Major Project

On

Design of FPGA based Co-Processor for Stereo Imaging Algorithms using Handel-C

By

Chavda Jaiminkumar Buddhisagar

(06MCE003)

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY

AHMEDABAD 382 481

MAY 2008

Major Project

On

Design of FPGA based Co-Processor for Stereo Imaging Algorithms using Handel-C

Submitted in partial fulfillment of the requirements

For the degree of

Master of Technology in Computer Science & Engineering

By

Chavda Jaiminkumar Buddhisagar
(06MCE003)

Under Guidance of

Dr. S. N. Pradhan

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING

INSTITUTE OF TECHNOLOGY

NIRMA UNIVERSITY OF SCIENCE & TECHNOLOGY

AHMEDABAD 382 481

MAY 2008

This is to certify that Dissertation entitled

Design of FPGA based Co-Processor For Stereo Imaging Algorithms Using Handel-C

Submitted by

Chavda Jaiminkumar B.

has been accepted toward fulfillment of the requirement

for the degree of

Master of Technology in Computer Science & Engineering

Dr. S. N. Pradhan
P.G. Coordinator

Prof. D. J. Patel
Head of The Department

Prof. A. B. Patel
Director, Institute of Technology

CERTIFICATE

This is to certify that the work presented here by Mr. Jaiminkumar B. Chavda entitled "**Design of FPGA based Co-Processor For Stereo Imaging Algorithms Using Handel-C** " has been carried out at Institute Of Technology, Nirma University during the period September 2007 – May 2008 is the bonafide record of the research carried out by him under my guidance and supervision and is up to the standard in respect of the content and presentation for being referred to the examiner. I further certify that the work done by him is his original work and has not been submitted for award of any other diploma or degree.

Dr. S. N. Pradhan
Guide and P.G. Coordinator,
Department of Computer Science & Engineering,
Institute of Technology,
Nirma University,
Ahmedabad.

Date:

ACKNOWLEDGEMENTS

The successful completion of a project is generally not an individual effort. It is an outcome of the cumulative efforts of a number of persons, each having own importance to the objective. This session is a vote of thanks and gratitude towards all those persons who have directly or indirectly contributed in their own specials way towards the completion of this project.

It gives me great pleasure in expressing thanks and profound gratitude to **Dr. S. N. Pradhan**, Guide and P.G. Coordinator, Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for his valuable guidance and continual encouragement throughout the project. I heartily thankful him for his time to time suggestions and the clarity of the concepts of the topic that helped me a lot during the project. I also express thanks to **Mrs. Swati Jain**, Project Co-Coordinator and Asst. Professor, Department of Computer Engineering, Institute of Technology, Nirma University, Ahmedabad for her support, guidance and continuous encouragement throughout the dissertation work.

I would like to give my special thanks to **Prof. D. J. Patel**, Head, Department of Computer Engineering, Institute of Technology, Nirma University for his continual kind words of encouragement and motivation throughout the Project. I am thankful to all faculty members for their special attention and suggestion towards the project work.

I extend my sincere thanks to my colleagues specially Pranav S. Tank and Gaurang Thakkar for their support in my dissertation work. I would like to express my gratitude towards my family members who have always been my source of inspiration and motivation.

Jaiminkumar Chavda
(06MCE003)

Abstract

Computer manipulation of images is generally defined as Digital Image Processing (DIP). DIP is employed in variety of applications, including video surveillance, target recognition, and image enhancement. These are usually implemented in software but may also be implemented in special purpose hardware to meet timing constrains.

FPGAs are often used as implementation platforms for real-time image processing applications because their structure is able to exploit spatial and temporal parallelism. Such parallelization is subject to the processing mode and hardware constraints of the system. These constraints can force the designer to reformulate the algorithm. This thesis represents some general techniques for dealing with the various constraints and efficient mappings for various image processing algorithms on parallel hardware.

In this work the Fourier transform operation on image has been proposed using hybrid architecture of DSP and FPGA to achieve high throughput and to reduce processing time for the same. Hybrid architecture has been developed using reconfigurable architecture and hardware is modeled using a C-like hardware language called Handle-C. The proposed architecture is capable of producing one complex multiplication (Fixed point and Floating point) on every clock cycle. The hardware modeled was implemented using the DK4 Design Suite on the Xilinx Spartan 1500L FPGA. Another part of work focuses on development of Stereo Image Matching Algorithm named as Hierarchical Image Matching on the Spartan Xilinx Spartan 4000L FPGA. This includes implementation of Image Correlation block and Square root block using CORDIC algorithm. The algorithm was tested on standard image processing benchmarks and significances of the result are discussed.

Table of Contents

LIST OF FIGURES

LIST OF TABLES

LIST OF ABBREVIATIONS

AHDL	Altera Hardware Description Language
ANSI	American National Standards Institute
API	Application Programming Interface
ASIC	Application Specific Integrated Circuits
ASSP	Application Specific Signal Processors
BRAM	Block RAM
CCS	Code Composer Studio
CLB	Configurable Logic Blocks
CMOS	Complementary Metal-Oxide Semiconductor
COM	Component Object Model
CORDIC	Coordinate Rotational Digital Computer
CPLD	Complex Programmable Logic Devices
DEM	Digital Elevation Model
DFT	Discrete Fourier Transform
DIF	Decimation in Frequency
DIP	Digital Image Processing
DIT	Decimation in Time
DRAM	Dynamic Random Access Memory
DSP	Digital Signal Processing
EDA	Electronic Design Automation
EDIF	Electronic Design Interchange Format
EMIF	Extended Memory Interface
FFT	Fast Fourier Transform
FIFO	First In First Out
FPGA	Field Programmable Gate Array
FPIC	Field Programmable Inter-Connect
HDL	Hardware Description Language
HIM	Hierarchical Image Matching
IEEE	Institute of Electrical and Electronics Engineers
IO	Input-Output
IP	Intellectual Property
IOB	Input-Output Blocks
ISO	International Organization for Standardization
LUT	Look Up Table
MGT	Multi Gigabit Transceivers
PC	Personal Computer
PDK	Platform Development Kit
PLD	Programmable Logic Device
PCI	Peripheral Component Interconnect
RAM	Random Access Memory
ROM	Read Only Memory
RTL	Register Transfer Level
SDRAM	Synchronous Dynamic Random Access Memory
SIMD	Single Instruction Multiple Data
SRAM	Static Random Access Memory
SRL	Shift Register LUT

SPLD	Simple Programmable Logic Devices
S-Video	Separate Video, Super Video
VB	Visual Basic
VC++	Visual C++
VLSI	Very Large Scale Integrated
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
WOM	Write Only Memory
YUV	1-Luminance + 2 Chrominance Components
YC_bC_r	Luminance, Blue Chrominance, Red Chrominance
µP	Microprocessor
µC	Microcontroller

1. INTRODUCTION

1.1. General

Image processing is, in general, characterized by area of very high computational demands. Although it can be handled by “standard” computers, such solution is not viable for an embedded system, where dimensions of the computer system, power consumption or data throughput are of concern. For these reasons, specialized hardware solutions based on a digital signal processor (DSP) or a Field Programmable Gate Array (FPGA) are usually used in embedded systems. As increasingly complex algorithms and applications are being developed, the performance demands of these algorithms increasing exponentially. For cost-sensitive, high-volume applications like stereo image processing and PC graphics cards, this has driven the development of extremely specialized Application Specific Signal Processors (ASSPs). However, for many other applications, the only options for implementing high-performance digital signal processing have been general-purpose Digital Signal Processors (DSPs) and, more recently, FPGAs.

DSPs have typically been used to implement many of these applications. Although DSPs are programmable through software, the DSPs’ hardware architecture is not flexible. Therefore, DSPs are limited by fixed hardware architecture such as bus performance bottlenecks, a fixed number of Multiply Accumulate (MAC) blocks, fixed memory, fixed hardware accelerator blocks and fixed data widths. The DSPs’ fixed hardware architecture is not suitable for many applications that require customized DSP function implementations. Embedded DSP processors vary in their customization for the specific problem at hand. These processor types will range from general purpose processors that handle a wide variety of applications, to application-specific processors like DSPs, which are specific to a particular application class such as signal processing, to single purpose processors, which are customized to a very specific function. A single purpose processor is a digital circuit designed and implemented to execute a very precise program. In a digital camera, for

example, a single purpose processor is often used to implement a JPEG codec, which can then be used to perform compression and decompression on video frames.

The heart of any digital signal processing architecture is the Multiply-and-Accumulate (MAC) unit. Most signal processing applications utilize a great deal of multiplication: The MAC unit of a DSP accelerates this type of calculation by performing the multiplication of two numbers and then adding the result to all of the previous multiplications in what is called an "accumulator". Another key enabling technology of DSPs is the ability to process several operations at the same time. Known as parallel or concurrent processing, the concept is that if you can process several operations simultaneously, you can finish a task that much faster. One way that DSPs can execute four operations at the same time is to use what is known as a Very Long Instruction Word or VLIW architecture. A VLIW is a single instruction that actually represent several operations. In the case of the C64x DSP architecture, the VLIW has eight fields, four of which tell the four MAC units what to do next.

The architecture of FPGA, on the other hand, is designed with fine-grain parallelism, which makes it well suited for massively parallel algorithms. The basic characteristics of FPGA are relatively small capacity of the on-chip memory and relatively narrow throughput of memory interfaces, lack of wide-word processing units, and high cost of performing complex numerical operations, such as division, square root, logarithmic, exponential, and goniometrical functions (in smaller devices, these operations cannot be implemented at all). FPGAs provide a reconfigurable solution for implementing traditional DSP applications and offer higher DSP throughput and raw data processing power than DSPs. Since FPGAs are reconfigured in hardware, FPGAs offer complete hardware customization while implementing various DSP applications. Therefore, DSP systems implemented in FPGAs can have customized architecture, customized bus structure, and customized memory, customized hardware accelerator blocks and a variable number of MAC blocks.

A major advantage of FPGAs for many system architectures is the availability of package vertical migration which enables a single board design to support flexible processing performance and cost without respinning the board. System architects use this capability to create products with various price points and performance capabilities without significantly affecting development costs or inventory.

In Image processing, most of the operations on an image are simple and very repetitive – best implemented in an FPGA. However, an imaging pipeline is often used to identify "blobs" or "Regions of Interest" in an object being inspected. These blobs can be of varying sizes, and subsequent processing tends to be more complex. The algorithms used are often adaptive, depending on what the blob turns out to be… so a DSP-based approach may be better for the back end of the imaging pipeline.

FPGA devices provide a reconfigurable DSP solution for various DSP applications. FPGA devices incorporate a variety of embedded features such as embedded processors, DSP blocks, and memory blocks. These device features provide very high DSP capability in FPGAs compared to DSP processors. Using FPGAs, DSP designers can customize their hardware for optimal implementation of their applications. Using embedded processors such as the Nios embedded processor; FPGAs also offer a software-based design flow similar to the traditional DSP software design flow. Using this design flow, a DSP designer can implement a complete DSP system in an FPGA and thereby develop a cost-effective, high-performance DSP system.

1.2. Motivation

Image processing is a one of the fast developing research area of computer vision. Faster image processing is very essential in current scenario for work automation .This work is useful in developing the vision using the computerized analysis , object detection and classification of the images captured by the sensors for better interpretation and analysis. When it comes to processing the digital images of high resolution in order of 4Kx4K, 8Kx8K and higher then this ,normal workstations can able to

deliver the output but by taking the more computational resources in terms of memory and time to generate the output (or with decreased performance). Space agencies use the special purpose satellites for the earth surveillance which results the high resolution satellite images. Processing of these satellite images and video takes more time in processing .Sometimes it is possible that the real time faster processing is required and that work is generally not supported by the workstations. To resolve this we require special purpose processors like DSPs or ASPs specifically designed for these types of applications.

As basic image processing algorithms focus on the image enhancement, correction, object detection and feature detection like tasks. While the specific applications like stereo imaging, medical imaging like applications needs special treatment than that of the basic tasks. Developing of library on the customized DSP and FPGA would require more attention than just implementation. FPGAs are use in such applications where the requirement of simple mathematical processing can be done very fast and in parallel. This is possible using multiple parallel processing units available for processing if higher data-rate I/Os are available. So FPGAs can work as ASPs as well as DSPs for high throughput achievement. Stereo imaging library requires large amount of computation power which is not available at the workstation end and also these workstations are not efficient and capable handling the resource requirement for these applications. These applications require special attention for real time execution in timely manner so that the constant high throughput can be achieved.

This thesis aims at the proposing the solution for faster throughput by the means of using the DSP and FPGA's hybrid architecture. To achieve the mentioned goal would require the development of Library of Image processing functions for DSP and FPGA designs developed as special purpose processors.

1.3. Scope of Work

As the title suggests the goal of this dissertation, work carried out in this

research is useful for the organizations which require the high computation power for processing the high resolution images and videos.

The work of dissertation is analysis, design and implementation of the Basic Image processing algorithms on the DSP and FPGA architecture. These includes the testing of the various image processing algorithms on the DSPs TI c64xx (Texas Instruments), TS-201(Analog Devices), and Pentium-4(on SimpleScalar 3.0 Simulator Toolkit) and comparing the performance of each Processor .This includes selection of right DSP processors and development of algorithms by partitioning the parallel computation tasks between the FPGA (Spartan-3) and DSP Processor for increased throughput.

In this thesis, development of the Hierarchical Image matching algorithm is done on Handel-C language and verification of results of FPGA is done on the DK Design Suite Handel-C simulator. Another algorithm FFT implemented on DSP-FPGA by implementation of Fixed/Floating Point complex Multiplication on FPGA. This work is done based on task division of various operations between DSP and FPGA for checking the throughputs for each division.

1.4. Outline of Thesis

This thesis is organized as follows:

- **Chapter 2** provides basics of FPGA architecture, Building blocks of FPGA, Working of FPGAs, EDA tools available for FPGAS programming and basics stereo imaging algorithms.

- **Chapter 3** provides information about DFT and FFT algorithms .This includes butterfly algorithms for DIT and DIF implementation of FFT. This chapter also discusses the fixed point and floating point data types on FPGAs and implementations of arithmetic functions of both on FPGAs

- **Chapter 4** provides in brief details of Hierarchical stereo image matching algorithm, image pyramid generation for

stereo image pair, implementation of matching algorithm by preprocessing ,the matching by correlation function by windowing method, and finding conjugate point pair from image pair. This also includes development of Square root function and correlation function in FPGA using Handel-C and CORDIC algorithm.

- **Chapter 5** provides detailed results of FPGA performance data and analysis of the results obtained.

- **Chapter 6** concludes this thesis with a summary, and provides possible directions for relevant future research.

2. LITERATURE SURVEY

2.1. INTRODUCTION

Programmable logic is loosely defined as a device with configurable logic and Flip-flops linked together with programmable interconnect. Memory cells control and define the function that the logic performs and how the various logic functions are interconnected. Though various devices use different architectures, all are based on this fundamental idea.

There are few major programmable logic architecture available today. Each of the architecture typically has vendor-specific sub-variants within each type [10]. The major types include:

- Simple Programmable Logic Devices (SPLDs),
- Complex Programmable Logic Devices (CPLDs), and
- Field Programmable Gate Arrays (FPGAs)
- Field Programmable Inter-Connect (FPICs)

2.2. FPGA - Field Programmable Gate Array

An FPGA consists of a matrix of logic blocks that are connected by a switching network. Both the logic blocks and the switching network are reprogrammable allowing application specific hardware to be constructed, while at the same time maintaining the ability to change the functionality of the system with ease. As such, an FPGA offers a compromise between the flexibility of general purpose processors and the hardware-based speed of ASICs. Performance gains are obtained by bypassing the fetch-decode-execute overhead of general purpose processors and by exploiting the inherent parallelism of digital hardware.

FPGA is a silicon chip with unconnected logic gates. It is an integrated circuit that contains many (64 to over 10,000) identical logic cells that can be viewed as standard components. The individual cells are interconnected by a matrix of wires and programmable switches. Field Programmable means that the FPGA's function is defined by a user's program rather than by the manufacturer of the device. Depending on the

particular device, the program is either 'burned' in permanently or semi-permanently as part of a board assembly process, or is loaded from an external memory each time the device is powered up [17].

