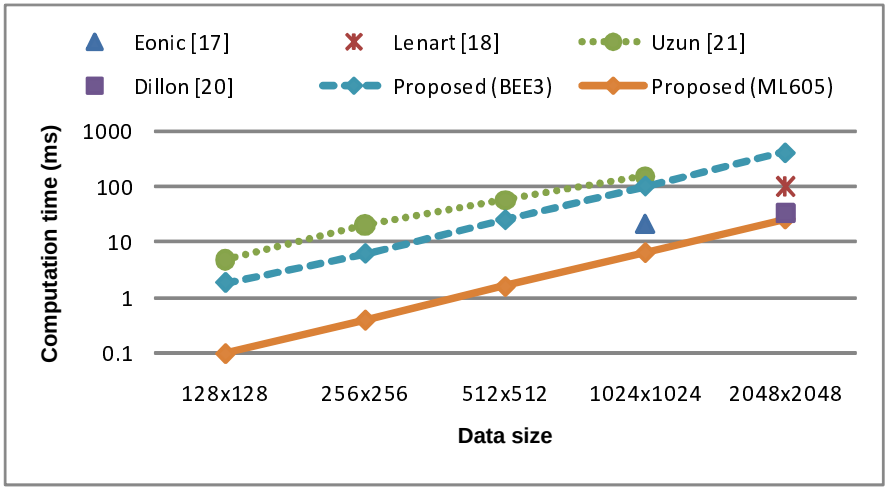


Figure 4.9: Comparison of computation times of 2-D DFT architectures. Note that the computation times are normalized for the same data width, 2×32 bits.

lower. Other competing solutions such as Lenart's [19] also requires transpose operations, and the one from Eonic [18] requires multiple SDRAMs (up to 4 banks). The performances of the 2-D DFT architectures have been illustrated in Fig. 4.9. We see that the proposed 2-D DFT on ML605 is the fastest implementation for different data sizes. Also, the straight lines in this log-scaled plot imply that the performances of the 2-D DFTs on BEE3 and ML605 are proportional to the data sizes.

To analyze the precision of the 2-D DFT, we first use MATLAB's 2-D FFT function to transform different sized images to the frequency domain. Then we use the spectral data to reconstruct the images using our 2-D DFT BEE3 implementation. In Table 4.5, we record the images' Signal-to-Noise-Ratio (SNR) and maximum reconstructive error, where SNR is defined as

$$SNR(dB) = 10\log_{10}\frac{P_{original\ image}}{P_{quantization\ noise}};$$

$P_{original\ image}$ is the power of original image (the ideal result), and $P_{quantization\ noise}$ is the power of quantization noise. We see that the SNR is around 130 dB, which is mainly due to Xilinx 1-D FFT IP, whose SNR is about 140dB [27]. Secondly, the maximum errors of all the images are fairly small. We conclude that the proposed architecture has high accuracy and can be used in most DFT-based image reconstruction applications.

Table 4.5: Measured accuracy of the proposed 2-D DFT.

Image	Size	SNR(dB)	Maximum norm error
Lena(1)	128×128	136.26	1.23e-4
Lena(1)	256×256	133.93	2.37e-4
Lena(1)	512×512	131.04	4.94e-4
MRI-brain(2)	1024×1024	128.02	1.21e-3
SAR(3)	2048×2048	126.59	2.07e-3

(1): From USC-SIPI Image Database [51].
(2): From University of Virginia Health System [52], and the image is resized to 1K×1K.
(3): From EUSAR06 [53], and the image is resized to 2K×2K.

Table 4.6: Measured computation time of 3-D DFT on BEE3.

Number of samples	Image size ($N_1 \times N_2 \times N_3$)	Total (ms)
2^{21}	$128 \times 128 \times 128$	348.24
2^{22}	$64 \times 256 \times 256$	757.21
	$256 \times 64 \times 256$	646.81
	$256 \times 256 \times 64$	697.09

4.5.2 3-D DFT

The performance of the 3-D DFT implementation is summarized in Table 4.6. The results presented here are measured values, but they match very well with estimated result derived from 2-D DFT measurements. In 128^3 DFT, for example, 2-D DFT on the 128 d_1-d_2 planes takes $128 \times 1.79 = 229.12$ms, and column DFTs on the 128 d_1-d_3 planes takes $128 \times 0.9 = 115.2$ms. Thus, the total estimated time of 128^3 3-D DFT is 344.32ms, which is pretty close to the measured result of 348.24ms. This means SDRAM's bandwidth efficiency in 2-D DFT is maintained in 3-D DFT. As in the 2-D case, the time-consuming transpose operations have been avoided. Note that, due to driver issues, we can currently only measure performance of data volumes up to 2^{22} samples on the BEE3 board.

Table 4.7 compares the performance of our architecture on the BEE3 and ML605 with Sasaki's architecture [23]. In [23], the computation kernel consists of three double-precision adders and two double-precision multipliers, which can implement a butterfly computation in two cycles. It utilizes the memory bandwidth efficiently and is computation-bound. To make the 3-D DFTs

Table 4.7: Comparison of 3-D DFT implementations with respect to hardware configuration and performance.

		Sasaki[23]	Proposed (BEE3$^{(1)}$)	Proposed (ML605$^{(2)}$)
	Technology	Virtex-II, 180nm	Virtex-5, 65nm	Virtex-6, 40nm
	External memory	Single SDRAM	Single SDRAM	Single SDRAM
	Clock freq. (MHz)	100	100	100
	Data width (bits)	2×64	2×32	2×32
Computation times	$128 \times 128 \times 128$	441	348 (696)†	22 (44)
for different	$256 \times 256 \times 256$	4,027	2,327 (4,654)	148 (296)
image sizes (ms)	$512 \times 512 \times 512$	–	19,241 (38,482)	1,223 (2,447)

(1): Except for $128 \times 128 \times 128$ 3-D FFT, the other results are extrapolated.
(2): The results are simulated based the hardware configuration of ML605.
(†: The computation times in the brackets are normalized for the same data width, 2×64 bits.)

comparable, we normalize our implementations to double precision. Since the implementations on BEE3 and ML605 are communication-bound, their computation times would be doubled due to 2x wider data width. The implementation on BEE3 is constrained by the data transfer as mentioned before. On ML605 board, however, the proposed architecture could be at least 10x faster than [23]. Since images larger than 1K×1K×1K with single precision require at least 8 GB memory, they cannot fit in the 4GB SO-DIMM, on the ML605 platform. Thus, we only simulate image sizes up to $512 \times 512 \times 512$.

4.6 Summary

An MD DFT IP has been proposed that is based on a decomposition algorithm which takes into account the burst access pattern of the SDRAM and the available FPGA resources. It does not require long stride memory accesses or transpose operations and is able to maintain the maximum SDRAM bandwidth throughout the computation. The MD DFT IP is automatically generated and the MD DFT IP generator integrated into the AlgoFLEX development platform. The input specifications such as image size, dimensionality, FPGA resources, memory bandwidth are input through the AlgoFLEX GUI, and the optimized HDL code is produced by the IP generator. The resulting architecture has been ported onto the BEE3 FPGA board and validated for different sized 2-D and 3-D data. To achieve higher performance, the architecture has been ported onto the high-end Xilinx ML605 FPGA board. Simulation results demonstrate the superior performance of these architectures compared to existing DFT implementations.

Chapter 5

Transpose-free SAR Imaging on FPGA Platform

5.1 Introduction

Synthetic Aperture Radar (SAR) has been widely used in military surveillance, environmental monitoring, and earth resource survey. From an airborne or a space-borne platform, a SAR system generates high resolution images covering large areas in all weather conditions, day or night.