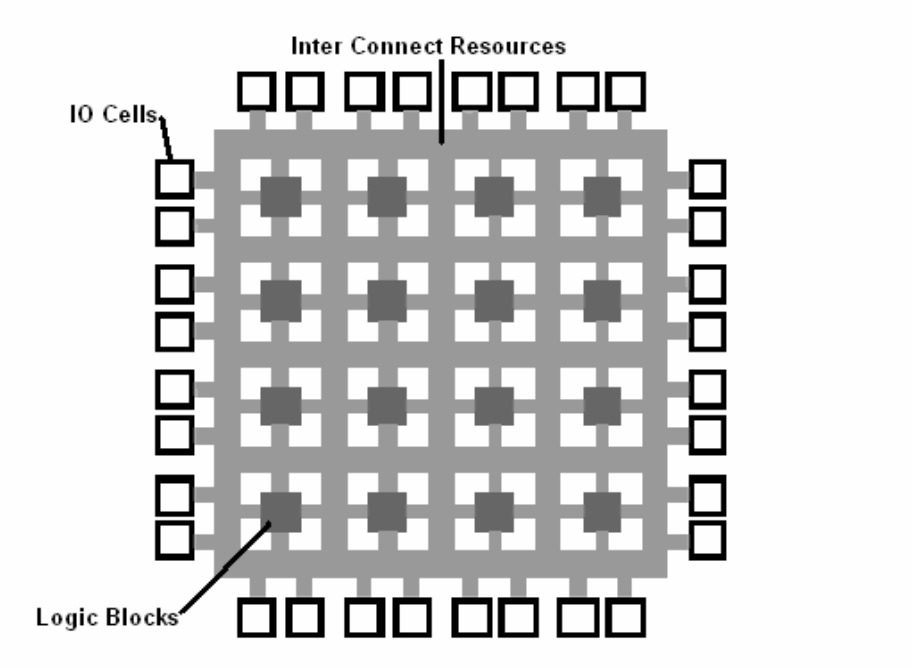


Figure 2.1: Basic FPGA Architecture [17]

The FPGA has three major configurable elements:

1. configurable logic blocks(CLBs),
2. input/output blocks,
3. Interconnects.

The CLBs provide the functional elements for constructing user's logic. The IOBs provide the interface between the package pins and internal signal lines. The programmable interconnect resources provide routing paths to connect the inputs and outputs of the CLBs and IOBs onto the appropriate networks. Using these three basic components FPGA can implement any functions. These basic components are building blocks for FPGA which provides flexibility of implementation.

As shown in Figure 2.2, each CLB contains a logic element which is implemented as a lookup table. This logic element operates on four one-bit inputs and outputs single data bit. Using CLB any Boolean function of

four inputs can be performed. These include 64K functions available functions using FPGA.

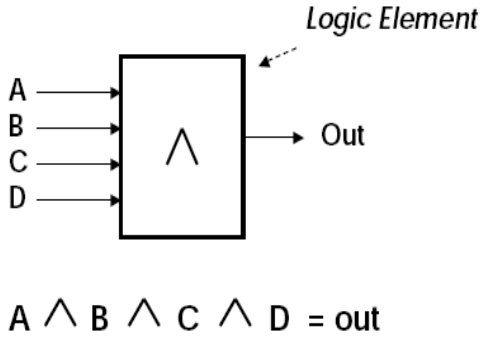


Figure 2.2: Basic CLB Architecture [17]

The Field-Programmable Gate Arrays (FPGAs) provide the benefits of custom Complementary Metal-Oxide Semiconductor (CMOS) Very Large Scale Integrated (VLSI), while avoiding the initial cost, time delay, and inherent risk of a conventional masked gate array. The FPGAs are customized by loading configuration data into the internal memory cells. FPGAs are becoming a critical part of every system design. There are many different FPGAs with different architectures but all of them have the same common feature: that the layout of unit is repeated in matrix form. In this case, the unit is consisting of PLDs, logic gates, RAM, and many other specific components. There are four main classes of FPGAs currently commercially available: symmetrical array, row-based, hierarchical PLD, and collection-of-gates.

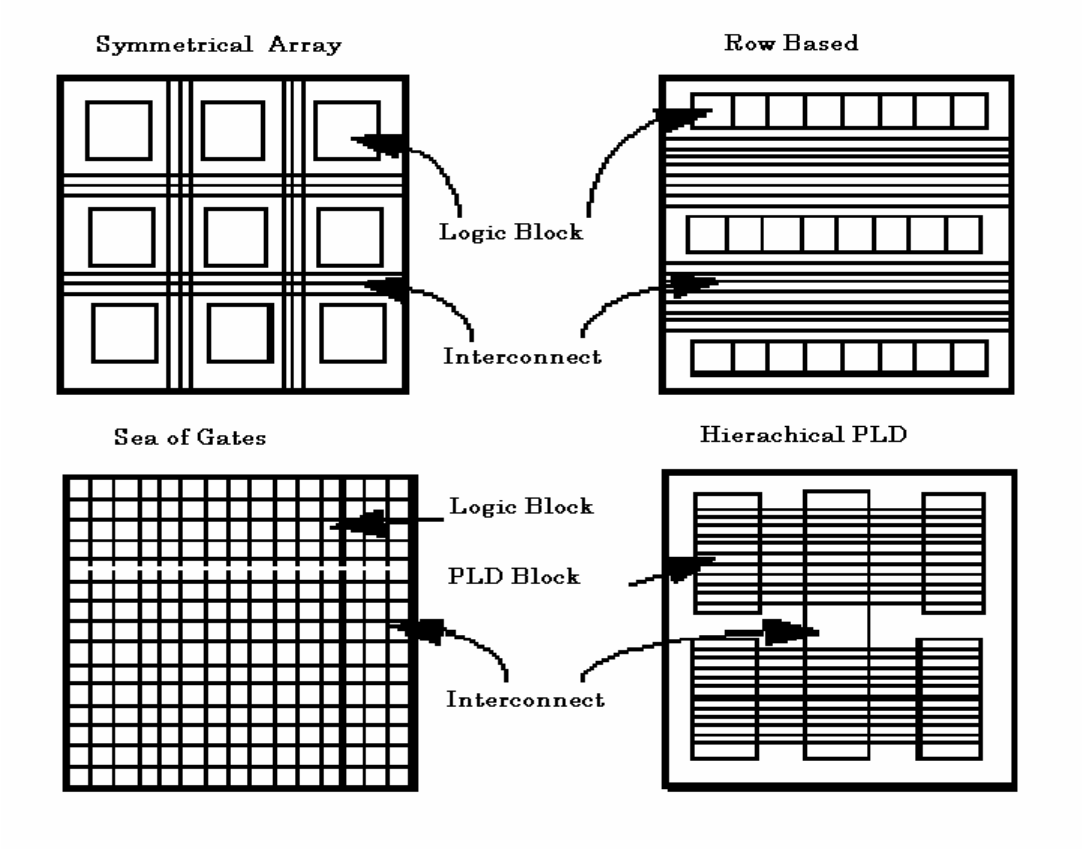


Figure 2.3: Classes of FPGA Architecture [17]

An FPGA has a large number of these cells available to use as building blocks in complex digital circuits. Custom hardware has never been so easy to develop. Like microprocessors, RAM based FPGAs can be infinitely reprogrammed in-circuit in only a fraction of a second. Design revisions, even for a fielded product, can be implemented quickly and painlessly. Taking advantage of reconfiguration can also reduce hardware. Logic networks realized in FPGA are slower than those realized in full custom design, but are much faster by several orders than simulation of logic functions by software. Even application programmers can be run on FPGAs and performed much faster than on general purpose computers in many cases. With FPGAs, debugging or prototyping of new design can be done as easily and quickly as software. Availability of reprogrammable technologies has enabled the configuration of flexible system allowing runtime configuration of system hardware and software. The design methodology combines a C-based software design targeting FPGAs as a

device and rapid FPGA hardware design flow based on Handel-C, a C-like programmable language [16].

2.3. FPGA Design Options

In order to create an FPGA design, a designer has several options for algorithm implementation. While gate-level design can result in optimized designs, the learning curve is considered prohibitory for most engineers, and the knowledge is not portable across FPGA architectures. The following text discusses several high-level hardware design languages (HDLs) in which FPGA algorithms may be designed [19].

2.3.1. Verilog HDL

Originally intended as a simulation language, Verilog HDL represents a formerly proprietary hardware design language. Currently Verilog can be used for synthesis of hardware designs and is supported in a wide variety of software tools. It is similar to the other HDLs, but its adoption rate is decreasing in favor of the more open standard of VHDL. Still, many designers favor Verilog over VHDL for hardware design, and some design departments use only Verilog. Therefore, as a hardware designer, it is important to at least be aware of Verilog [33].

2.3.2. AHDL-a Hardware Design Language

Altera Hardware Design Language (AHDL) is proprietary, and is only supported in Altera-specific development tools. This may be seen as a drawback, but since AHDL is proprietary, its use can also result in more efficient hardware design, when code portability is not an issue. In typical design environments, different FPGA architectures are used for different designs, meaning that time spent learning AHDL may be wasted if a Xilinx FPGA is later chosen [30].

2.3.3. VHSIC Hardware Design Language

In recent years, VHSIC (Very High Speed Integrated Circuit) Hardware Design Language (VHDL) has become a sort of industry standard for high-level hardware design. Since it is an open IEEE

standard, it is supported by a large variety of design tools and is quite interchangeable (when used generically) between different vendors' tools. It also supports inclusion of technology-specific modules for most efficient synthesis to FPGAs.

2.3.4. Handel-C

Handel-C is a C-based language with true parallelism and priority-based channel communication, which can be compiled to hardware. Handel-C is a programming language developed by the Hardware Compilation Group at Oxford University Computing Laboratory, and now sold by Celoxica Ltd. It is ANSI-C based, with extensions based upon Concurrent Sequential Programs (CSP), such as parallelism and channel-based communication. Handel-C compiles directly to low-level hardware such as field-programmable grid arrays (FPGAs). To support such hardware, Handel-C features several extensions for dealing with data types of arbitrary widths [32].

2.3.5. Catapult-C

The Catapult C Synthesis tool from Mentor Graphics, targets designers developing application-specific integrated circuits (ASICs) or field-programmable gate arrays (FPGAs) for next-generation, compute-intensive applications such as wireless communication, satellite communication and video/image processing. By uniting system-level and hardware design, the Catapult C Synthesis tool combines with the Mentor Graphics® ModelSim® simulator to create the central foundation for a C-based design flow [31].

2.3.6. SystemC™

SystemC™ is a language built in standard C++ by extending the language with the use of class libraries. SystemC addresses the need for a system design and verification language that spans hardware and software. The language is particularly suited to model system's partitioning, to evaluate and verify the assignment of blocks to either hardware or software implementations, and to

architect and measure the interactions between and among functional blocks. Leading companies in the intellectual property (IP), electronic design automation (EDA), semiconductor, electronic systems, and embedded software industries currently use SystemC for architectural exploration, to deliver high-performance hardware blocks at various levels of abstraction and to develop virtual platforms for hardware/software co-design [28].

2.4. Celoxica's Handel-C

A custom edition of Celoxica's market leading DK Design Suite is included which enables designers to implement complex C algorithms directly into optimized FPGA hardware implementations. This capability removes the burden and time consuming effort of manually rewriting algorithms into Hardware Description Languages and enables multiple design iterations supported by very fast system simulation and verification [4].

Handel-C Language

Handel-C is a truly innovative language for implementing algorithms in hardware, architectural design space exploration, and hardware/software co-design. Based on ISO/ANSI-C, it has extensions required for hardware development. Therefore programs designed for Handel-C, are inherently sequential. It includes flexible data widths, parallel processing and communications between parallel elements. The language is designed around a simple timing model that makes it very accessible to system architects and software engineers.

Sequential Expressions	Parallel Expressions

`{`
`.....`
`a = 1;`
`b = 2;`
`.....`
`}`
	`par`
	`{`
	`... ...`
	`a = 1;`
	`b = 2;`
	`... ...`
	`}`

This executes the two statements, one after the other sequentially	This executes both statements in parallel

Table 2.1: Comparison between Sequential and Parallel execution

Handel-C provides special constructs, which enable expressions to be evaluated in parallel. It also provides the ability to specify the width of a data variable.Handel-C also enables the use of user defined variable sizes. E.g.:-

int n x;

This defines a variable *x* of type int and size of *n* bits.

When expressions are evaluated in parallel, communication between the parallel branches becomes a problem due to synchronization. Handel-C provides a design construct known as a channel to get around this. Channel provides a link between branches executing in parallel. One parallel branch outputs data onto channel and other branch reads data from the channel. Channels can be constructed with or without FIFO facility.

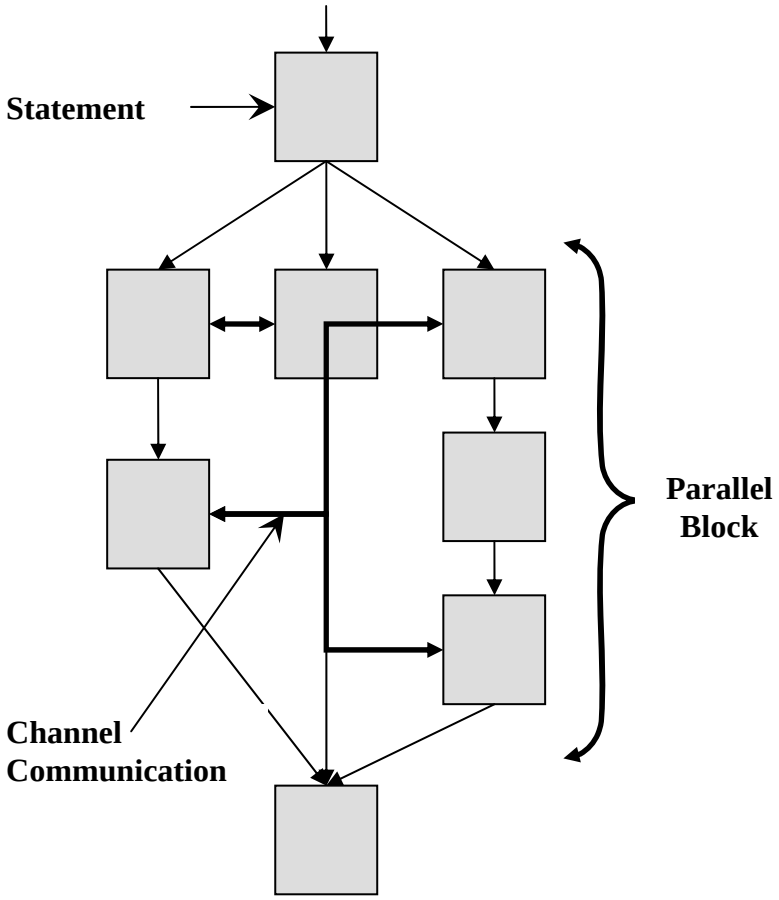


Figure 2.4 Parallel Programming Flow in Handel-C

Similarities with ANSI C

Handel-C has many similarities with C. At the same time Handel-C has many features which are not found in C and vice versa. Handel-C doesn't support a large variety of data types as C does, the only data types supported by Handel-C are Integers and Characters. But unlike C, the users can specify the width of Integers. This is possible as the implementation is directly in Hardware.

Handel-C	ANSI-C	Both
	double	
	float	
chan	enum	
ram	register	int
rom	static	unsigned
chain	extern	char
chanout	struct	long
undefined	volatile	short
interface	void	
	const	
	union	

Table 2.2: Data types of Handel-C and ANSI C

Design Procedure

Handel-C provides a simulator to test the program implementation before implementing it in Hardware. The simulator can step through each cycle of execution, and display the values of the variables after each cycle.

Once the designer is satisfied with his/her design, it can be compiled in to hardware. When compiling in to hardware the designer can target a specific hardware platform. Handel-C compiler currently produces net lists for Xilinx and Altera devices. Design procedure is given in Figure 2.5.

Comparison between Handel-C and VHDL

Prototyping new concepts or building first generation electronic devices is time consuming and costly, and in some cases high risk. Most algorithms

are prototyped in C and then translated into VHDL or Verilog—a process that introduces risks and errors.

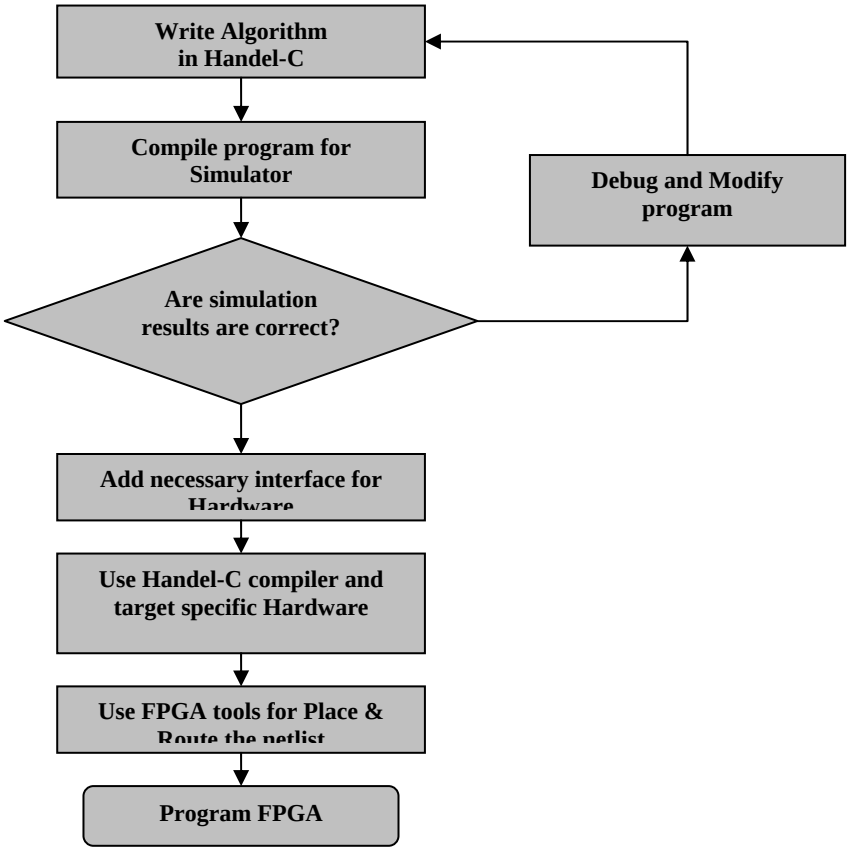


Figure 2.5: FPGA Design Flow using Handel-C

Handel-C avoids this problem because it is a language based on C and designed to describe algorithms, which are subsequently compiled down to hardware. Changes to the Handel-C code produce predictable changes in the resulting hardware. By targeting FPGAs directly, Handel-C provides a fast route for hardware prototyping and development of first generation electronic products. Functions can be compiled into libraries and used in other projects, with a simple declaration providing the interface to other code. Cores written in Handel-C can be exported as EDIF or VHDL “black boxes” for design reuse.

HDLs have evolved from an exclusively parallel world for describing the hardware rather than describing the desired function. What is needed is a language the raises the level of abstraction sufficiently to enable the designer to describe in the briefest possible way the desired function rather than its underlying structural detail. While Register Transfer Level (RTL) subsets of HDLs, such as VHDL and Verilog, do provide a functional

interpretation of the hardware description to enable the generation of hardware structure at compile time, their parallel nature requires the design engineer to add extra logic for sequential execution. By introducing a language that is similar to ANSI-C to the hardware design process, designers gain a language that is sequential by default with a high-level flow that is geared for programming functionality. But hardware is parallel and this needs to be accounted for if a software-influenced hardware design methodology based entirely on C is going to succeed at the RTL level [16].

Features	Benefits
High level language solution	Allows rapid development of multi-million gate FPGA designs and system-on-chip solutions
Based on ISO/ANSI-C	Allows application engineers to migrate concepts directly to hardware, for rapid prototyping and first generation electronics products
Well defined timing	Fast external I/O Simplifies pipeline
Explicit parallelism	'par' statement Simultaneous assessment
Supports complex C functionality including structures ,pointers and functions(shared and inline)	Shallow learning curve for software engineers, allows rapid implementations of very complex, modular systems
Includes extended operators for bit manipulation, and high level mathematical macros(including floating point)	Allows rapid translation of DSP Algorithms to efficient hardware
No state machines to design, control flow comes from C statements like if, case and while	Simplifies design of complex sequential control flows, intuitive to software engineers
Simple and consistent syntax extensions for specific hardware features like RAMs / ROMs, signals and external pin connections	Enables efficient use of available hardware without cumbersome syntax
Automatically deals with clocks, clock enables, and data transfers across clock domain boundaries	Abstracts away much of the complexity of hardware design

Table 2.3: Handel-C features and benefits

2.5. Celoxica DK4 Design Suite

The Celoxica DK4 design suite is a unique C direct-to-hardware solution that enables application specialists to migrate concepts directly to hardware without requiring the generation, simulation, or synthesis of hardware description languages (HDLs). The DK4 design suite focuses on the design, validation, iterative refinement and implementation of complex algorithms in hardware. It includes built-in design entry, simulation, and synthesis, driven directly by Handel-C, a programming language based on ISO/ANSI-C. The output of the compiler is either architecture optimized EDIF netlist appropriate for FPGAs, or RTL VHDL for existing tool suites.

The debugger of DK4 design suite provides in-depth features normally found only in software development. These include breakpoints, single stepping, variable watches, and the ability to follow parallel threads of execution. The hardware designer can step through the design just like a software design system using this approach. Co-simulation and verification facilities are built into the tool-chain, facilitating co-design with instruction set simulators, VHDL simulators such as ModelSim. A key benefit of this is that hardware/software partitioning decisions can be changed at any stage in the design process.

Synthesis in this design system correlates to software compilation in that it is very fast; software designers are accustomed to compiling changes to their designs very quickly and testing the results. This enables them to take many turns, make smaller changes and quickly recompile. The speed of Celoxica's design system brings this benefit to hardware design as well [16].

Predefined Hardware Libraries

Predefined libraries much like the standard libraries of ANSI-C and other software environments to hardware design create opportunities for simplifying the development of new functionality as well as encouraging design reuse. Handle-C has capabilities for accessing internal and external memory as well as registers. Via libraries of predefined functions, common APIs shield users from low level interfaces to ease the integration of

FPGAs to physical resources including both peripherals and processors – the latter enabling hardware/software co design. It is an approach that can mean significant time savings, giving the designer more time to concentrate on core functionality.

The Benefits of Using Handel-C

Handel-C supports a large set of ANSI C constructs, easy porting between the two languages is possible.

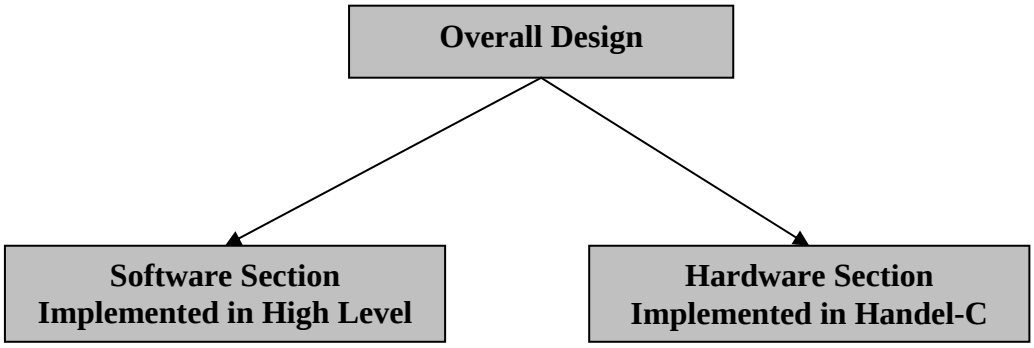


Figure 2.6: System Design Partitioning between Hardware & Software

In Custom Computing applications, part of the Design is implemented in Hardware while part of it is implemented in software .Since Handel-C is itself very similar to an imperative language, it makes the task of dividing the original design into a Hardware section and a Software section that much easier. The imperative nature of Handel-C also makes it easier to debug and upgrade the hardware design. This means that the Hardware component can be easily modified to take in to account modifications made to the Software component and vice versa.

Due to Handel-Cs high level nature it makes it possible for the same person to do both the Hardware and Software implementation. This greatly reduces the development cost as you do not need a two people to handle the Hardware and Software design separately.

2.6. Image Processing Algorithms

Parallelism in image processing algorithms exists in two major forms: spatial parallelism and temporal parallelism. FPGA implementations have the potential to be parallel using a mixture of these two forms. For

example, in order to exploit both forms the FPGA could be configured to partition the image and distribute the resulting sections to multiple pipelines all of which could process data concurrently. In practice, such parallelization is subject to the processing mode and hardware constraints of the system. This in turn forces the designer to deal with hardware issues such as concurrency, pipelining and priming, which many image processing experts are unfamiliar with [15].

Constraints

1. Timing Constraint

   Pipelining is a relatively easy optimization to perform, since it does not require that the algorithm be modified. Given enough resources, any desired throughput can be achieved by pipelining, at the expense of added latency. A number of higher-level languages already offer automatic pipelining capabilities.

2. Bandwidth Constraint

   FPGAs have very limited amounts of on-chip RAM (Xilinx calls this BlockRAM). The logic blocks themselves can be configured to act like RAM (termed distributed RAM) but this is usually an inefficient use of the logic blocks. Typically some sort of off-chip memory is used but this only allows a single access to the frame buffer per clock cycle, which can be a problem for the many operations that require simultaneous access to more than one pixel from the input image. This does not allow simultaneous access to elements /pixels in one clock cycle like Convolution or interpolation operation

3. Resource Constraint

   Resource contention arises due to the finite number of available resources in the system such as local and off-chip RAM or other function blocks implemented on the FPGA. If there are a number of concurrent processes that need access to a particular resource in a given clock cycle then some sort of scheduling must be performed. The worst case involves redesigning the underlying algorithm. Care must also be taken to ensure that concurrent processes avoid writing to the same register during a given clock cycle. Pipelining results in an increase in logic block usage. This is caused by the

need to construct pipeline stages and registers rather than being able to reuse the small number of sequential computing elements (ALU and registers), as can be done with offline processing. Flip-flops introduced by pipelining typically incur a minimum of additional area on an FPGA, as they are mapped onto unused flip flops within logic blocks that are already used for implementing other combinatorial logic in the design.

Image Processing Operations

1. Point Operations

   Point operations are a class of transformation operations where each output pixel's value depends only upon the value of the corresponding input pixel. The mapping of point operations to hardware can be achieved by simply passing the image though a hardware function block, that is designed to perform the required point operation. For more complex functions LUTs can be used.

2. Window based Operations

   A more complex class of low-level operations are local filters. Conceptually, each pixel in the output image is produced by sliding an $N \times M$ window over the input image and computing an operation according to the input pixels under the window and the chosen window operator. The result is a pixel value that is assigned to the centre of the window in the output image, as shown below in Figure 2.7 [15].

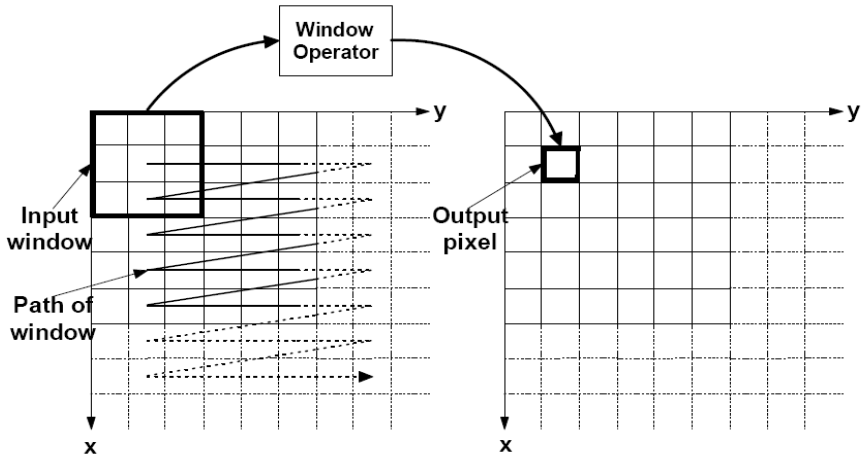


Figure 2.7: Conceptual example of window filtering [15]

For processing purposes, the straightforward approach is to store the entire input image into a frame buffer, accessing the neighborhood pixels and applying the function as needed to produce the output image. If real-time processing of the video stream is required N×M pixel values are needed to perform the calculations each time the window is moved and each pixel in the image is read up to N×M times. Memory bandwidth constraints make obtaining all these pixels each clock cycle impossible unless some form of local caching is performed. Input data from the previous N-1 rows can be cached using a shift register (or circular memory buffer) for when the window is scanned along subsequent lines. This leads to the block diagram shown below in Figure 2.8. Instead of sliding the window across the image, the above implementation now feeds the image through the window.

3. Global Processing Operations

   Intermediate level operations are often more difficult to implement on FPGAs as they convert pixel data to higher-level representations such as chain codes or regions of interest. These algorithms often require random access to memory that cannot easily be achieved in stream processing mode. The algorithm must be rewritten without the requirement of random access to memory using either single or multiple passes through the image. Chain coding is an example of an algorithm for which this must be performed. For example, finding chain code of contours in an image.

Using FPGAs for image processing applications presents system designers with some interesting problems. The analysis and manipulation of video images is inherently a high-bandwidth process, while software simulations do not always provide engineers with enough information to establish the performance of their algorithms because they are not in real time. For this reason, there is a growing demand for FPGA hardware prototyping systems targeted specifically at the requirements of image processing engineers – development platforms that provide the real-time I/O and

memory interface capabilities necessary to prototype today's most demanding signal processing applications.

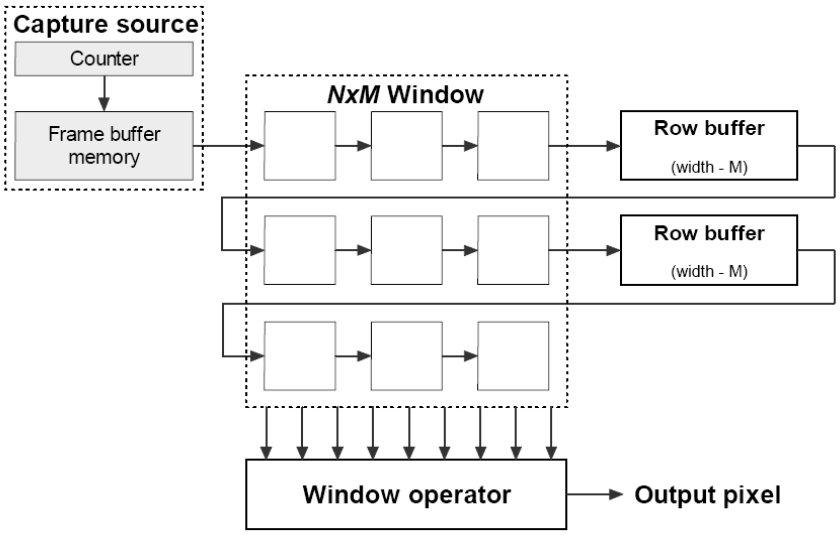


Figure 2.8: Block diagram for hardware implementation of window filtering [15]

2.7. FPGAs for Image Processing Applications

Required Elements

When considering the design of an FPGA platform for image processing algorithm development, it is useful to review the principal elements that are common to the most popular image analysis techniques, and what hardware resources are needed to support these elements [11] [12].

1. High-Speed I/O

Getting real-time data in and out of an image processor is obviously critical to the performance of the system. The images may alternatively be transferred asynchronously through a processor or backplane bus (PCI (Peripheral Component Interconnect) directly into frame stores.

The latest generation of serial bus interfaces, such as PCI Express, has sufficient bandwidth for multiple streams of uncompressed high-resolution images to be transferred in real-time. PCI Express interface and multiple MGTs (Multi Gigabit Transceivers), provide an excellent range of I/O capabilities for data transfers.

2. Frame Stores

The frame stores used for temporary storage of image data are at the core of many image processing functions. They serve many purposes, such as synchronizing multiple image sources, image re-sizing and

manipulation, motion or object detection and tracking, noise reduction, and de-interlacing.

The storage of multiple video frames typically requires a large quantity of memory that is accessed sequentially. For this reason, single- or double-data-rate SDRAM is normally employed. However, in applications where image transformations include perspective, rotation, or non-linear "warps" of the source image, the frame store architecture is typically based on very high speed synchronous static RAMs to give low-latency random access operation (rather than linear data bursts from SDRAM).

3. Filtering and Interpolation

Applications that involve re-sampling of the source image data also need to interpolate or filter the image to avoid aliasing problems. Efficient implementation of filters is therefore important. Spartan-3A DSP and Virtex-5 devices are useful here, as the architecture of their DSP blocks is ideally suited to the efficient implementation of all kinds of filters.

4. Delay Elements

Image processing algorithms typically require a wide range of delay elements. For delays of a few pixels – to equalize pipeline processing latency or for poly-phase filter taps – the SRL16 blocks in Xilinx FPGAs are highly efficient. Block RAMs, with their dual-port architecture, are ideal for line delays, which are needed where the vertical columns of data within 2D raster scan images are processed.

5. Look-Up Tables

Adjustments to the dynamic range of image data and non-linear transfer functions can be efficiently implemented using look-up tables. How such tables are implemented – and how they are referred to – depends on the number of inputs they have. Look-up tables (LUTs) are said to be "1D" where the table output value depends on only one input value, "2D" where the output depends on two inputs, and "3D" where it depends on three inputs. The 1D and 2D cases can usually be implemented in block RAM (depending on the size of the input vectors).

6. Software Control and Analysis

Virtually all image processing systems require some form of software-programmed control system, either to configure the system correctly or to analyze the data and present the results in an efficient manner. These control systems may range from a simple soft- or hard-IP core processor embedded in the FPGA (such as the PowerPC embedded in Virtex FPGAs). With such high performance processors, system designers now have the choice to split the image processing tasks between hardware and software; hence the importance of implementing a high-bandwidth data pipe between the two.

2.8. Stereo Image Processing

Stereo Imaging is the process of constructing the 3-Dimentional model using the 2-Dimentional Images for better human understanding. The task of building a general purpose computational-vision system is a grand challenge due to the compute-intensive nature of many vision algorithms. However, researchers have been successful in designing algorithms and building systems that deal with some specific tasks of the human vision system. One important feature of the human vision system is its ability to perceive depth of a viewed scene. This ability to perceive depth, known as stereo vision, or stereopsis is made possible by the difference in viewpoints of the scene when sensed by our left and right eyes. The information about depth in a scene is of great importance because it helps us navigate in a three-dimensional environment and aids us in recognizing objects of interest, among other tasks. In computer based stereo-vision systems, a stereo-rig is a pair of cameras placed side-by-side, much like our eyes, to capture the left and right images. The processing required extract depth information from the image pair may seem second nature when performed by the human brain due to its immense and complex computational capabilities. In a stereo-vision system, this processing is carried out using a computing platform that can be based on software, hardware, or a mixture of the two. The depth information is encoded in the disparity, defined as the difference in pixel locations of corresponding points in the image pair. The disparity is inversely proportional to the distance of an object from the cameras, so the disparity increases as

objects get closer to the cameras. The estimation of this disparity then becomes the primary task of a stereo-vision system [12].