The raw data collected by a SAR system is highly unfocused due to electromagnetic wave scattering and the relative motion between the radar and the earth surface. In fact, the unprocessed raw image looks like random noise. Focusing based on Fourier optics [54] is achieved using laser beams and lenses [55]. The lenses perform real-time two-dimensional Fourier Transform (2-D FT) on raw images projected from the film, and images are focused using diffraction gratings. Another set of lenses implement 2-D inverse Fourier Transform (2-D IFT) to convert the focused image back to space domain and record the final image on film. This optical process can be done in real-time but requires many high-quality lenses to be precisely aligned on a large optical bench. The process is hard to automate and requires a well-trained technician to maintain the image quality. In addition, the film, which records the output image, limits the dynamic range of the final image.

Today, digital SAR imaging processors have replaced the optical counterparts. Several algorithms have been developed for digital SAR imaging. These include range-Dopper algorithm (RDA) [56], chirp scaling algorithm (CSA) [57], and Omega-K algorithm (WKA) [58]. The key kernels of these algorithms are Discrete Fourier transform (DFT), interpolation, and convolution, all of which can easily be computed on PCs or workstations. Since the SAR imaging algorithms can be highly parallelized, they can be computed even more efficiently on modern GPUs, or on application-specific hardware, like ASIC or FPGA, with multiple customized processing elements (PEs).

The real bottleneck that impedes acceleration of SAR imaging is data transfer between the processing unit and the external memory. SAR images are typically very large, e.g. 4,096$\times$4,096, and must be stored in an external memory, which is usually a Synchronous Dynamic RAM (SDRAM).

SDRAM's transfer rate is slow compared to processors' computation speeds, and so the performance of SAR imaging systems today is dominated by the memory bandwidth.

Furthermore, SAR imaging algorithms need to perform computations along row and column directions of a 2-D image several times. Because SDRAM only favors row-wise burst access, most SAR imaging processors [59] [60] [61] [62] need to perform matrix transpose before column-wise operations. This is done by transferring column-wise data from the SDRAM to the chip, realigning, and storing back to the memory. Then, another transpose operation is required before the next row-operation. Typically, all SAR imaging algorithms require multiple matrix transposes. Since the memory transfer rate is the bottleneck, multiple transpose operations make the performance of a SAR processor even worse.

To eliminate matrix transpose, the method in [63] stores SAR data into a multi-chip SDRAM array and takes advantage of the multi-banking memory organization to reduce the overhead of accessing column-wise data. However, this design does not support general SDRAM DIMMs, and significant customization had to be done. Another way to eliminate transpose operations is to increase the locality of data along column direction in a SDRAM DIMM. This is done in [64] and [65], where the column-wise data is re-mapped into a physical page of SDRAM to increase the access efficiency. Such a method achieves 80% of SDRAM's peak rate for both row-wise and column-wise access. However, the data re-mapping before the SAR imaging process takes extra time.

In this chapter, we propose transpose-free SAR imaging flows for RDA and CSA. The corresponding implementations have superior timing performance, since there are no transpose operations, and efficient memory bandwidth utilization is efficiently utilized. The flows are mapped to a unified architecture which is implemented on the Xilinx ML605 platform. Simulation results on the ML605 platform show that the RDA and CSA computations with data size 4096×4096 can be completed in 323ms and 162ms respectively. This implementation outperforms existing SAR image accelerators, including FPGA- and GPU-based solutions [65, 66, 67].

The rest of the chapter is organized as follows. A brief description of RDA and CSA is given in Section 5.2. In Section 5.3, the transpose-free imaging flows for RDA and CSA are

proposed. A unified FPGA architecture for RDA and CSA is proposed in Section 5.4, and evaluated in Section 5.5. The chapter is concluded in Section 5.6.

5.2 Background

In this section, we describe the traditional RDA and CSA process flows and explain their drawbacks.

5.2.1 Range-Doppler Algorithm

In SAR imaging, range-Doppler algorithm (RDA) is the most popular imaging method. Its process flow is depicted in Fig. 5.1 and described as follows.

Phase 1: Range Compression

A SAR system transmits long-duration linear frequency modulation (FM) pulses, so that the pulse has a lower peak power. After the radar receives the reflected pulse, matched filtering is done to compress the raw data and form a narrow pulse. Matched filtering is usually done in frequency domain and on each range line. Range compression consists of three steps:

• Step 1. Range DFT: DFT transforms the data into frequency domain along each range line.

• Step 2. Matched filtering: A multiplication implements matched filtering in frequency domain.

• Step 3. Range IDFT: IDFT transforms the data back to space domain.

Phase 2: Azimuth Compression

In the second phase, the image is focused along azimuth direction. It consists of the following steps:

• Step 4. Azimuth DFT: The data is transformed to range-Doppler domain first.

• Step 5. Range cell migration correction (RCMC): In range-Doppler domain, the trajectory formed by a target needs to be straightened before azimuth compression. Along the range direction, interpolation is performed to obtain the peak values of the trajectory and mapped onto a straight line.

• Step 6. Matched filtering: A phase multiplication is performed to implement azimuth compression along each azimuth line.

• Step 7. Azimuth IDFT: The compressed data is transformed back to the space domain.

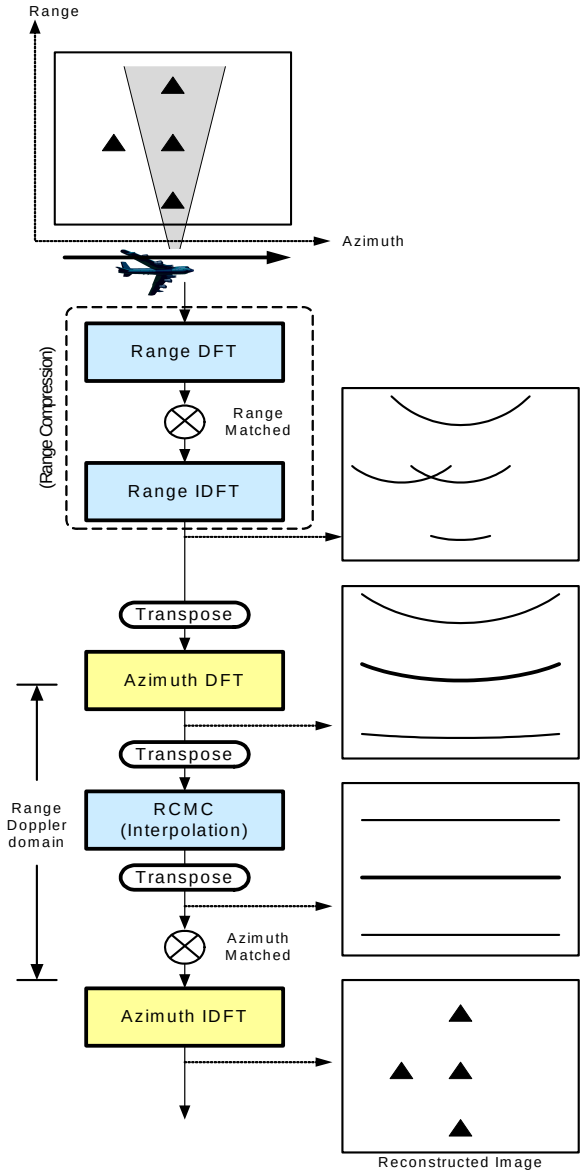


Figure 5.1: Traditional process flow for Range Doppler algorithm (RDA) [56].

The main downside of RDA is that the interpolation for RCMC is computation-intensive. In addition, it requires multiple transpose operations, as shown in Fig. 5.1. After range compression, a transpose operation is required before Azimuth DFT. Furthermore, since RCMC is implemented by interpolations along range direction, we need two additional transpose operations before and after RCMC. Thus, RDA requires three transpose operations which results in significant timing overhead.

5.2.2 Chirp Scaling Algorithm

The chirp scaling algorithm (CSA) avoids the interpolation for RCMC and is computationally less intensive compared to RDA. CSA requires only complex multiplications and Fourier transforms, as shown in the operation flow in Fig. 5.2. The sequence of steps is described as follows:

• Step 1. Azimuth DFT: Azimuth DFT is first performed in order to transform the data into range/Doppler domain.

• Step 2. Chirp scaling: Then, in range-Doppler domain, chirp scaling, a quadratic phase function, is applied to equalize the curvatures of the curves at different ranges.

• Step 3. Range DFT: It is performed to transform the data into the 2-D frequency domain.

• Step 4. Bulk RCMC: A phase multiplication is performed with a reference function, which includes range compression, secondary range compression, and bulk RCMC in one operation.

• Step 5. Range IDFT: Range IDFT transforms the data back to range/Doppler domain.

• Step 6. Azimuth Compression: A phase multiplication is performed to implement both azimuth compression and a phase correction, compensating the chirp scaling applied in Step 1.

• Step 7. Azimuth IDFT: The compressed data is transformed back to the SAR image domain.

Although interpolation is avoided in CSA, it still requires two matrix transposes between Range and Azimuth DFT/IDFT.

5.2.3 Problem Statement

SAR raw data is so large that it needs to be stored in an external memory. Dynamic memory, such as synchronous dynamic RAM (SDRAM), is usually adopted due to its large storage density, high performance, and low cost. However, SDRAM only favors burst access to consecutive addresses. For example, if the 2-D SAR data is stored in range-major order, i.e. the range-wise data is stored in consecutive address, the data can be retrieved from the SDRAM very efficiently along the range direction. However, accessing the azimuth-wise data is a lot slower, because the data are not in adjacent addresses any more. This is why both RDA and CSA require multiple transpose operations to keep data in consecutive location in the memory throughout the computations.

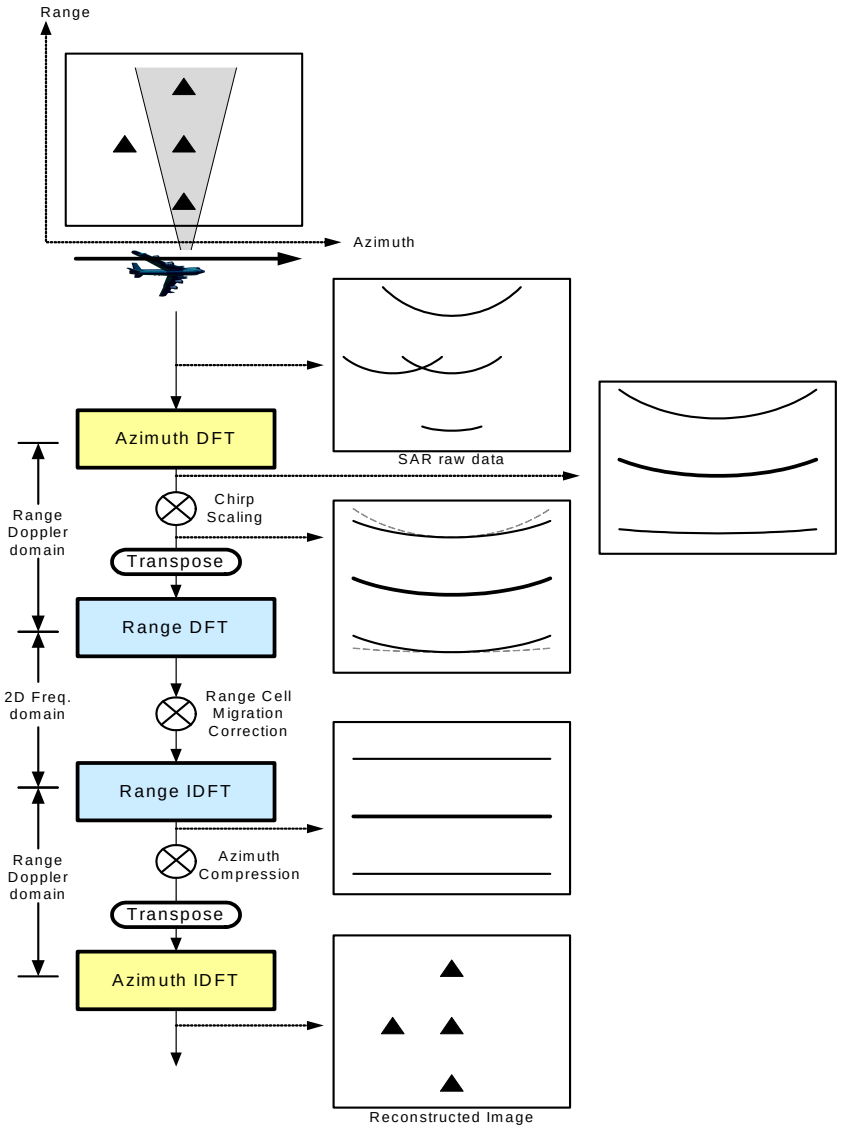


Figure 5.2: Traditional flow of chirp scaling algorithm (CSA) [57].

However, transpose operations hog the memory bandwidth and keep it busy. A consequence of this is that the computation units have to wait for the memory data and are idle longer. Thus, transpose operations need to be removed to improve the timing performance of SAR imaging implementations.

5.3 Transpose-free SAR Imaging Methods

To avoid the transpose operations and exploit the SDRAM access characteristics matrix transposes, we utilize three access patterns as shown in Fig. 5.3:

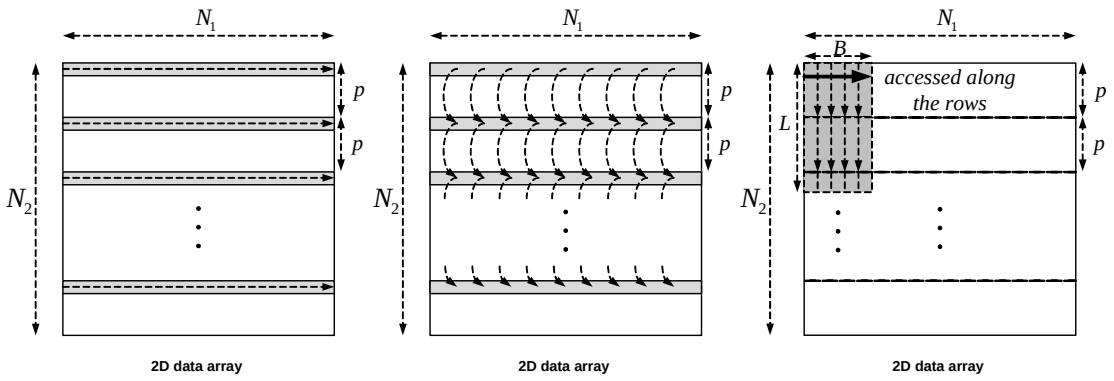
• **Pattern 1-a. Row Access:** Rows spaced p rows apart are accessed. It supports row-wise DFT and interpolation.

• **Pattern 1-b. Column Stride Access:** Data is processed across the rows read with Pattern 1-a. This pattern is required for q-point Column Stride DFT [68].

• **Pattern 2. Column Local Access**: Sub-columns are accessed for p-point Column Local DFT [68].

Note that: (1) The accessed data of Patterns 1-a and 1-b are from the same set of addresses in the memory. Therefore, if the adjacent two steps require these two access patterns, they can be merged into one step; (2) Patterns 1-b and 2 together can be used to compute full-length ($N_2(= p \times q)$-point) Column DFT.

The proposed method is based on decomposing the lengthy column DFT into shorter ones, so that multiple more adjacent sub-columns of data can be read from or written to the SDRAM. Consequently, long row-wise bursts can be formed, and peak performance of the SDRAM can be sustained throughout the computation. Most importantly, this reorganization ensures that transpose operations are no longer required.



(a) Pattern 1-a. Row Access (b) Pattern 1-b. Column Stride Access (c) Pattern 2. Column Local Access

Figure 5.3: Data access patterns of the transpose-free 2-D DFT.