In the simplest setup of a stereo-rig, where the optical axes of the two cameras are parallel and the vertical axes are aligned, corresponding pixels lie at the same vertical coordinate in the image pair. The search for the corresponding pixel is therefore limited to the same scanline in the image pair, which allows processing of each scanline as they arrive. In the more general case where the cameras are not aligned as described above, the search for corresponding pixel may span across numerous scanlines and this increases the computational load of the system. When the cameras are not in the ideal setup, Image rectification of input images can be performed. Rectification is the process by which the input image pair is warped to resemble the output from an aligned stereo-rig.

Often, when viewing a scene from different viewpoints as in a stereo setup, objects visible in one image may not be visible in the other image. A foreground object hides, or occludes, different parts of the background in the left and right views, a phenomenon known as occlusion. In addition, the information present at the left edge of the image captured by the left camera is not available in the right image and vice-versa as this part of the scene falls outside the viewing area of the other camera. This further complicates the task of accurate disparity estimation because pixels visible in one image may not have a corresponding match in the other image of the pair. Related areas of Stereo Imaging [19]:

* Aerial Stereo Photogrammetry
* Robotic Vision/Machine Vision
* 3D Computer Graphics
* Computer Vision Geometry

2.9. Epipolar Geometry

The key problem in stereo computation is to find corresponding points in the stereo images. Corresponding points are the projections of a single point in the three-dimensional scene. When camera attributes are known, corresponding image points can be mapped into three-dimensional scene

locations. In addition to providing the function that maps pairs of corresponding image points onto scene points, a camera model can be used to constrain the search for matching pairs of corresponding image points to one dimension (Figure 2.10). Any point in the three-dimensional world space, together with the centers of projection of two camera systems, defines a plane (called an "epipolar" plane). The intersection of an epipolar plane with an image plane is called an epipolar line. Every point on a given epipolar line in one image must correspond to a point on the corresponding epipolar line in the other image. The search for a match of a point in the first image may therefore be limited to a one-dimensional neighborhood in the second image plane, as opposed to a two-dimensional neighborhood, with an enormous reduction in computational complexity.

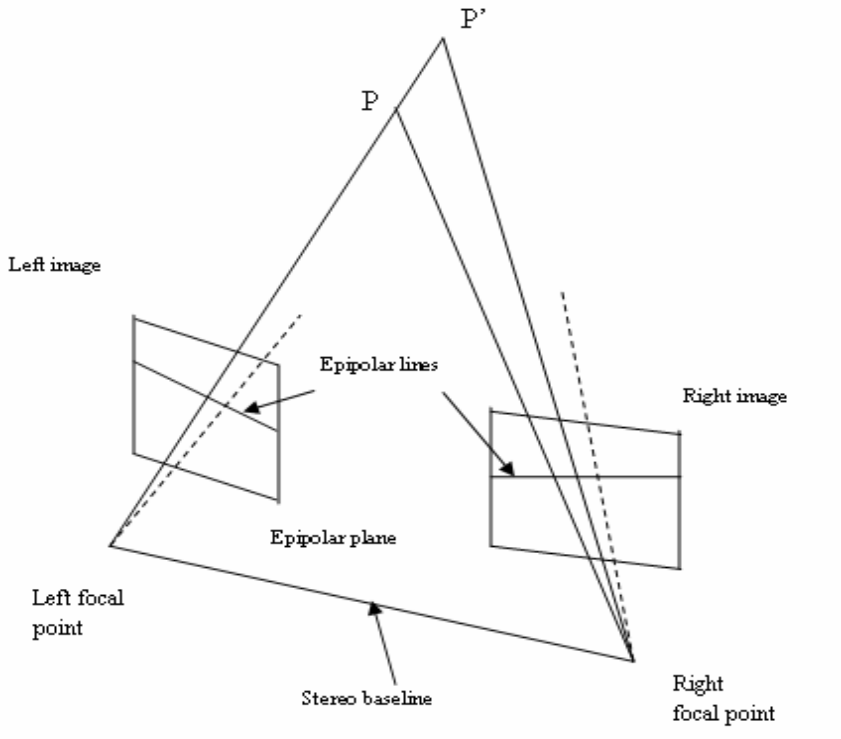


Figure 2.19: Epipolar Geometry [2]

When the stereo cameras are located and oriented such that there is only a horizontal displacement between them, then disparity can only occur in the horizontal direction, and the stereo images are said to be "in correspondence." When a stereo pair is in correspondence, the epipolar lines are coincident with the horizontal scan lines of the digitized pictures-

-enabling matching to be accomplished in a relatively simple and efficient manner. Stereo systems that have been primarily concerned with modeling human visual ability have employed this constraint. In practical applications, however, the stereo pair rarely is in correspondence.

The line connecting the focal points of the camera systems is called the *stereo baseline*. Any plane containing the stereo baseline is called an *epipolar plane*. The intersection of an epipolar plane with an image plane is called an *epipolar line*. If the geometrical relationship between the two camera systems is known, we need only search for a match along the epipolar line in the right image.

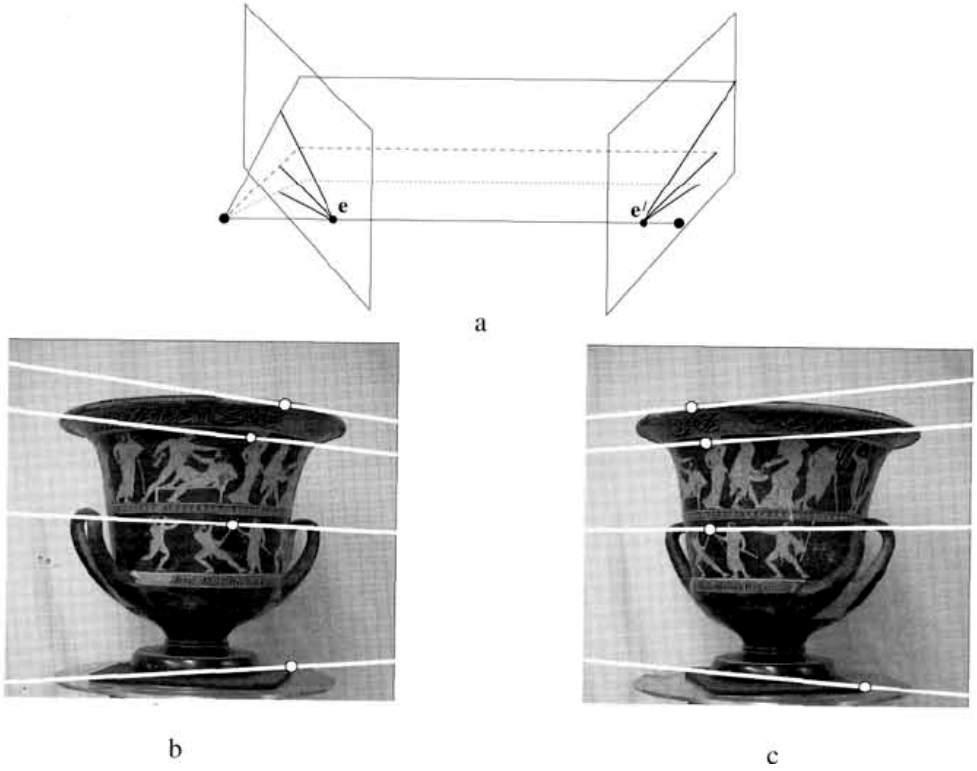


Figure 2.10: Stereo image Pair and Epipolar Geometry [18]

Various terminology of Epipolar Geometry

Epipole: The *epipole* is the point of intersection of the line joining the camera centers (the baseline) with the image plane. Equivalently, the epipole is the image in one view of the camera centre of the other view. It is also the vanishing point of the baseline (translation) direction.

Epipolar Plane: An *epipolar plane* is a plane containing the baseline. There is a one-parameter family (a pencil) of epipolar planes.

Epipolar Line: An *epipolar line* is the intersection of an epipolar plane

with the image plane. All epipolar lines intersect at the epipole. An epipolar plane intersects the left and right image planes in epipolar lines, and defines the correspondence between the lines.

Terminology shown above are explained well in Figure 2.10 (a) shows Epipolar geometry for converging cameras, (b) and (c) show a pair of images with superimposed corresponding points and their epipolar lines (in white). The motion between the views is a translation and rotation. In each image, the direction of the other camera may be inferred from the intersection of the pencil of epipolar lines. In this case, both epipoles lie outside of the visible image [2] [3].

3. FFT IMPLEMENTATION ON DSP-FPGA

3.1 Fourier Transform

The essence of the Fourier transform of a waveform is to decompose or separate the waveform into a sum of sinusoids of different frequencies. In other words, the Fourier transform identifies or distinguishes the different frequency sinusoids, and their respective amplitudes, which combine to form an arbitrary waveform. The Fourier transform is then a frequency domain representation of a function. This transform contains exactly the same information as that of the original function; they differ only in the manner of presentation of the information. Fourier analysis allows one to examine a function from another point of view, the frequency domain [1].

The Discrete Fourier Transform (DFT) is described by the following formula:

$$F(k) = \sum_{n=0}^{N-1} f(n) e^{-j\frac{2\pi nk}{N}}, 0 \le k \le N-1$$

Where,

F(k) =Fourier Transform of k^{th} point

N =Total no of Points

f(n) =Value of function f at n^{th} Point

DFT transforms the sequence of N complex numbers x_0. . . x_{N-1} (time domain samples) into the sequence of N complex numbers X_0 , . . . , X_{N-1} called frequency domain samples. If x_0. . . x_{N-1} are real numbers, as they often are in practical applications, then the DFT obeys the symmetry $X_k = X^*_{N-k}$, where the $*$ denotes complex conjugation and the subscripts are interpreted modulo N. Therefore, the DFT output for real inputs is half redundant, and one obtains the complete information by only looking at roughly half of the outputs.

Computation of N-point DFT requires N^2 complex valued multiplications ($4\times N^2$ real valued multiplications). Typical case in digital signal processing is transformation of real valued signals, so the DFT needs only

$2\times N^2$ real valued multiplications. For both cases DFT's computational complexity is O (N^2)[1].

3.2 FFT (Fast Fourier Transform)

FFT algorithm has been implemented on 1024 inputs, 512-complex twiddle factors. The N-point Discrete Fourier Transform (DFT) of a finite duration sequence x(n) is defined as follows.

$$X(k) = \sum_{n=0}^{N-1} x(n) W^{nk}, k = 0,1,..., N-1,$$

where $W = e^{-j\frac{2\pi}{N}}$ is referred as the twiddle factor, N is the transform size and $j = \sqrt{-1}$. The FFT is an efficient algorithm to compute the DFT and its inverse (Cooley and Tukey). It generally falls into two classes: Decimation In Time (DIT), and Decimation In Frequency (DIF). The DIT algorithm first rearranges the input elements in bit reversed order and then builds the output transform. The DIF algorithm first transforms and then rearranges the output values. The basic idea of these algorithms is to break up an N–point DFT transform into successive smaller and smaller transform known as a butterfly (basic computational element). The smallest transform used is a 2-point DFT known as radix-2, it processes groups of 2 samples [9].

1.

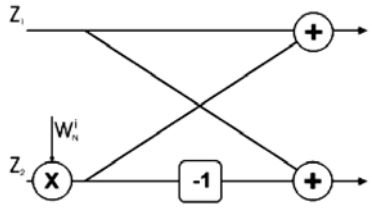


Figure 3.1: Simplified Butterfly operation

To calculate FFT for N number of inputs, Decimation in Time algorithm requires following points.

- L = $\log_2 N$ stages.
- ((N/2) * L) number of complex multiplications.
- (N * L) number of additions.
- (N/2) twiddle factors should be stored.

Number of Points N	Complex Multiplications in Direct Computation N^2	Complex Multiplications in FFT Algorithm $(N/2)*\log_2 N$	Speed Improvement
4	16	4	4.00
8	64	12	5.33
16	256	32	8.00
32	1024	80	12.80
64	4096	192	21.33
128	16384	448	36.57
256	65536	1024	64.00
512	262144	2304	113.78
1024	1048576	5120	204.80

Table 3.1: Calculation multiplications for varying no of Inputs

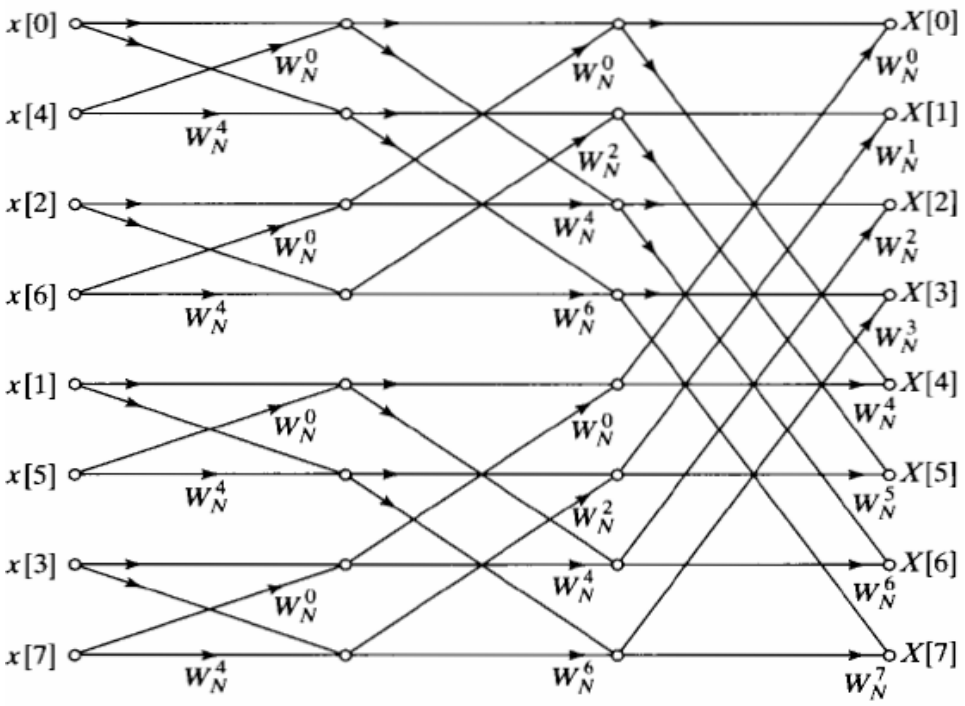


Figure 3.2: Butterfly diagram for FFT algorithm (DIT)

In order that the computation may be done in place, the input sequence must be stored in a non-sequential order. In fact, the order in which the input data are stored and accessed is referred to as bit-reversed order. If (n_2, n_1, n_0) is the binary representation of the index of the sequence x[n] , then the sequence value $x[n_2, n_1, n_0]$ is stored in the array position $X0[n_0,$

n_1, n_2]. That is, in determining the position of x [n_2, n_1, n_0] in the input array, one must reverse the order of the bits of the index n.

0(000) » 0(000)
1(001) » 4(100)
2(010) » 2(010)
3(011) » 6(110)
4(100) » 1(001)
5(101) » 5(101)
6(110) » 3(011)
7(111) » 7(111)

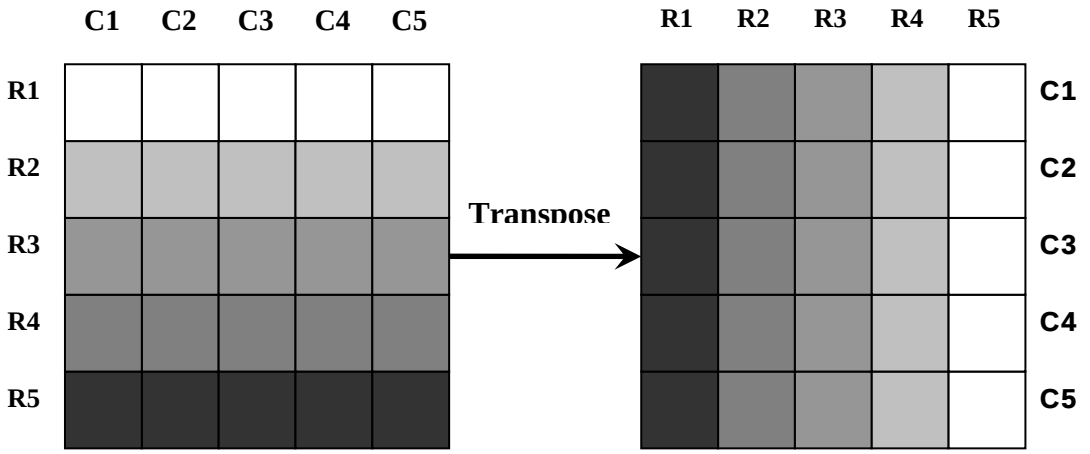


Figure 3.3: Transpose Operation for 2D FFT of image

3.3 2D FFT for Image

As described in above section 3.2, 1D FFT is requires $\frac{N}{2}\log_2 N$ computations. Image processing is also easy in frequency domain. In frequency domain, to generate 2D FFT of image, first 1D FFT of each of the rows is calculated. Then 1D FFT on each of the columns is performed. Some times instead of doing row wise and column wise computations, transpose of the matrices is performed in between two consecutive row wise 1D FFTs. After getting done 2D FFT of image, image enhancement is done in frequency domain filtering and enhancement, after that 2D IFFT on image is done in reverse order of FFT. Figure 3.4 shows frequency domain processing flow by 2D FFT and 2D IFFT [1].