5.3.1 Transpose-free RDA Processor

Based on the data access patterns, we propose a new RDA flow, which is illustrated in Fig. 5.4. Note that there are several storage formats of SAR raw data; they depend on both type of radar

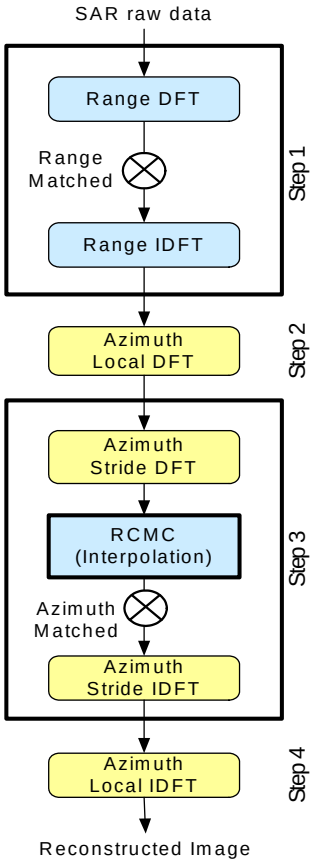


Figure 5.4: Proposed flow of range-Doppler algorithm.

as well as the host system. Here, we assume that the SAR raw data is preprocessed and stored in the external memory in range-major order, that is, the data along the range direction are stored in adjacent addresses in SDRAM. This helps in accessing memory data efficiently during computation of Range DFT and IDFT in Step 1. To avoid transpose operations, Azimuth DFT is decomposed into Local DFT in Step 2 and Stride DFT in Step 3. To perform Azimuth Stride DFT in Step 3, the entire row along range direction needs to be loaded onto the local memory. After the DFT, RCMC can be computed along the rows in the same step. To transform the data back to the space domain, the Azimuth Stride and Local IDFT are performed. Note that Azimuth Stride IDFT is computed in Step 3 to avoid extra memory transactions. Compared to the original flow, the new flow still requires 4 steps but eliminates the 3 transpose operations.

Table 5.1 compares SDRAM bandwidths of traditional and proposed RDA flows. Note that every step of the two flows needs to access the whole matrix, i.e. N_1N_2 pixels, once. Since the

Table 5.1: Comparison of RDA implementations.

Step	Traditional	Proposed
1	Range FFT + Range IFFT	Range DFT + Range IDFT
2	[Transpose]	Azimuth Local DFT
3	Azimuth DFT	Range Stride DFT + RCMC + Azimuth IDFT
4	[Transpose]	Azimuth Local IDFT
5	RCMC	
6	[Transpose]	
7	Azimuth IDFT	
SDRAM Bandwidth (# of Pixels Accessed)	$7N_1N_2$	$4N_1N_2$

traditional flow has 7 steps, overall $7N_1N_2$ pixels are accessed during the computation. As for the proposed flow, the number of accessed pixels is $4N_1N_2$. Thus, the SDRAM bandwidth is saved by about 42%. This makes the proposed flow faster and also reduces the power consumption of the SDRAM.

5.3.2 Transpose-free CSA Processor

We assume that the SAR raw data is preprocessed and stored in the SDRAM in azimuth-major order. With this data organization, Azimuth DFT or IDFT achieves the highest performance because row data can be retrieved efficiently using long row-wise burst access along rows in SDRAM. To avoid transpose, we decompose the Range (I)DFT into (1) Range Stride(I)DFT and (2) Range Local (I)DFT, and obtain a new process CSA flow, as illustrated in Fig. 5.5. There are only three major steps now:

Step 1. Since Azimuth DFT can access data with Pattern 1-a, and Range Stride DFT needs to access data with Pattern 1-b, they can be merged into one step. In addition, chirp scaling function can be implemented with a multiplication right after Azimuth DFT.

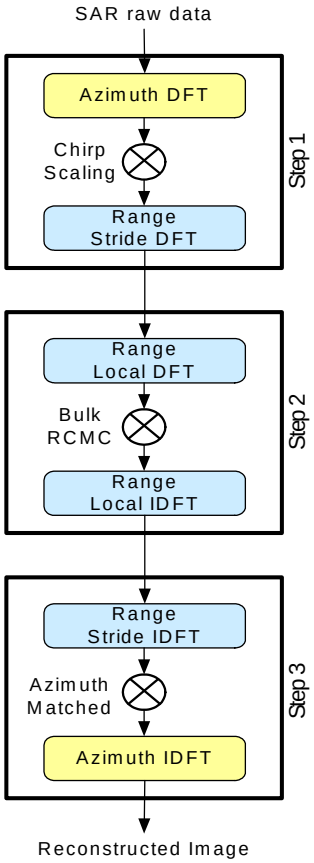


Figure 5.5: Proposed flow for chirp scaling algorithm.

Step 2. Range DFT and IDFT are computed on the same set of data in every iteration, so they are merged into one step. The RCMC function is just a multiplication after Range local DFT.

Step 3. This step is just an inverse operation of Step 1, except for the Azimuth compression function. Range Stride IDFT accesses data with Pattern 1-b, while Azimuth IDFT accesses data with Pattern 1-a, and thus they can be merged into one step.

To sum up, the proposed CSA procedure reduces the number of steps from four to three, while avoiding the two extra transpose operations.

A comparison of the traditional and proposed CSA flows is given in Table 5.2. The traditional CSA requires two additional transpose operations. Note Range DFT and Range IDFT can be computed within the same step, i.e. Step 3, because no transpose is required between these two

Table 5.2: Comparison of CSA implementations.

Step	Traditional	Proposed
1	Azimuth DFT	Azimuth DFT + Range Stride DFT
2	[Transpose]	Range Local DFT + Range Local IDFT
3	Range DFT + Range IDFT	Range Stride IDFT + Azimuth IDFT
4	[Transpose]	
5	Azimuth IDFT	
SDRAM Bandwidth (# of Pixels Accessed)	$5N_1N_2$	$3N_1N_2$

computations. Thus, $5N_1N_2$ pixels need to be accessed in the traditional flow, while the proposed method only accesses $3N_1N_2$ pixels. The bandwidth of SDRAM is saved by 40%.

In short, by taking advantage of the proposed access patterns, we can improve the memory efficiency of the two mainstream SAR imaging algorithms. The computation times can be shortened, and power consumption of the memory can be reduced, too.

5.4 Unified Architecture for Proposed RDA/CSA Flows

In this section, we propose a unified architecture for computing both RDA and CSA. From the proposed flows, there are three basic kernels that need to be implemented: (1) Forward and Inverse Fourier Transform, (2) Phase factor (Multiplication), and (3) Interpolation. While RDA requires all three, CSA needs only the first two. We choose FPGA as our target platform, for ease of prototyping. The unified architecture for SAR imaging is shown in Fig. 5.6. It consists of an FPGA for accelerating SAR computation and an external SDRAM for storing the SAR data. The main components are described in details as follows:

PE Array: Processing Element (PE) Array contains one or multiple PEs. Each PE consists of an FFT IP, an interpolator, and a multiplier to support RDA/CSA computations.

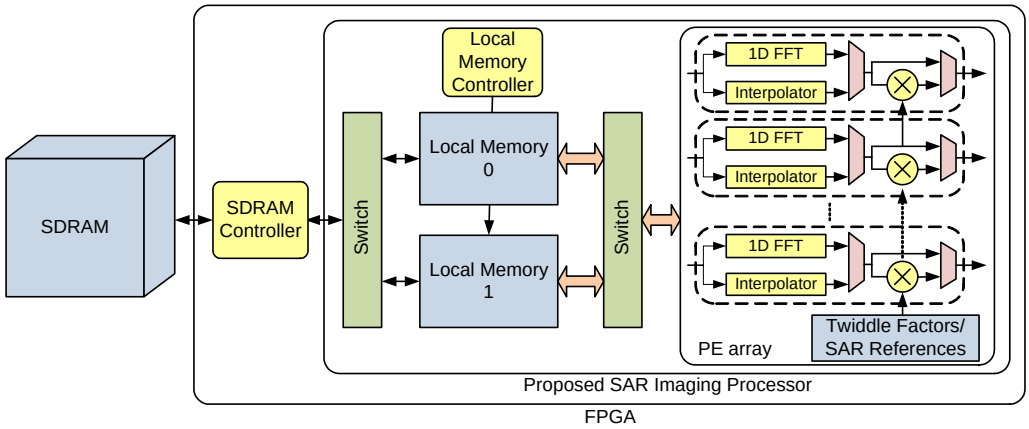


Figure 5.6: Proposed architecture for RDA and CSA imaging.

- FFT IP: The FFT IP should be able to support all DFT/IDFT with different sizes in the proposed flows of RDA/CSA. To satisfy the requirements, we adopt Xilinx's streaming FFT IP [27]. The size of the FFT IP can be reconfigured dynamically, and it has a very high throughput. In addition, the FFT IP is based on block floating-point (BFP) implementation and can achieve high dynamic range. Its I/O is converted to single-precision floating-point format.

- Interpolator: The single-precision interpolator supports the computation of RCMC in RDA and can be removed if only CSA is to be supported. It is a pipelined 8-tap FIR filter with programmable coefficients.

- Multiplier: The single-precision multiplier implements the multiplications with twiddle factors and reference phase functions.

- Local memory for twiddle factors/reference phases: These coefficients are pre-computed and stored in a local memory. To save the on-chip memory, only $N/8$ twiddle factors from 0 through $\pi/4$ are stored, where N is the largest size that the FFT IP supports. These twiddle factors can be used to regenerate other required twiddle factors on the fly. For RDA, the range/azimuth compression coefficients, are also pre-computed and saved in the memory. As for CSA, the phase factors can be separated into functions of range, $\Phi(r)$, and functions of azimuth, $\Phi(a)$, which are pre-computed and stored in the local memory. $\Phi(r)$ and $\Phi(a)$

can be used to generate phase factors, $\Phi(r,a)$, on the fly. Then, a Φ-to-$e^{j\Phi}$ lookup table is used to convert the phases into complex numbers for the computations.

Note that RDA and CSA can be highly parallelized, and the computations on multiple range and azimuth lines can be executed simultaneously. Therefore, if we have N PEs, we can accelerate the computations N times. However, the number of PEs is limited by the available hardware resource.

Local Memory on FPGA: Two local memories form a ping-pong buffer to overlap the computation and communications between the FPGA and SDRAM. Each local memory is of size S pixels. It is divided into r banks of Xilinx Block RAMs (BRAMs), so that r PEs can access the data in the local memory at the same time. An addressing scheme [68] is adopted to support conflict-free memory access for all row- and column-wise computations. For example, let the r banks of BRAMs be named as Bank 0-$r-1$. For the row operation, the SDRAM controller fetches a row from the SDRAM and stores them into the banks starting from Bank 0. Then, the SDRAM controller fetches the another row starting from the pth row and stores them starting from Bank 1, and so on. Such a storage scheme guarantee that r PEs can simultaneously access the data from different banks. The same addressing scheme also works for column-wise access.

SDRAM: An SDRAM controller fetches the input data from SDRAM and sends it to the local memory. The processing elements (PE) read this data, compute Range/Azimuth (I)DFT and twiddle multiplications, and store the results back to the local memory. Finally, the SDRAM controller reads these results from the local memory and stores them back to the SDRAM.

In this design, we offer two configurations for the user: (1) Dual-mode: This configuration supports both RDA and CSA. It offers flexibility when the user want to accelerate RDA or CSA computations without reprogramming the FPGA; (2) CSA-only: Only CSA is supported with this configuration. By removing the interpolator from the PE, more PEs can fit on the FPGA and achieve very high performance.

5.5 Evaluation

5.5.1 Resource

Our target platform is the Xilinx ML605 FPGA board, which is equipped with a Xilinx Virtex-6 XC6VLX240T FPGA and a DDR3-800 SDRAM module. In Table 5.3, we list the FPGA resources required by the major components. To support up to 4096×4096 data size, we adopt a 4096-point Xilinx FFT IP in the PE. It is programmable and can also support smaller FFT sizes from 2048- to 8-point, that would be required to calculate stride/local FFT/IFFT in the proposed flows. The FFT IP requires 60 DSP48E1 slices, but the 8-tap single-precision interpolator requires 80 DSP48E1s! If we only need to support CSA, we can remove the interpolator. In this case, the total number of the DSP480E1s would be only 78, and hence, we can put more PEs on the FPGA.

Each local memory of the ping-pong buffer can accomodate 32,768 complex single-precision data, which corresponds to 240 BRAMs. To support on-line phase regeneration of CSA, we use an 8K-word Φ-to-$e^{j\Phi}$ lookup table. Each word is 18-bit, so that it can fit in the 18-bit wide BRAMs. In each PE, the lookup table requires 14 BRAMs.

Table 5.3: Number of dedicated multipliers required by one single PE, on a Xilinx Virtex-6 FPGA (Data format: Single precision).

	4096-point FFT	Twiddle Multiplier	Phase Generator for CSA	8-tap Interpolator	Ping-Pong Buffer
DSP48E1s	60	16	2	80	–
BRAMs	36	2	14	–	240

Table 5.4 compares the occupied resources of both Dual-mode and CSA-only configurations. Since the Dual-mode configuration includes an interpolator in the PE, only 4 PEs can fit on the Virtex-6 FPGA. In CSA-only configuration, there is no interpolator and so overall 8 PEs can be implemented on the FPGA.

5.5.2 Performance

The SAR processor is clocked at 100MHz, and our customized memory controller can achieve 80% of peak transfer rate of the DDR3-800 SDRAM module. The computation times for the two configurations for three data sizes are listed in Table 5.5. Since the dual-mode configuration only has

Table 5.4: Hardware resource utilization of the proposed SAR imaging processor on a Xilinx Virtex-6 XC6VLX240T FPGA (Data format: Single precision).

Configuration	# of PEs	DSP48E1s	BRAMs	Slices
Dual mode (with interpolators)	4	81% (624/768)	54% (448/832)	47% (17,701/37,680)
CSA only (no interpolator)	8	81% (624/768)	79% (656/832)	78% (29,385/37,680)

four PEs, in most steps, the computations take longer than the data transcations between the FPGA and SDRAM. As a result, CSA takes about 60% longer time on this configuration, compared to CSA-only configuration which only supports CSA and houses 8 PEs. The timing performance of the CSA-only configuration is only constrained by the SDRAM bandwidth, while that of the dual mode configuration is constrained by the on-chip resources. For RDA with dual-mode configuration, Step 2 and Step 4 only have one operation respectively, so they can be computed faster than the other steps. On the contrary, Step 3 of RDA has three operations, including the computation-intensive interpolation, and therefore, this step takes the longest time. For both configurations, the computation time is proportional to the image size.

Table 5.5: Simulated computation times (ms) of the proposed SAR imaging processor.

Data Size	1024×1024			2048×2048			4096×4096		
# of PEs	4 (Dual Mode)		8 (CSA only)	4 (Dual Mode)		8 (CSA only)	4 (Dual Mode)		8 (CSA only)
Algorithm	RDA	CSA	CSA	RDA	CSA	CSA	RDA	CSA	CSA
Step 1	5.45	5.43	3.39	20.78	20.95	13.09	85.19	84.55	52.78
Step 2	3.37	5.49	3.90	13.20	20.86	14.29	53.81	86.78	55.90
Step 3	8.30	5.49	3.36	31.29	21.03	13.15	129.78	85.21	53.10
Step 4	3.31	–	–	13.52	–	–	54.21	–	–
Total	20.43	16.41	10.65	78.80	62.84	40.53	322.99	256.54	161.78

In Table 5.6, we compare our implementation with other SAR imaging processors. Zhou [65] also proposed a transpose-free FPGA design for CSA computation. By reordering the SAR data, this implementation minimizes the frequency of opening/closing physical pages of memory, and hence, sustains the performance of the SDRAM. However, it still requires extra time for reordering the data, while our design needs no data rearrangement. As for RDA, we compare the proposed design with three GPU solutions [66, 67, 69]. While GPUs have numerous parallel computation engines running at very high clock rates, it takes extra time for the host computer to generate sufficient number of GPU threads, so that the computational power of the GPUs can be

highly or fully utilized. Besides, memory transaction is still a bottleneck for GPUs based on the experiments in [69]. With the proposed transpose-free procedure, our design is able to outperform the GPU-based accelerators in terms of timing. In addition, since the power consumption of the Virtex-6 FPGA is no more than 15 Watts, while the GPUs usually consume more than 70 Watts, our FPGA-based SAR processor is highly power-efficient compared to the GPU-based solutions.