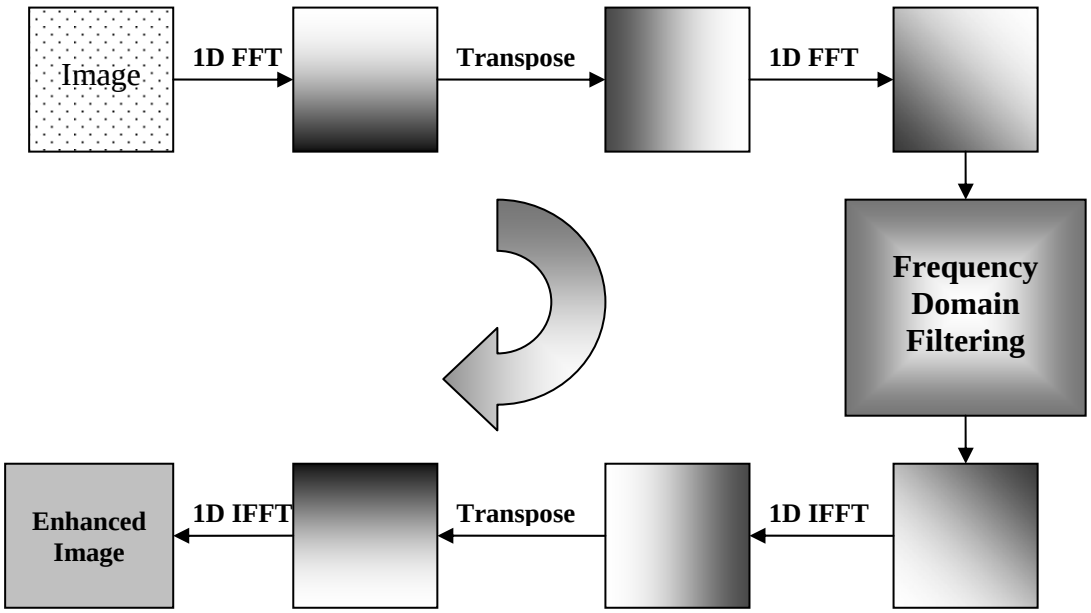


Figure 3.4: Frequency Domain Filtering on Image

3.4 DSP-FPGA Co Processing Architecture

Digital signal processors and FPGA are good solutions for DSP applications, each one has its own pros and cons. Digital signal processors provide fast arithmetic processing, implementation of complex algorithms for various applications. On the other side FPGA serves purpose of real time processing at high speed, reprogram ability (easy updating), parallel execution of simple algorithm and inherent SIMD architecture for image processing like applications [10].

In proposed architecture given here, involves the use of Digital signal processor (Texas Instrument C64xx) [10] and Xilinx Spartan 3 1500L FPGA combination for calculation of FFT algorithm on hybrid DSP-FPGA architecture. Xilinx Spartan 1500L FPGA is used as co-processor for the DSP for generating complex Floating/Fixed point multiplications for the performance enhancement FFT. When Fixed point arithmetic is needed, TI's C64xx DSP processor is used, in case of floating point arithmetic is used , TI's C64xx DSP processor is chosen for compatibility.

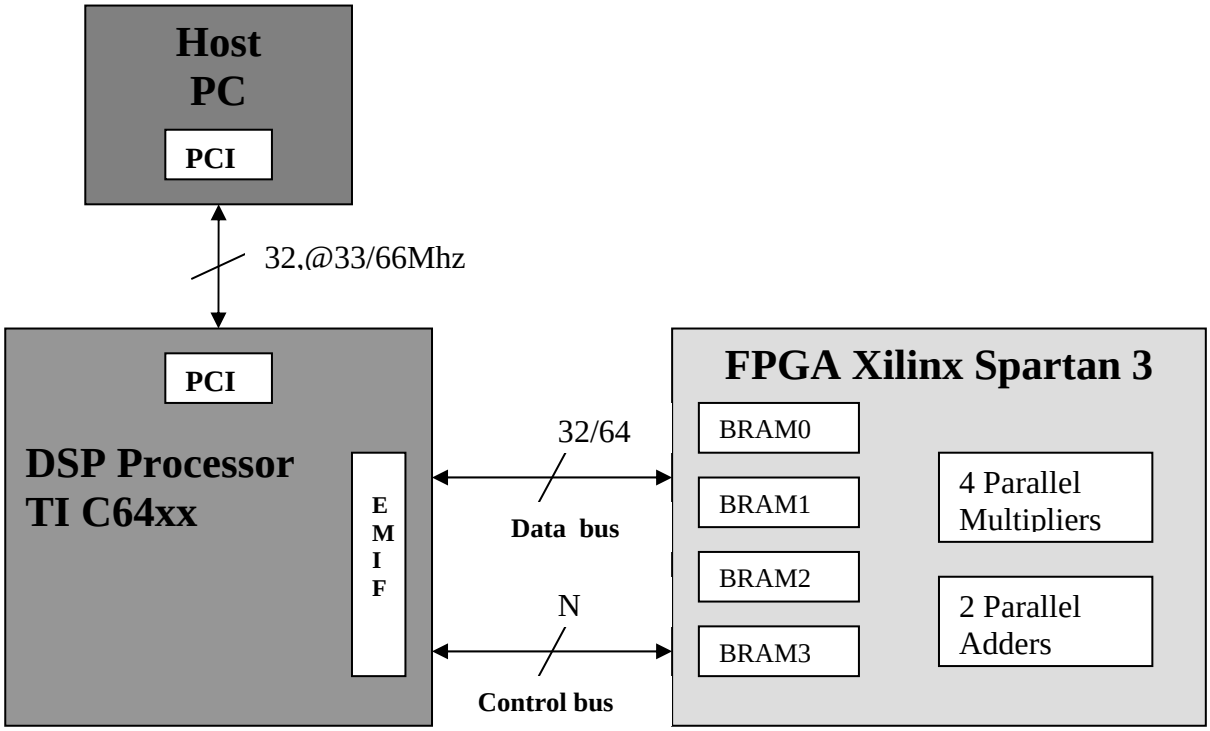


Figure 3.5: Architecture for DSP-FPGA Co-Design

The architecture can be described as below:

1. Image will be transferred from the Host PC to DSP.
2. For the particular algorithm (FFT, Convolution, etc.) complex data arrangement operations is done on DSP and multiplication of complex values is done on the FPGA via EMIF. Other basic operation which can run parallel, are implemented and executed on FPGA,
3. Data transfer to and from FPGA is done through EMIF from DSP side and Block-RAM at the FPGA side.

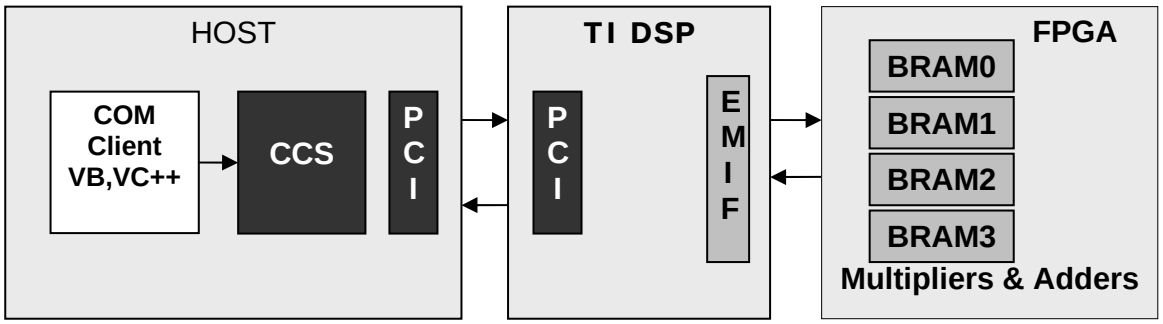


Figure 3.6: Host PC to DSP-FPGA Communication

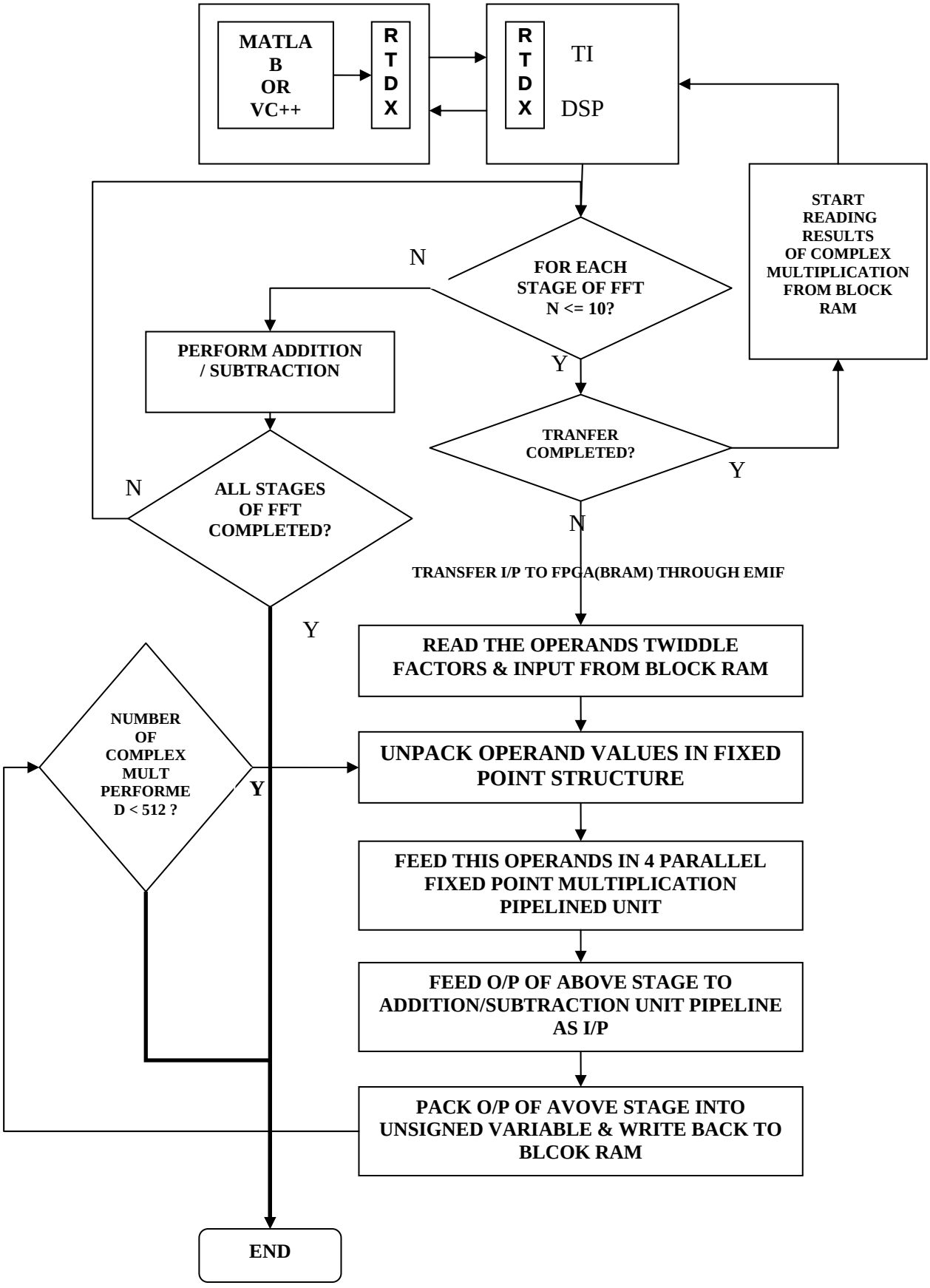


Figure 3.7: Flow chart of 1024 point 1-D FFT.

As described above in this architecture Host PC communicate with DSP using CCS (Code Composer Studio) APIs(Application Programming Interfaces). On the Host PC an application communicates with CCS using COM (Component Object Model) Client like VB (Visual Basic), VC++ (Visual C++), C# and other clients [15]. CCS has capability to communicate with FPGA without intervention of Host PC. TI DSP communicates with Spartan III FPGA using EMIF (Extended Memory Interface). FPGA has no of Block RAMs available, from which 4 Block RAMs are used. Each pair of Block RAM is used for storing complex operands and complex multiplication results. Block RAMs support simultaneous multiple simultaneous read operations and multiple simultaneous write operations on different elements of the Block RAMs. Using this unique feature one can write data to one portion of Block RAM from DSP using EMIF and same time data can be read/write to/from the another portion of Block RAM by proper scheduling of read –write operations from both side (DSP and FPGA) .

3.5 Floating Point Library for Handel-C

Handel-C has supported floating point library to carry out various floating point operations like addition, subtraction, multiplication, division, Square root, etc. and comparison operations. This functionality for FPGA is given in PDK (Platform Development Kit) by Celoxica Ltd. This library supports functions for IEEE 754 single precision floating point number standard. To use Floating point a library needs to be included in the program and **float_pipe.hcl** linked from PDK library [5].

```
#include<float_pipe.hch>
```

Like the name of the header file would suspect, the library uses a pipeline for all the mathematical operations. Since the floating point representation follows the IEEE 754 definition of single precision floating point numbers, the integer and fractional bits do not need to be defined. According to the IEEE 754 standard, a floating point number is build, as shown in figure.

In case of a single precision: ‘sign’ is one bit, ‘exponent' equals 8 bits and `mantissa' 23 bits, adding to a total of 32 bits of word length. The floating point value is calculated by:

$$\mathbf{Value = sign \times 2^{e} \times m}$$

Where:

s = +1 or -1;

e = exponent – 127;

m = 1."Fraction in binary": 1≤ m < 2.

Internally, the floating point value is stored as a structure.

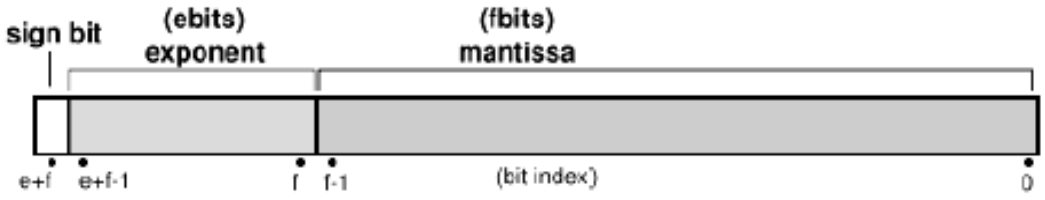


Figure 3.8: IEEE 754 Floating Point number standard

Storage Structure for FPGA in Handel-C Library

```
struct
 {
unsigned int 1 sign;
unsigned int 8 exponent;
unsigned int 23 mantissa;
}
```

3.6 Reasons for migration from Floating point to Fixed point

Floating-point mathematics :

1. Provides excellent dynamic range and accuracy
2. Does not require a designer to worry about overflow, or rounding
3. Is better than fixed-point in every respect but one: it is too expensive in terms of resources and time.
4. Executing floating-point math on the DSP µP, FPGA or ASIC, it is difficult to achieve real-time performance.
5. Requires more DSP µP clock cycles.
6. Requires more logic on an FPGA/ASIC.

Operation	Cycles
FloatPipeFromIntCycles	9
FloatPipeToIntCycles	11
FloatPipeToUIntCycles	10
FloatPipeAddCycles	10
FloatPipeSubCycles	10
FloatPipeDivCycles	27
FloatPipeMultCycles	7
FloatPipeSqrtCycles	26
FloatPipeEqCycles	1
FloatPipeGtCycles	2
FloatPipeLsCycles	2

Table 3.2: Operational Delay in Cycles in Floating Point Library

Applications which require high precision results, wide dynamic range of input-output floating point arithmetic is generally used these types of applications. But drawbacks of floating point processors requirement of higher power consumption, higher cost of recourses (gates), slower than fixed point arithmetic and large implementation size. This leads to use of fixed point arithmetic instead of floating point arithmetic for real time applications and also on FPGA developments.