5.5.3 Accuracy

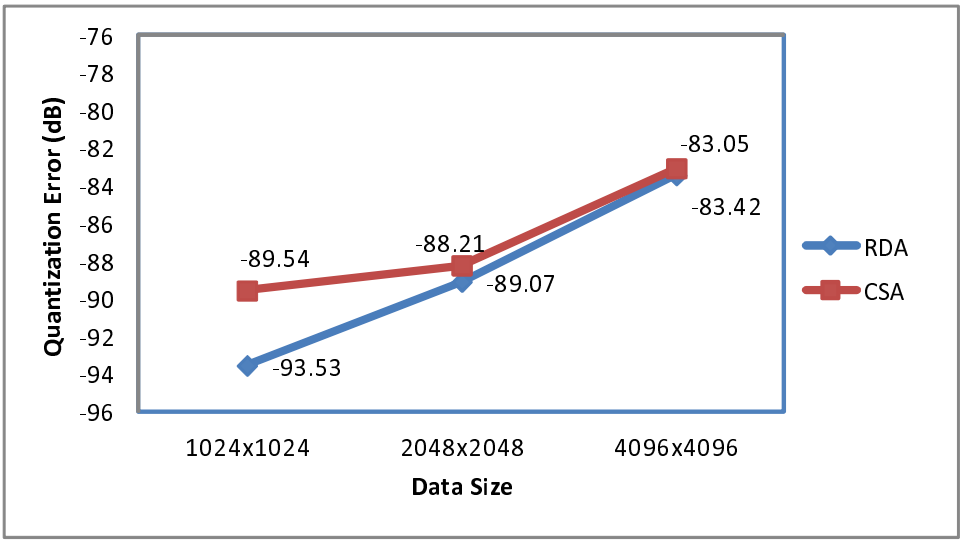


Figure 5.7: Image focused by the proposed SAR imaging processor.

We use MATLAB to generate synthesized RAW data to verify the function of the proposed SAR imaging processor. The reconstructed images are compared with the ideal outputs generated by MATLAB, and their differences are shown in Fig. 5.7. For the proposed RDA implementation, the main quantization noise source is the block floating-point FFT IP. The larger the data size, the larger the noise it generates. For the CSA implementation, the phase factor generators cause additional noise, whose noise level is about -90dB based on our simulations. The noise level is larger than that of the 1024-point FFT, which is only -95dB. This is why the overall noise level of CSA is higher than that of the RDA, when the data size is 1024×1024. Note that the noise levels of 2048- and 4096-point FFT are -89dB and -85dB respectively, which are higher than that of the phase generators. As a consequence, the FFT IP dominates the accuracy performance when data size is larger than 1024×1024. As mentioned above, the Φ-to-$e^{j\Phi}$ lookup table has 8K words.

Table 5.6: Comparison on the performance of SAR imaging accelerators.

		Liu[66]	Biseglie[69]	Ning[67]	Zhou[65]	Proposed (4-PE)		Proposed (8-PE)
	Device	nVidia Quadro FX3700	nVidia Tesla C1060	nVidia Tesla C2050	Altera Stratix-II	Xilinx Virtex-6		
	Clock freq.	500 MHz	1.30 GHz	1.15 GHz	128 MHz	100 MHz		
	External memory	GDDR3	GDDR3	GDDR5	SDRAM PC-100	DDR3-800 SDRAM		
	Data format	Single precision	Single precision	Double precision	Single precision	Single precision		
	Algorithm	RDA	RDA	RDA	CSA	RDA	CSA	CSA
Computation times for different image sizes (ms)	1024×1024	–	–	498.00 (249.00)†	60.87	20.43	16.41	10.65
	2048×2048	13.00	–	–	–	78.80	62.84	40.53
	4096×4096	392.00	4,400.00	832.00 (416.00)	1070.10	322.99	256.54	161.78

(†: The computation times in the brackets are normalized for single-precision data format.)

If we double the words in the table such that the $e^{j\Phi}$ can be approximated better, its noise level can be reduced by about 6dB. As a result, the accuracy performance of the CSA would be very close to that of the RDA, when the data size is 1024×1024. However, a 16K-word lookup table would require too many BRAMs and the design would not fit on the FPGA. Also, the quantization error generated by both configurations is fairly small, and so increaing the table size carries little merit. Overall, the proposed architecture has high accuracy and can be used for SAR imaging applications.

5.6 Summary

In this chapter, we propose transpose-free flows for two popular SAR imaging algorithms, RDA and CSA. This is done by reorganizing the flows so that the row-wise burst access pattern favored by SDRAM can be utilized. Thus data transactions with the external memory are reduced, and the high transfer rate of SDRAM can be sustained. In addition, the computation units are no longer idle waiting for memory data. The proposed flows are mapped to a unified architecture that can support RDA and CSA and implemented onto a ML605 platform. Two configurations are supported: Dual-mode which supports both RDA and CSA, and CSA-only configuration that achieves very high speed on CSA computation. The simulation results show that the proposed design has superior performance compared to existing FPGA and GPU-based SAR image accelerators.

Chapter 6

Conclusion

MD DSP algorithms are not only computation-intensive but also demand high memory bandwidth. Today's processors have significant computational power but the available memory bandwidth is not sufficient for real-time implementations of many of the MD DSP algorithms. In this work, we show how algorithm transformations can be used to circumvent this bottleneck. We focus on MD DFT, a widely used MD DSP algorithm. We choose FPGA as the target platform, because of its flexibility. Moreover, today's FPGA have abundant on-chip resources, making it very suitable for MD DSP applications.

The first problem that we addressed is the development of a high-performance 2-D DFT IP for large-sized data. The 2-D DFT computation is decomposed into computations on smaller sub-blocks, so that it is sufficient to load only part of the large data onto the FPGA at a time. The proposed architecture exploits the parallelism exposed by this decomposition and, in addition, exploits the row-wise burst access pattern of the external memory. We show through experimental results that the proposed design outperforms existing 2-D DFT solutions. In order to automate the process, we developed an IP generator, which can generate optimized 2-D DFT implementations based on the hardware resource of the FPGA platforms.

The second problem involved development of a flexible MD DFT IP to support transpose-free 2-D and 3-D DFT for different data shapes and sizes. Transpose operations are time consuming and degrade the overall system time performance. To avoid transpose operations, strips of data are accessed from the SDRAM onto the local memory, where the width of the strips match the row-wise burst size. As a result, the maximum memory bandwidth can be sustained throughout the 2-D/3-D DFT computation. Our experiments show that the proposed architecture has superior performance to other existing FPGA-based 2-D/3-D DFT processors. The corresponding automated IP generator is embedded in a full-fledged FPGA development framework, where the user can configure the DFT using a user-friendly graphical interface, and generate, synthesize, and port the optimized IP onto the FPGA.

The third problem that we addressed is development of an FPGA-based architecture for SAR imaging. Multiple transpose operations are required in conventional range-Doppler (RDA) and chirp scaling (CSA) algorithms used in SAR imaging. We develop transpose-free imaging flows for both RDA and CSA. These utilize the memory access patterns derived for MD DFT IP and result in high memory bandwidth utilization. The flows are then mapped to a unified architecture which can compute both RDA and CSA. The simulation results shows that it has good accuracy and higher timing performance compared with existing FPGA- and GPU-based solutions.