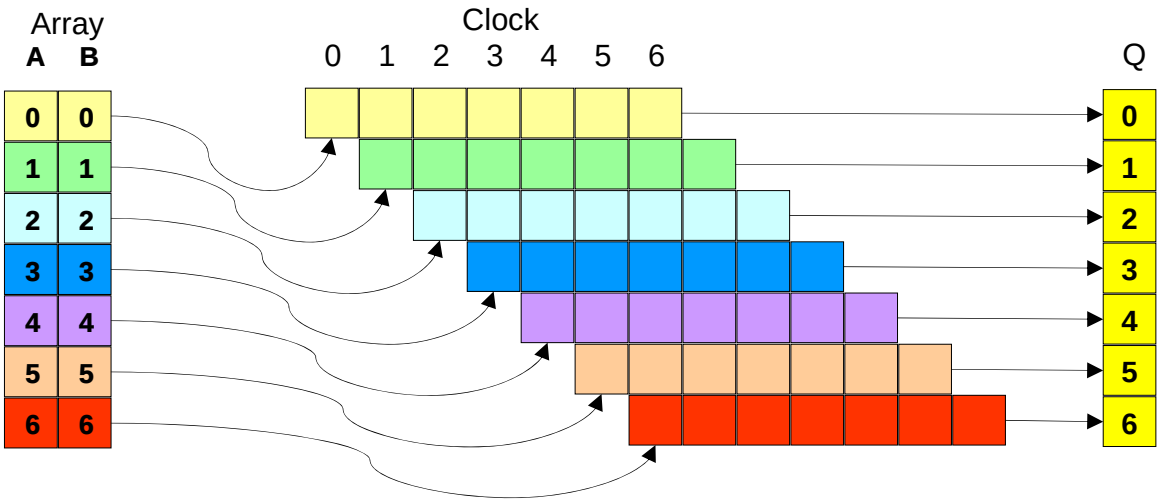


Figure 3.9: Pipeline Floating Multiplication Implementation

3.7 Fixed Point Library for Handel-C

Handel-C has also supported fixed point library to carry out basic various fixed point operations like addition, subtraction, multiplication, division, etc. and comparison operations. This functionality for FPGA is given in PDK (Platform Development Kit) by Celoxica Ltd. This library supports functions for IEEE 754 single precision floating point number standard. To use Floating point a library needs to be included in the program and **fixed.hcl** linked from PDK library [6].

#include<fixed.hch>

As there does not exist any particular fixed point representation, general structure is fixed no of digit before decimal point (Integer Part) and fixed no of digits after decimal point (Fraction part) is used by Handel-C as shown below.

XXXXXX. XXXXXX

Integer: Fraction

So, fixed point number is declared as Fixed (Integer bits, Fraction bits) in Handel-C. In Handel-C Fixed (16, 8) would occupy same no of bits as Fixed (8, 16) or Fixed (12, 12). Using Fixed point numbers overflow – underflow conditions must be explicitly handled by programmer due to range limitation of fixed point numbers [6].

The usual method for porting an application from floating point to fixed-point data types involves mathematical analysis of the algorithms involved to determine the dynamic range of all values. If the analysis shows that the range and precision needed is too great for the memory requirements of the application, say 40 bits are needed but only 32 bit data types are supported on the target architecture, alterations are made to the algorithm to scale the values at various stages of the algorithm so they fit into the memory available. For example, if it was found that the values were always a multiple of ten, all values would be pre-divided by the highest common factor before calculation, and then the final result would be multiplied by the highest common factor. As the project does not have a mathematician at its disposal, and this method could prove timely, this route was not taken.

Embedded processors that use fixed-point arithmetic usually saturate the result of the calculation on overflow. When the result of an operation is greater than the greatest positive number that can be represented, the result is set to the maximum number that can be represented. When the result is less than the largest negative number that can be represented, the result is given as the largest negative number that can be represented. In the best case, overflow causes the signal to be distorted support signed numbers, as well as negative overflow. Addition and subtraction were added for completeness sake and to support saturation of addition and subtraction.

Fixed point migration was needed because research suggested that floating point values not only required more processing than integral values, they also required more gate area on the reconfigurable device. In order to maximize the number of functions on the FPGA gate area had to be conserved. This would allow a greater level of parallelism in the application, especially between the host CPU and the FPGA, which should improve performance. As integral operations execute in less time than floating point operations, further performance gains are expected from migration to fixed-point arithmetic. Also, "all Handel-C conversions start by converting floating-points to integers [fixed-point]".

3.8 3 Stage Complex Multiplication Pipeline

For two complex number A+iB and C+iD, complex multiplication is done as:

Real := AC - BD
Imag := AD + BC
Result:=**Real** + *i***Imag**

Complex multiplication pipeline is divided in four stages

1. **FETCH:** Fetch operands from 4 Block RAMs (Each BlockRAM contains operand in Real-Imaginary operand pair).
2. **MULT:** Same time Fixed/Floating point multiplication of operand fetched at N-1 clock cycle is performed.
3. **ADD:** At this time Fixed/Floating point addition-subtraction is performed on operand fetched at N-2 clock cycle and on which multiplication is already performed.

4. **WRITE BACK:** In last stage of pipeline results of subtraction-addition is write-back to Block RAM for operand fetched at N-3 clock cycle.

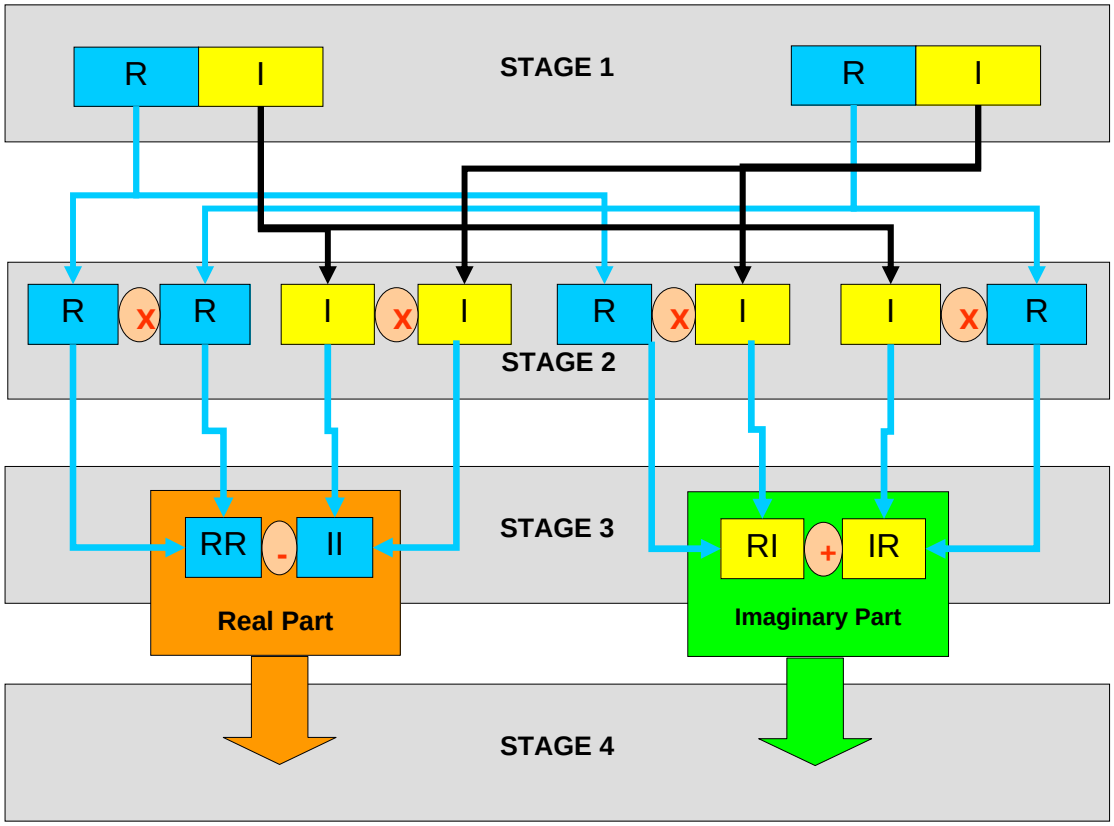


Figure 3.10: Complex Multiplication Pipeline

Operator Pair	Calculation	Cycle
64	1+64+7+10	82
128	1+128+7+10	146
256	1+256+7+10	274
512	1+512+7+10	530

Table 3.3: No of Cycles for variable no of Pairs

Pipeline Overhead

- Initialization Overhead 1 Cycle
- Multiplication Pipeline Delay 7 Cycles
- Addition Pipeline Delay 10 Cycles

18 Cycles

Total Cycles=Initialization (1) + No of Operator Pair (N) + Multiplication Delay (7) + Addition / Subtraction Delay (10)

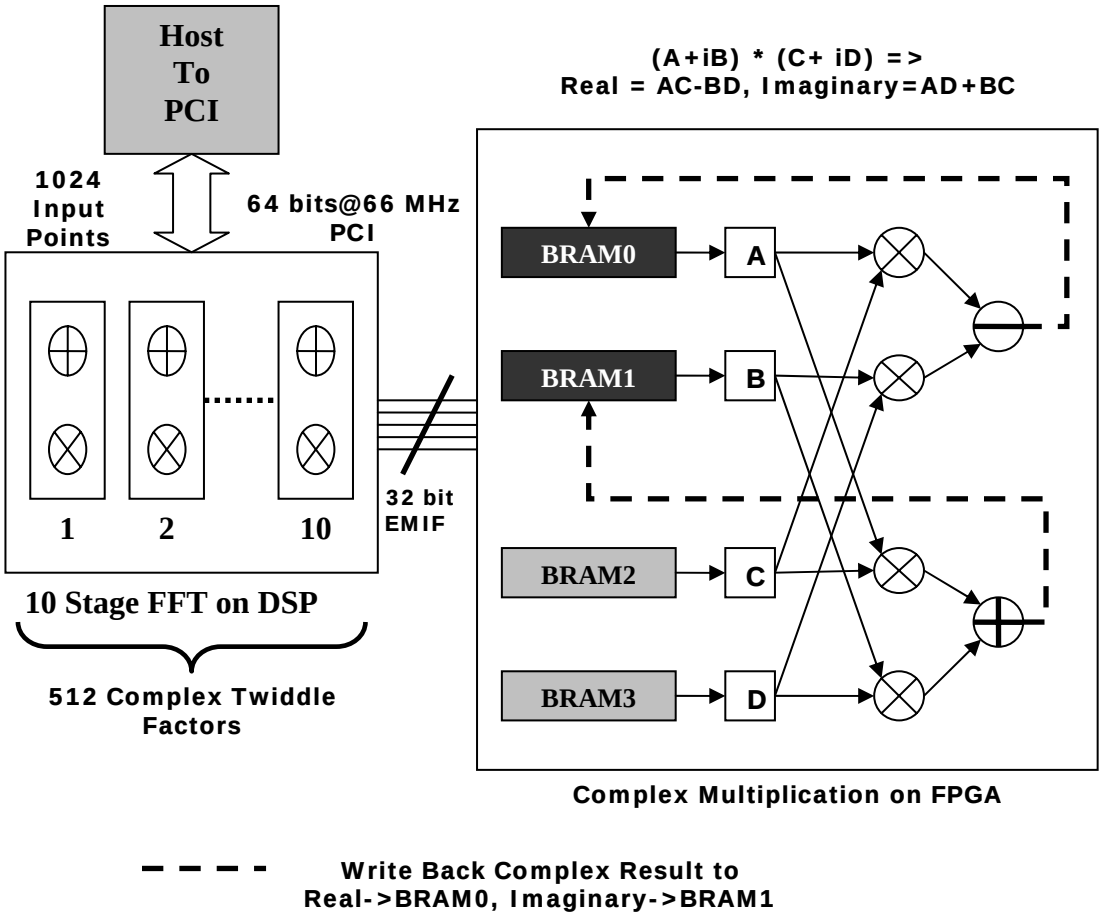


Figure 3.11: 1024 points 1-D FFT on DSP & FPGA architecture

3.9 Results

For calculating 10 stage FFT for 1024 inputs, Total time for execution of FFT algorithm

= Data transfer between DSP and FPGA (only inputs) +
Execution time of multiplication on FPGA +
Data transfer between FPGA and DSP (only outputs).

= 1024 + 515 + 1024

Here one thing is to be noted, that as and when multiplication are getting done in FPGA, simultaneously result of those multiplication are to be sent from FPGA to DSP. So, if we don't consider 515 multiplication time then total time for execution of single stage FFT algorithm will be,

= 1024 + 1024=2,048 for one FFT stage,

= 2048 * 10 =20,480 for all 10 stages,

How much time 10-Stage FFT with 1024 points inputs will take on hybrid architecture, can be shown from following table-5.

Clock cycles	Description
133,000	Clock cycles on Single DSP.
33,062	Clock cycles without multiplications on DSP.
20,480	Clock cycles multiplication on FPGA.
53,542	Clock cycles with hybrid architecture.
(133,000 - 53,542) = 79,458	Clock cycles benefit in comparison with single DSP.

Table 3.4 : Time taken on Hybrid DSP-FPGA architecture

It can be concluded that rather using image processing on single DSP processor, if such algorithms are executed on hybrid architecture such as DSP & FPGA, in which FPGA will serve the purpose as a co-processor then number of clock cycles can be saved.

4. STEREO IMAGE MATCHING ON FPGA

4.1. Introduction

The final accuracy of the DEM is dependent on conjugate point identification in the stereo images. Hence image matching is the most important task in DEM generation in digital mode. In general the major difficulties in automatic image matching are due to temporal changes of the data sets, different view angles, scale changes between the images and sensor differences etc.

Hierarchical image matching is the most used methodology among available all strategies stated as in previous chapter. This method has advantage of improving of speed up many different approaches by guiding search through progressively finer resolutions. The details of an approach are shown in subsequent topics [7].

4.2. Hierarchical Image Matching Technique

Objects represented in the image space may vary enormously in the size and extent. In order to identify and qualitatively describe events in the object space, it is necessary to evaluate and combine the image at different scales, a procedure known as the multi-space technique. By smoothing the original image with a Gaussian low pass filter of varying sizes results in images at various scales (levels of hierarchy) [19]. At each scale, the corresponding images called image pyramids. Selection of optimal number of pyramids depends on the viewing angle of stereo pair used, terrain undulations and the seed point selection in the matching process. After forming the image pyramids hierarchical matching uses four basic steps at each level to get the final match points at the lowest level.

1. Interest point identification.
2. Local mapping between stereo images.
3. Digital correlation up to sub-pixel accuracy.
4. Blunder detection.

The hierarchical procedure is shown in detail in figure 4.1 and 4.2. At the highest level of pyramid the seed point are identified manually or through an interest operator on reference image and blind correlation of these points in the other image. The match points of particular level will be used

to establish the local correspondence between the stereo pair images at next level.

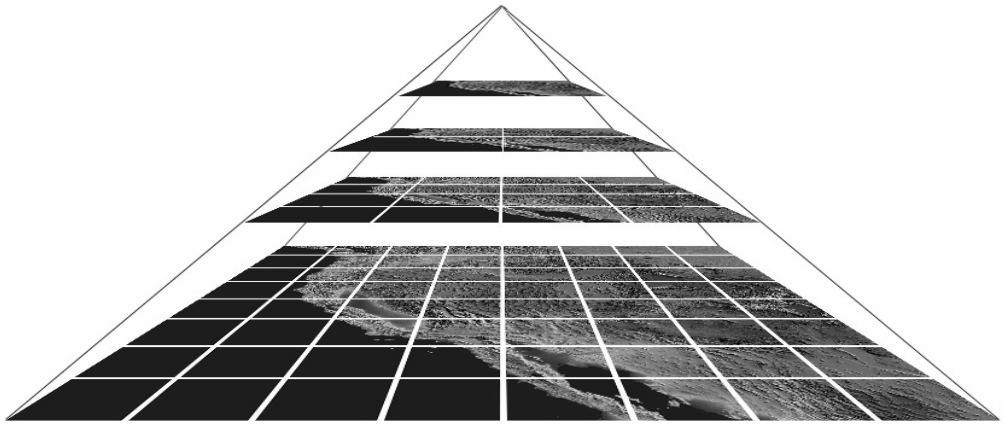


Figure 4.1: Image Pyramid

After this at each level first interest operator is obtained to get some candidate points for match. Local mapping using the previous level's match point establishes a correspondence with the second image for all the interest points. Then digital correlation finds the exact location of the interest points in the second image. Blunder detection eliminates mismatches at each level, if any. This procedure is repeated up to last level that is full resolution. The match points obtained in the last level are the conjugate points for the Digital Elevation Model (DEM).

Hierarchical matching can be performed on raw data directly or the data can be put in epipolar before the image pyramid generation. This has the advantage of using one sided mask. But this has disadvantage of twice the model accuracy in calculating DEM, as we have to go back from epipolar to raw geometry once again after matching, to compute DEM.

4.2.1. Candidate Feature Selection (Interest Operator)

Good feature extraction is a reliable pre-processing step for good image matching. Therefore selecting reliable and accurate approximate values fro succeeding fine correlation attracts ever –increasing interest. In the field of computer vision and pattern recognition many different operators have been developed for feature extraction. Most widely used operators are Moravec operator and Forstner .In this section and improved Forstner operator, in terms of fastness and simplicity in thresholds.

The interest operator has two steps.

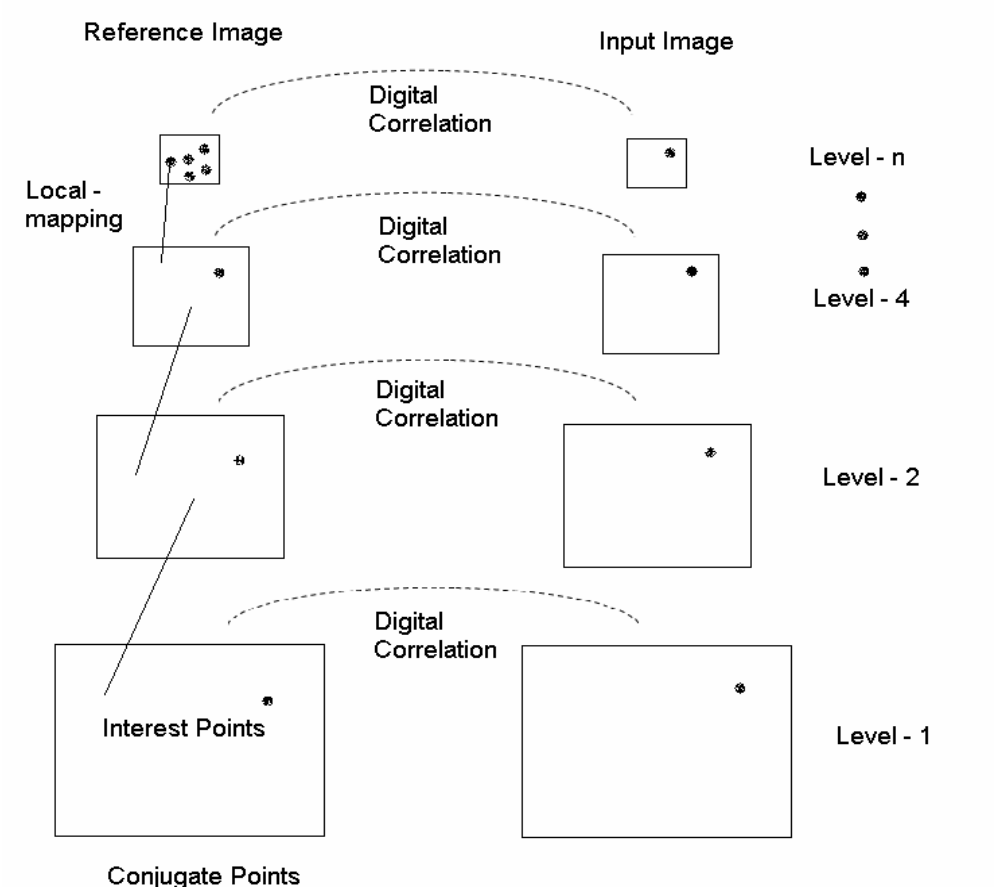


Figure 4.2 Hierarchical Matching using Image Pyramid

A. The Ground Operator

At each point, the four gradients to the neighbouring pixels are calculated (As Figure 4.3.).A point is kept only for the second step, if at least two of the gradients are larger than a threshold Dg. As for the determination of Dg it can be manually set to achieve visually acceptable results, before one have found a method, in which the optimal threshold Dg for different images could be automatically determined.

B. Selection of interest points:

For the second step two versions are possible

Version I

1.1 Interest values (IV) for points selected by the ground operator.

$$IV = \sum_{i=1}^{8} abs(Dg_i)$$

where Dg_i is the difference of grey levels between two adjacent pixels.

1.2 Suppression of local non-maxima, in which compared window is progressively enlarged. Once IV within the compared window (the light hatched parts of the window (Figure 4.4), which indicates suppression windows are 3x3 and 5x5) is greater than the center's value of the suppression window, the comparison stops, and the center's IV is set to zero. The size of the suppression window can be selected accordingly to the density of interest points expected for the results.

Version II

Improvement of the Forstner operator, to achieve speed and accuracy.

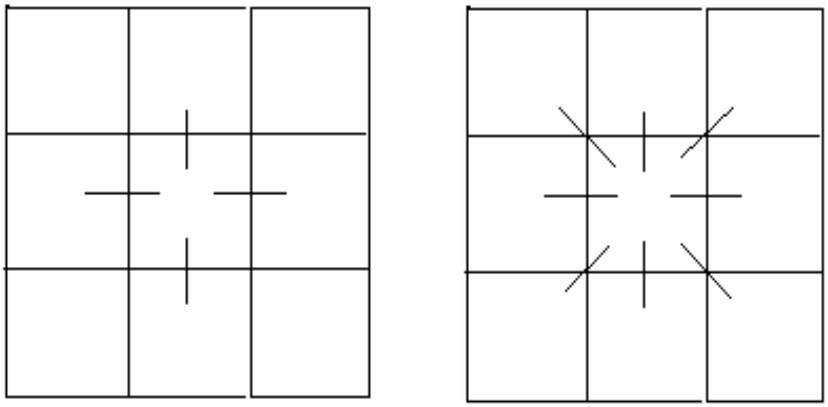


Figure 4.3: Pixel Neighbourhood (4 & 8)

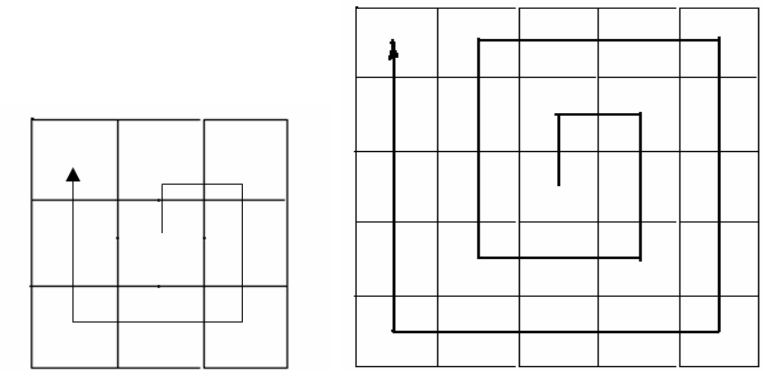


Figure 4.4: Suppression for finding local maxima

The interest operator at each pyramid level gives number of interest points, almost four times of that of number at previous level.

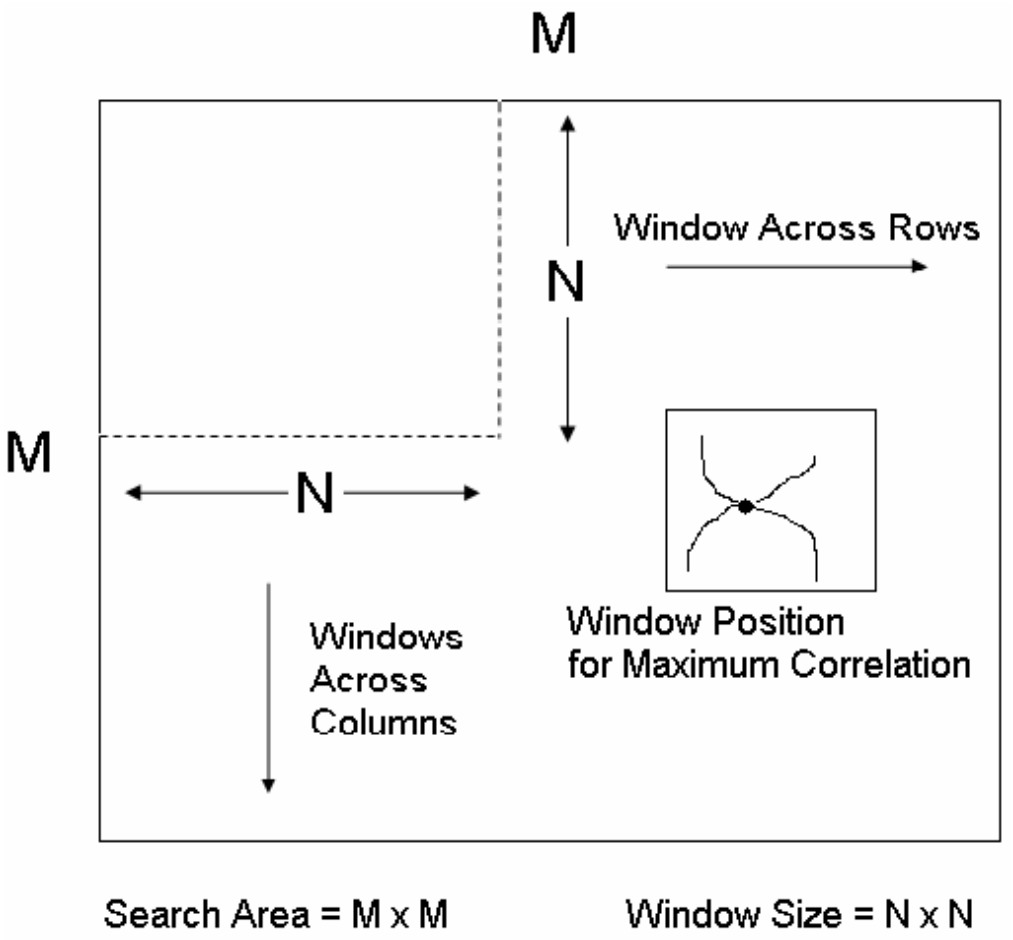


Figure 4.5: Correlation Window in Search Window

4.2.2. Local Mapping

This is basically to establish a local transformation between reference image and the second image of the stereo pair for each interest point located in the reference image .For any interest point ,nearest ten neighbours in the previous level 's match points are selected and a first order polynomial between match points is used for mapping.

$$Xr = a_0 + a_1 X_1 + a_2 Y_1$$

$$Yr = b_0 + b_1 X_1 + b_2 Y_1$$

(Xr,Yr) and (X_l, Y_l) are the scanline ,pixels positions of the match points in left (reference image) and right (second image of stereo pair) images. The polynomial coefficients (a_0, a_1, a_2) and (b_0, b_1, b_2) are obtained through least square solutions on ten points.

Local mapping is generally done using Normalized cross correlation or using SSD, SAD kind of matching functions. Criteria of maximize or minimize depends purely on the algorithm used for matching.

4.2.3. Decision Making and Blunder detection

The following decision making and blunder detection criteria are commonly used in the matching process:

1 Correlation Coefficient

A decision can be taken, whether a point is matched or not by the correlation coefficient magnitude. As explained earlier the correlation coefficient 1 indicates a perfect match and the window & search areas are extracted from same images. However this will not happen anytime, since the image chips used are from different images taken at different times .Hence always the correlation coefficient is less than 1. But a threshold can be fixed on correlation coefficient (>0.5) to identify the probable match point. This is further confirmed by two-way correlation, in the sense, a reverse correlation taking reference chip from input image and search chip from reference image. A point is said to be a match point when the correlation coefficients from the both forward and reverse correlations are within a particular limit (<0.01).

2 Epipolar Geometry

This is another criteria used for strengthening imager matching process by going through object space. The match points in the reference image are transformed to ground by using precise relation ship obtained after space resection. Then these co-ordinates are transformed to second input (image) using the second image's ground to image relationship .Ideally the obtained scanline pixel co-ordinates should match with earlier computed co-ordinate of the same point through conjugate point matching algorithm. If the matching at that point is correct .However the pixel value will not match, since the height is not used in the computation. At least the scanline difference will show the indication of correct match. A point is considered as match point, if the scanline difference lies within the model accuracy.

3 Height Thresholding

The minimum and maximum elevations of the area for which DEM is to be

obtained ,are known apriori, from the maps .These values can also be used for blunder detection /decision making in deciding match points .If the DEM computed for a point is below the minimum elevation of it is above the maximum elevation then the points are rejected.

4 Interactive Editing

Though this is time consuming process ,interactive DEM editing is the powerful tool for blunder detection and correlation of the conjugate points identified are digital correlation .in this the conjugate points are viewed through a stereo display mode. Cross cursors are put at the conjugate points locations in both left and right images. The cursors corresponding to, all the points ,which are correctly matched ,normally appear on the surface of the terrain in the stereo mode .The mismatches can easily identifiable in stereo mode, as they appear out of the model surface or below the model surface . All three points can be identified interactively, and can be deleted or recomputed .This eliminates the blunders, but this process is tedious, if many, mismatches are clustered in a small area.

4.3. FPGA Implementation

Implementation of HIM is done by designing several components like Square Root block, Correlation block and combining them together for execution.

4.3.1. Fixed Point Square Root Implementation

CORDIC Algorithms

CORDIC (Coordinate Rotation Digital Computer) is a simple and efficient algorithm to calculate hyperbolic and trigonometric functions. It is commonly used when no hardware multiplier is available (e.g., simple microcontrollers and FPGAs) as the only operations it requires are addition, subtraction, bitshift and table lookup [15]. The modern CORDIC algorithm has proposed in 1959 by Jack E. Volder. John Stephen Walther at Hewlett-Packard further generalized the algorithm, allowing calculating hyperbolic and exponential functions, logarithm, multiplication, division, and square root.

Originally, CORDIC was implemented using the binary numeral system. In the 1970s, decimal CORDIC became widely used in pocket calculators, most of which operate in binary-coded-decimal (BCD) rather than binary [14].

Mode of operation

CORDIC can be used to calculate a number of different functions. CORDIC is part of the class of "shift-and-add" algorithms. Another shift-and-add algorithm which can be used for computing many elementary functions is the BKM algorithm, which is a generalization of the logarithm and exponential algorithms to the complex plane . CORDIC methods describe essentially the same algorithm that with suitably chosen inputs can be used to calculate a whole range of scientific functions including; sin, cos, tan, arctan, arcsin, arccos, sinh, cosh, tanh, arctanh, log, exp, square root and even multiply and divide. To compute a square-root with CORDIC the number is yielded by multiplying, adding and testing [14].

CORDIC Square Root

Following table shows trace of CORDIC Square root algorithm. Here initial value is **X=12056,** for which square root algorithm is verified [14].

L	2^L	y	x=	12056
		0	initial value	
7	128	0	128 x 128 > 12056	do nothing
6	64	64	64 x 64 < 12056	add 64 to $y_{initial}$ → 64
5	32	96	$(64 + 32)^2$ < 12056	add 32 to last y → 96
4	16	96	$(96 + 16)^2$ > 12056	do nothing
3	8	104	$(96 + 8)^2$ < 12056	add 8 to last y → 104
2	4	108	$(104 + 4)^2$ < 12056	add 4 to last y → 108
1	2	108	$(108 + 2)^2$ > 12056	do nothing
0	1	109	$(108 + 1)^2$ < 12056	add 1 to last y → 109
-1	0.5	And so on.	And so on	and so on

Table 4.1: CORDIC Square Root Example

Code segment shown in Figure 4.6 is for finding square root for integer value and results integer part of square root of any integer number.

```
int sqrt (int x)
{
int base, i, y ;
      base = 256 ;
      y = 0 ;
      for (i = 1; i <= 15; i++)
      {
            y + =  base ;
            if  ( (y * y) > x )
            {
                  y -= base ;  // base should not have been added, so we substract again
            }
            base >> 1 ;      // shift 1 digit to the right = divide by 2
       }
       return y ;
}
```

Figure 4.6: CORDIC Square root Algorithm in C

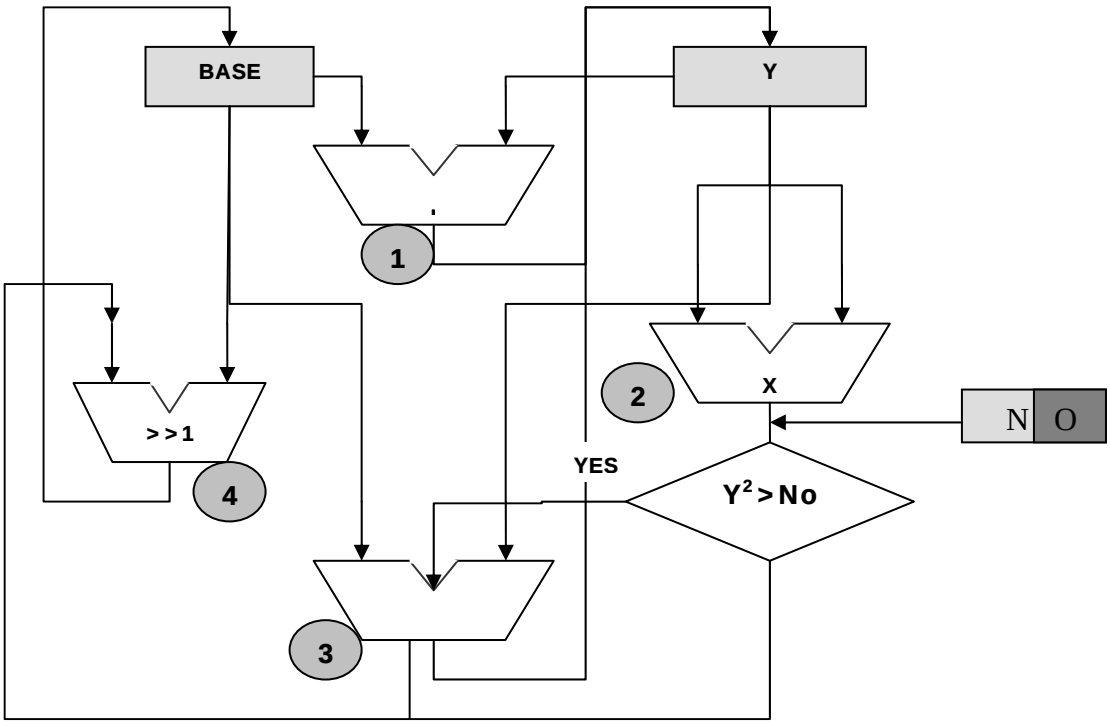


Figure 4.7: Fixed Point Square root Hardware Design

Square root finding procedure for Fixed Point (16, 8) is given as:

BASE=256, Y=0,

Step 1: Y = Base + Y

Step 2: Y= Y * Y

Step 3: Y= Y-Base IF Y2>No

Step 4: Base= Base>>1 Right shift by 1 Division operation

This process continues until Base becomes 0.