Future Work

The main concept behind the MD DFT IP is efficient memory bandwidth utilization. Specifically, the DFT algorithm steps were reorganized to take advantage of the burst access of the SDRAM. This concept can be extended to other MD DSP algorithms that also access SDRAM data with different stride. One example of such an algorithm is pyramid generation, which is needed in multiple image processing and computer vision algorithms. In pyramid generation, the output data of every level is decimated. Thus, the relevant data is not generated in continuous cycles, and if we directly store the data back to an SDRAM, the data transfer rate would be seriously degraded. To avoid this, the local memory on the FPGA should be used to collect sufficient amount of data, so that long data bursts can be formed and bandwidth well-utilized.

Another part that we can improve on are the automation tools for generating the IPs. The automation tools were fully customized for the MD DFT to increase the efficiency of the hardware implementations. As a result, new users would take extra effort to learn the tools and configure them for their own working environment. For higher portability, the plan is to integrate IPs into some widely used commercial software, such as Simulink or LabVIEW. These tools also support well-developed hardware infrastructure for FPGA, and many users in DSP area are familiar with these tools. Unfortunately, the bit stream generated by these tools does not result in an efficient implementation. More effort needs to be directed to make the bitstreams competitive with those generated by customized tools.

Finally, for SAR imaging, while the proposed unified architecture supports both RDA and CSA, other advanced imaging algorithms, such as Extended CSA [70] or Omega-K algorithm [71] should also be considered. Most of these algorithms also require FFT, phase multiplication, and interpolation, and hence these components can be easily reutilized. More importantly, memory access patterns should be studied and the design flow reorganized to utilize the memory bandwidth. Only then can SAR imaging processors truly achieve high performance.

Reference

[1] J. Backus, "ACM Turing Award Lecture," 1977.

[2] I. G. Cumming and F. H. Wong, *Digital Processing Of Synthetic Aperture Radar Data: Algorithms And Implementation*. Boston: Artech House, 2005.

[3] A. Souza and R. Senn, "Model-based super-resolution for MRI," in *Proc. 30th Annual Int. of the IEEE Engineering in Medicine and Biology Society*, August 2008, pp. 430 –434.

[4] M. Frigo and S. G. Johnson, "FFTW: An adaptive software of the FFT," in *Proc. IEEE Int. Conference on Acoustics, Speech and Signal Processing*, vol. 3, 1998, pp. 1381–1384.

[5] ——, "The design and Implementation of FFTW3," *Proceedings of the IEEE, special issue on Program Generation, Optimization, and Adaptation Computer Physics Communications*, vol. 93, no. 2, pp. 216–231, 2005.

[6] M. Püschel, J. M. F. Moura, J. Johnson, D. Padua, M. Veloso, B. Singer, J. Xiong, F. Franchetti, A. Gacic, Y. Voronenko, K. Chen, R. W. Johnson, and N. Rizzolo, "SPIRAL: Code generation for DSP transforms," *Proceedings of the IEEE, special issue on "Program Generation, Optimization, and Adaptation"*, vol. 93, no. 2, pp. 232– 275, 2005.

[7] J. Johnson and X. Xu, "Generating symmetric DFTs and equivariant FFT algorithms," in *Proc. the 2007 Int. Symposium on Symbolic and Algebraic Computation*, 2007, pp. 195–202.

[8] "Intel Math Kernel Library (MKL)," http://software.intel.com/en-us/intel-mkl/.

[9] "Intel Integrated Performance Primitives (IPP)," http://software.intel.com/en-us/intel-ipp/.

[10] M. Eleftheriou, B. G. Fitch, A. Rayshubskiy, T. J. C. Ward, and R. S. Germain, "Scalable Framework for 3D FFTs on the Blue Gene/l Supercomputer: Implementation and Early Performance Measurements," *IBM Journal of Research and Development*, vol. 49, pp. 457–464, 2005.

[11] B. Fang, Y. Deng, and G. Martyna, "Performance of the 3D FFT on the 6D Network Torus QCDOC Parallel Supercomputer," *Computer Physics Communications*, vol. 176(8), pp. 531–538, April 2007.

[12] Y.-W. Lin and C.-Y. Lee, "Design of an FFT/IFFT processor for MIMO OFDM systems," *IEEE Trans. Circuits and Systems I: Regular Papers*, vol. 54, no. 4, pp. 807–815, 2007.

[13] Y.-N. Chang and K. K. Parhi, "An efficient pipelined FFT architecture," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 50, no. 6, pp. 322–325, 2003.

[14] Y. Chen, Y.-C. Tsao, Y.-W. Lin, C.-H. Lin, and C.-Y. Lee, "An indexed-scaling pipelined FFT processor for OFDM-based WPAN applications," *IEEE Trans. Circuits and Systems II: Analog and Digital Signal Processing*, vol. 55, no. 2, pp. 146–150, 2008.

[15] B. Baas, "A low-power, high-performance, 1024-point FFT processor," *IEEE Journal of Solid-state Circuits*, vol. 34, no. 3, pp. 380–387, March 1999.

[16] W.-C. Yeh and C.-W. Jen, "High-speed and low-power split-radix FFT," *IEEE Trans. Signal Processing*, vol. 51, pp. 864–874, March 2003.

[17] Y.-W. Lin, H.-Y. Liu, and C.-Y. Lee, "A 1-GS/s FFT/IFFT processor for UWB applications," *IEEE Journal of Solid-State Circuits*, vol. 40, pp. 1726–1735, Aug. 2005.

[18] "PowerFFT ASIC," http://www.eonic.com/index.asp?item=32.

[19] T. Lenart, M. Gustafsson, and V. Öwall, "A hardware acceleration platform for digital holographic imaging," *Journal of Signal Processing System*, vol. 52, no. 3, pp. 297–311, September, 2008.

[20] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "Formal datapath representation and manipulation for implementing DSP transforms," in *Design Automation Conference*, 2008, pp. 385–390.

[21] T. Dillon, "Two Virtex-II FPGAs deliver fastest, cheapest, best high-performance image processing system," *Xilinx Xcell Journal*, vol. 41, pp. 70–73, 2001.

[22] I. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of Fast Fourier Transforms for real-time signal and image processing," in *IEE Proc. Vision, Image, and Signal Processing*, vol. 152, no. 3, June 2005, pp. 283–296.

[23] T. Sasaki, K. Betsuyaku, T. Higuchi, and U. Nagashima, "Reconfigurable 3D-FFT Processor for the Car-Parrinello Method," *The Journal of Computer Chemistry, Japan*, vol. 4, no. 4, pp. 147–154, 2004.

[24] P. D'Alberto, P. A. Milder, A. Sandryhaila, F. Franchetti, J. C. Hoe, J. M. F. Moura, M. Püschel, and J. Johnson, "Generating fpga accelerated DFT libraries," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007, pp. 173–184.

[25] P. Kumhom, J. Johnson, and P. Nagvajara, "Design, optimization, and implementation of a universal FFT processor," in *Proc. 13th Annual IEEE Int. ASIC/SOC Conference, 2000.* IEEE, 2000, pp. 182–186.

[26] J. W. Cooley and J. W. Tukey, "An algorithm for the machine computation of complex Fourier series," *Mathematics of Computation*, vol. 19, pp. 297–301, 1965.

[27] "Xilinx FFT Logicore," http://www.xilinx.com/products/ipcenter/FFT.htm.

[28] H. Shousheng and M. Torkelson, "Designing pipeline FFT processor for OFDM (de)modulation," vol. 29, Oct. 1998, pp. 257–262.

[29] B. G. Jo and M. H. Sunwoo, "New continuous-flow mixed radix CFMR FFT using novel in-place strategy," *IEEE Trans. Circuits and Systems I: Regular Papers,*, vol. 52, no. 5, pp. 911–919, May 2005.

[30] B. M. Baas, "A generalized Cached-FFT algorithm," in *Proc. IEEE Int. Conference on Acoustics, Speech, and Signal Processing*, vol. V, Mar. 2005, pp. 89–92.

[31] H.-Y. L. Y.-W. Lin and C.-Y. Lee, "A dynamic scaling FFT processor for DVB-T applications," *IEEE J. Solid-State Circuits*, vol. 39, pp. 2005–2013, Nov. 2004.

[32] H. Kee, N. Petersen, J. Kornerup, and S. S. Bhattacharyya, "Systematic generation of fpga-based fft implementations," in *Proc. IEEE Int. Conference on Acoustics, Speech, and Signal Processing*, March 2008, pp. 1413–1416.

[33] P. A. Milder, F. Franchetti, J. C. Hoe, and M. Püschel, "FFT compiler: From math to efficient hardware," in *IEEE Int. High Level Design Validation and Test Workshop*, nov. 2007, pp. 137 –139.

[34] I. Uzun, A. Amira, and A. Bouridane, "FPGA implementations of fast Fourier transforms for real-time signal and image processing," *IEE Proceedings of Vision, Image and Signal Processing*, vol. 152, no. 3, pp. 283–296, 3 June 2005.

[35] N. Miyamoto, L. Karnan, K. Maruo, K. Kotani, and T. Ohmi, "A small-area high-performance 512-point 2-dimensional FFT single-chip processor," Sep. 2003, pp. 603–606.

[36] H. Kee, S. S. Bhattacharyya, N. Petersen, and J. Kornerup, "Resource-efficient acceleration of 2-dimensional Fast Fourier Transform computations on FPGAs," in *Third ACM/IEEE Int. Conference on Distributed Smart Cameras*, Aug. 2009, pp. 1–8.

[37] "VID471: Full camera link frame grabber with video processing resources," Website, http: //www.4dsp.com/VID471.htm.

[38] P. A. Milder and et al., "Discrete Fourier transform compiler: from mathematical representation to efficient hardware," Carnegie Mellon University, Tech. Rep. CSSI-07-01, 2007.

[39] C. Van Loan, *Computational framework of the fast Fourier transform*. SIAM, 1992.

[40] N. P. Pitsianis, "The Kronecker product in approximation and fast transform generation," Dissertation for the degree of Doctor of Philosophy, Cornell University, Jan. 1997.

[41] H. R. Wu and F. J. Paoloni, "The structure of vector radix Fast Fourier Transforms," *IEEE Trans. Acoustics, Speech, and Signal Processing*, vol. 37, no. 9, September 1989.

[42] "Hard tri-mode MAC," Website, http://www.xilinx.com/products/design_resources/conn_central/protocols/gigabit_ethernet.htm.

[43] "lwIP," Website, http://www.sics.se/~adam/lwip/.

[44] D. Elam and C. Lovescu, "A block floating point implementation for an N-point FFT on the TMS320C55X DSP," *Application Report SPRA948, Texas Instruments, Dallas, Texas, USA,*, 2003.

[45] P. Welch, "A fixed-point fast Fourier transform error analysis," *IEEE Trans. Audio and Electroacoustics*, vol. 17, no. 2, pp. 151–157, Jun 1969.

[46] D. Takahashi, "Efficient Implementation of Parallel Three-dimensional FFT on Clusters of PCs," *Computer Physics Communications*, vol. 152, pp. 144–150, 2003.

[47] "Processor Local Bus," http://www.xilinx.com/support/documentation/ip_documentation/plb_v46.pdf.

[48] "Xilinx Multi-port Memory Controller," http://www.xilinx.com/support/documentation/ip_documentation/mpmc.pdf.

[49] "The BEE3 Hardware Platform," http://www.beecube.com/platform.html.

[50] "Virtex-6 FPGA ML605 Evaluation Kit," http://www.xilinx.com/products/devkits/EK-V6-ML605-G.htm.

[51] "The USC-SIPI Image Database," http://sipi.usc.edu/database/.

[52] "3T MRI," http://www.healthsystem.virginia.edu/internet/physicians-direct/images/julystories/01MN_Tim_App_MR_002.jpg.

[53] "Dresden - Historical City Center," http://www.dlr.de/hr/en/Portaldata/32/Resources/images/eusar/EUSAR2006-home-E-SAR-image.jpg.

[54] J. W. Goodman, *Introduction to Fourier Optics*. New York: McGraw-Hill, 1968.

[55] R. O. Harger, *Synthetic Aperture Radar Systems: Theory and Design*. New York: Academic Press, 1970.

[56] J. R. Bennett and I. G. Cumming, "A digital processor for the production of SEASAT synthetic aperture radar imagery," in *Proc. SURGE Workshop, Frascati*, Jul. 1979, pp. 16–18.

[57] R. Raney, H. Runge, R. Bamler, I. Cumming, and F. Wong, "Precision SAR processing using chirp scaling," *IEEE Trans. Geoscience and Remote Sensing*, vol. 32, no. 4, pp. 786–799, jul. 1994.

[58] G. Franceschetti and G. Schirinzi, "A SAR processor based on two-dimensional FFT codes," *IEEE Trans. Aerospace and Electronic Systems*, vol. 26, no. 2, p. 356 366, Mar. 1990.

[59] H. Jeong, J. Park, H. Ryu, J. Kwon, and Y. Oh, "VLSI architecture for SAR data compression," *IEEE Trans. Aerospace and Electronic Systems*, vol. 38, no. 2, pp. 427–440, Apr. 2002.

[60] H. Izumi, K. Sasaki, K. Nakajima, and H. Sato, "An efficient technique for corner-turn in SAR image reconstruction by improving cache access," in *Proc. Int. Parallel and Distributed Processing Symposium*, 2002, pp. 3–8.

[61] Y. Pi, H. Long, and Huang, "A SAR parallel processing algorithm and its implementation," in *Proc. Future Intelligent Earth Observing Satellites Conference*, vol. 1, 2004, pp. 211–214.

[62] M.-M. Bian, L.-N. Gao, Y.-Z. Xie, and X.-B. Tan, "High-performance system design of SAR real-time signal processing," in *Proc. Int. Conference on Computer Application and System Modeling*, vol. 12, 2010, pp. 126–129.

[63] B. Tu, D. Li, and C. Han, "Two-dimensional image processing without transpose," in *Proc. Int. Conference on Signal Processing*, vol. 1, Aug. 2004, pp. 523–526.

[64] Y. Dou, J. Zhou, Y. Lei, and X. Zhou, "FPGA SAR processor with window memory accesses," in *Proc. IEEE Int. Conference on Application-specific Systems, Architectures and Processors*, 2007, pp. 95 –100.

[65] J. Zhou, Y. Dou, Y. Lei, and Y. Dong, "Window memory accesses method in alternate row/column matrix access systems," in *Proc. Int. Conference on Computer Engineering and Technology*, vol. 3, 2010, pp. 201–205.

[66] B. Liu, K. Wang, X. Liu, and W. Yu, "An efficient signal processor of synthetic aperture radar based on GPU," in *European Conference on Synthetic Aperture Radar*, June 2010, pp. 1054–1057.

[67] X. Ning, C. Yeh, B. Zhou, W. Gao, and J. Yang, "Multiple-GPU accelerated range-Doppler algorithm for synthetic aperture radar imaging," in *IEEE Radar Conference*, May 2011, pp. 698–701.

[68] C.-L. Yu, K. Irick, C. Chakrabarti, and V. Narayanan, "Multidimensional DFT IP generator for FPGA platforms," *IEEE Trans. Circuits and Systems I: Regular Papers*, vol. 58, no. 4, pp. 755 –764, April 2011.

[69] M. di Bisceglie, M. Di Santo, C. Galdi, R. Lanari, and N. Ranaldo, "Synthetic aperture radar processing with GPGPU," *IEEE Signal Processing Magazine*, vol. 27, no. 2, pp. 69 –78, March 2010.

[70] A. Moreira, J. Mittermayer, and R. Scheiber, "Extended chirp scaling algorithm for air- and spaceborne SAR data processing in stripmap and scansar imaging modes," *IEEE Trans. Geoscience and Remote Sensing*, vol. 34, no. 5, pp. 1123 –1136, Sep 1996.

[71] C. Cafforio, C. Prati, and F. Rocca, "SAR data focusing using seismic migration techniques," *IEEE Trans. Aerospace and Electronic Systems*, vol. 27, no. 2, pp. 194 –207, Mar 1991.