4.3.2. Normalized Correlation Implementation

Cross–correlation is a basic statistical approach for the image matching. In Hierarchical matching for conjugate point identification, cross correlation is used as matching cost function. Other cost functions can be sum of absolute differences (SAD), sum of squared differences (SSD),sum of multiplications(SM) and many more [14]. Cross correlation gives a measure of similarity between an image and a template. For hierarchical image matching a window WA of size NxN (N<M) is sliding in a search areal SA of size MxM and best match is selected based on highest correlation value(Ideally for best match correlation value is 1). Following equation shows the correlation formula [1].

$$R(u,v)=\frac{\sum_{x=1}^{N}\sum_{y=1}^{N}(s(u+x,v+y)-\overline{S_{uv}})(w(x,y)-\overline{w})}{[\sum_{x=1}^{N}\sum_{y=1}^{N}(s(u+x,v+y)-\overline{S_{uv}})^2\sum_{x=1}^{N}\sum_{y=1}^{N}(w(u+x,v+y)-\overline{w})^2]^{1/2}}$$

$$u,v=0,1,2,\ldots,M-N$$

Where s(x, y) and w(x, y) are pixel values at location (x, y) of s and w respectively. The value of R changes between 0 to +1, and the R more closer to +1, more similar the two windows will be. R has to be calculated for $(M-N+1)^2$ shift positions. Numerator gives the simple cross correlation, while denominator in equation gives normalizing factor. Correlation can be explained in frequency domain by using Fourier transform as follows:

$$FT^{-1}\{FT(w)FT^{*}(s)\}$$

Where FT is a Fourier transform, FT^{-1} is a inverse Fourier transform,* denotes complex conjugate.

4.3.3. Hierarchical Matching Implementation on FPGA

Hierarchical image matching algorithm as described in above section, here same matching algorithm is proposed to develop on FPGA. Development

of hierarchical matching algorithm starts with image at the top of image pyramid and sparse set of features are detected as describen in [7].

Steps for implementation of Hierarchical Matching Algorithm can be given as below:

STEP 1: Select Ground Points (GPs) using appropriate threshold value of Dg_i as described in section 4.2.1.

STEP 2: Assign IV values to All GPs found by STEP 1.

STEP 3: Filter the GPs by suppression of GPs which not local maximum in their corresponding window. This will generate candidate feature for next stages. Size of suppression window directly affects density of features.

STEP 4: Select appropriate search area (SA) around feature selected in both directions.

STEP 5: Select Appropriate correlation window and for each feature (pixels) in reference image , find pixels in second image in SA using maximum correlation as cost function of similarity measure.

STEP 6: Map the point selected on the second level image of image pyramid and apply STEP 1 to STEP 6 until original image available for processing.

5. RESULTS AND ANALYSIS

5.1. Floating Point Complex Multiplication Pipeline Unit

Floating point numbers can accommodate wide range of numbers ranging from 1.2E-38 to 3.4E38. As proposed DSP-FPGA architecture has suggested that FPGA will be used for carry out complex multiplications. To implement this requirement of complex floating point multiplications on FPGA, PDK of DK Design Suite, Celoxica Ltd. has been used. PDK provides already implemented library for various blocks like Floating Point, Fixed Point libraries for rapid FPGA development. IEEE 754 Floating point library supports various functions/blocks for *Addition, Multiplication, Subtraction, Square root, Division and comparison blocks*. All of these blocks are implemented as pipeline manner, so the library is called ***Pipelined Floating Point Library.*** Latency of each of block is given in Table 3.2. Following table gives the no of clock cycles to calculate complex multiplications for the different no of input operand pairs.

No of Inputs	Result Latency
1	19
2	20
4	22
8	26
16	34
32	50
64	82
128	146
256	274
512	530

Table 5.1: Floating Point Complex Multiplication Pipeline latency input

Floating Point Bits	Gates	Flip-Flops	Latency
32	760764	13058	7+10=17

Table 5.2: Floating Point Pipeline Resources

Floating point library had implemented IEEE 754 floating point standard as described in Chapter 3. For complex multiplication of floating point we require resources as given in table 5.2. As floating point operations multiplication is implemented by using gates instead of inbuilt multipliers, so it requires higher no of resources in terms of gates/flip-flops. Graph given below gives the latency verses no if input pairs.

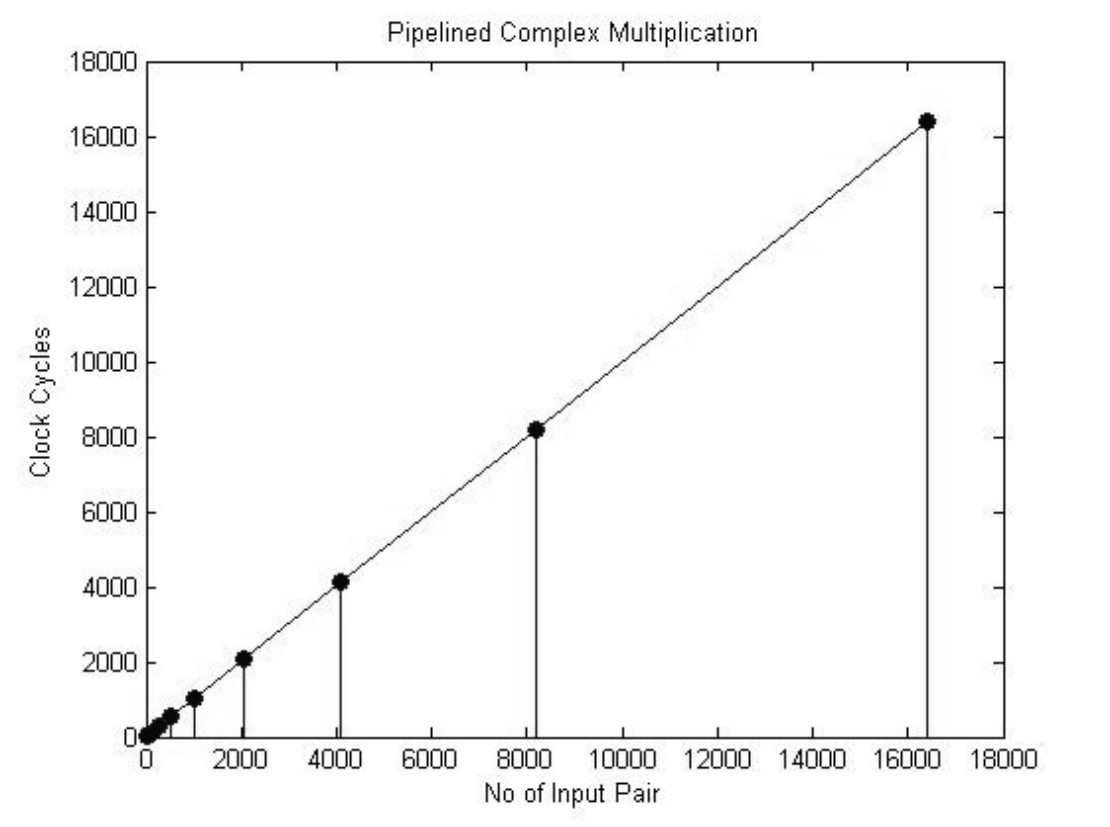


Figure 5.1 Graph of no of inputs-Execution latency

This complex multiplication is useful when use of floating point DSP like TI C6713 is used, which is a floating point DSP by Texas Instruments.

5.2. Fixed Point Complex Multiplication Pipeline Unit

Fixed Point Bits		Gates	Flip-Flops	Max No	Accuracy
Integer	Fraction				
32	16	904397	3924	±2147483647.999984741	1.52588E-05
32	8	514749	3284	±2147483647. 99609375	0.00390625
16	16	526937	2652	±32767.999984741	1.52588E-05

Table 5.3: Fixed Point Complex Multiplication Pipeline

Since implementing floating point operation on FPGAs are never preferred due to high amount of resource usage. So it is always preferred to avoid the floating point multiplications wherever possible. Same complex multiplication operation is implemented on fixed point number representation. Here fixed point multiplication is implemented; because of it is being compatible with TI's C6416 DSP processor. From the figures of

gates being used by fixed point implementation, it can be noticed that the no of gates increase in high amount as the no of bits representing number increases. So it is a tradeoff between *no of resources (gates)* to the *accuracy* in representing number and *range* represented by that no of bits.

Handel-C is also supports ***Fixed Point Library*** for carry out various arithmetic operation on the fixed point numbers of variable no of integer and fraction bits as described in Chapter 3.

5.3. Square Root Unit

To find normalize cross correlation it is required to implement square root block to calculate denominator. Implementation of square root function is done here by using CORDIC as described in Chapter 4. Here in table 5.4 statistics and comparisons for various configurations of square root block are given in Table 5.4.

Here it can be noticeable from the figures shown in table 5.4 that as the no of bits representing increases no of gates used also increases. For example Fixed point number (I:8 , F:8) requires 38027 gates while number with (I:16, F:16) require almost four times gates(135219 gates). It can also observe that as the no of bits increases the delay in finding square root also increases. Delay for finding square root can be given as following equation.

Delay **D= I/2+F+1,** where, **I**=**I**nteger Bits and **F**=**F**raction Bits

Fixed Point		Total Bits	Gates	Flip Flops	Result Latency	Max	Accuracy
Integer	Fraction						
8	1	9	9450	103	14	16	0.5
8	8	16	38027	139	41	16	0.0039062
16	1	17	24006	144	32	256	0.5
16	8	24	64035	178	47	256	0.0039062
16	16	32	135219	219	77	256	1.5258E-05
32	1	33	73178	224	56	65536	0.5
32	8	40	136587	258	68	65536	0.0039063
32	16	48	234974	299	101	65536	1.5258E-05
32	32	64	509675	379	149	65536	2.3283E-10
IEEE 754 Floating Point Number		32	262250	2712	29	3.4E38	----------

Table 5.4 Statistics for Square Root block on FPGA

5.4. Correlation on FPGA

As given in [7], here normalized cross correlation is used as a cost function for similarity measure. Maximum correlation value signifies better match. Correlation value ranges from -1 to +1 whew +1 signifies Perfect match, while -1 signifies no match. To implement correlation function requires fixed size window from both of the images and correlation is performed on that window of values. Correlation value gives the similarity of pixel of one image with pixel of another image.

There are two stages in calculating correlation value.

Window Size	Without Parallel Execution		With Parallel Execution		Clock Savings	
	Total Cycles	SQRT Cycles	Total Cycles	SQRT Cycles	%	%
4x4	377	197	252	126	33.16	36.0
6x6	588	202	396	126	32.65	37.6
8x8	874	202	596	126	31.81	37.6
12x12	1,679	195	1,164	126	30.67	35.3
16x16	2,810	194	1,956	126	30.39	35.0

Table 5.5 Statistics of Correlation Window with Different Size on FPGA

STAGE 1: In this stage average value of both window is calculated. This is carried out by following simple equation for NxN size of window W.

$$Average = \frac{\sum_{i=1}^{N}\sum_{j=1}^{N} W(i,j)}{N * N}$$

STAGE 2: In this stage differences with average value for window W and S are calculated as follows:

Numerator:

$$Numerator = \sum_{i=1}^{N}\sum_{j=1}^{N} (W(i,j) - \overline{W}) \times (S(i,j) - \overline{S})$$

Denominator:

$$Denomimtor = \sum_{i=1}^{N}\sum_{j=1}^{N} (W(i,j) - \overline{W})^2 \sum_{i=1}^{N}\sum_{j=1}^{N} (S(i,j) - \overline{S})^2$$

After this, Correlation is calculated as following equation:

$$C = \frac{N}{\sqrt{D}}$$

Where, C= Correlation, N=Numerator, D=Denominator

Here, Square root is implemented as a one block using CORDIC algorithm as given in 4.3.1, so it is also useable other applications [17].

5.5. Image Matching Results

MATLAB is a one of the best tools for developing image processing algorithms. Here in this thesis work it also helped for verification of algorithms before implementation on FPGA. Initially here results of MATLAB implementation of Hierarchical algorithm is discussed in 5.5.1, then results of FPGA implementation is discussed in section 5.5.2.

5.5.1. Results of Implementation in MATLAB

As the title of dissertation suggests, *Design of FPGA based co-processor for stereo imaging algorithms using Handel-C*, here Hierarchical image matching algorithm and FFT algorithm on DSP-FPGA hybrid architecture is implemented. In order to garner a full understanding of the algorithms, the algorithms are written using MATLAB prior to any hardware development. This provides better understanding of the algorithms and also provides result verification support for hardware implementation. Because in hardware results are available in form of numbers, so visualizing the results is done on MATLAB. Main reason for selection of this algorithm is the speed of the algorithm. The speed of the algorithm is mainly because of the less computation involved in the algorithm and the correlation is calculated for only identified interest points.

Implementation

As part of the implementation the hierarchical algorithm has been implemented for 5 levels of calculations. The whole implementation has been done in MATLAB (version2007-b). The algorithm has been applied on high resolution satellite images. The algorithm works as follows:

Step 1. From the image pair highest level of image pyramid is calculated.

Step 2. For top level reference image interest points are identified.

Step 3. For all these interest-points correlation is performed within search window in second image.

Step 4. All the interest points and its matches are mapped to image pair of next lower level.

Step 5. For all lower level image pair Step 1 to 4 are performed.

Step 6. Finally all matched pixel pair are used for disparity map generation.

Results

This implementation of the algorithm has been applied to get the disparity map. The input image and matches found are shown in preceding Figure 5.2 , Figure 5.3 and Figure 5.4. To calculate disparity map correlation algorithm has been implemented and disparity map calculated from those interest points are used as input and disparity map is calculated. This is shown in Figure 5.5 and ground truth disparity map is shown in figure 5.6. It can be seen in the Figure 5.5 the disparity map shows most of the objects well distinguished. The correlation window size is 32 and Dg is 10.

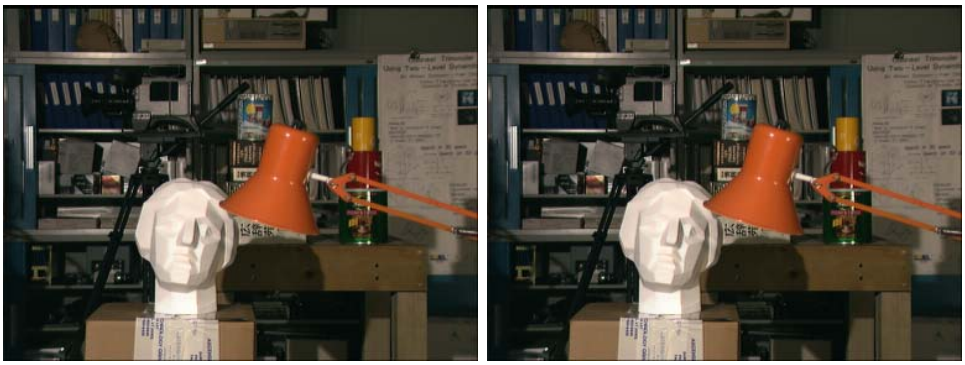
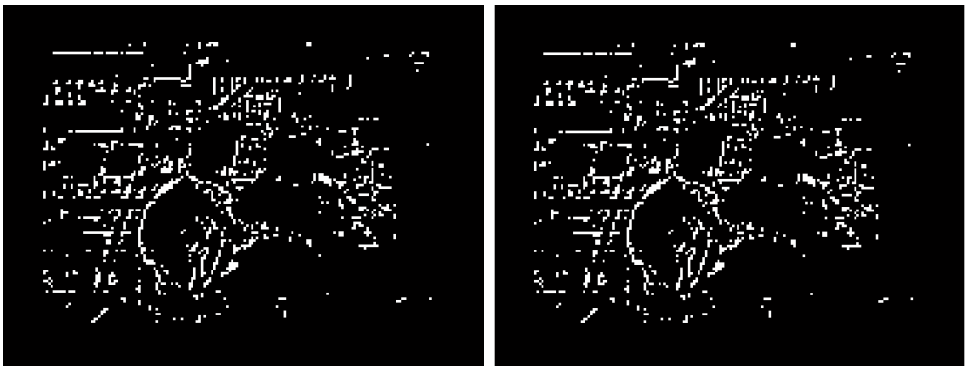


Figure: 5.2 Input stereo image pair.



'A' 'B'

Figure:5.3 –'A' Interest points –'B' matches found window-size:32

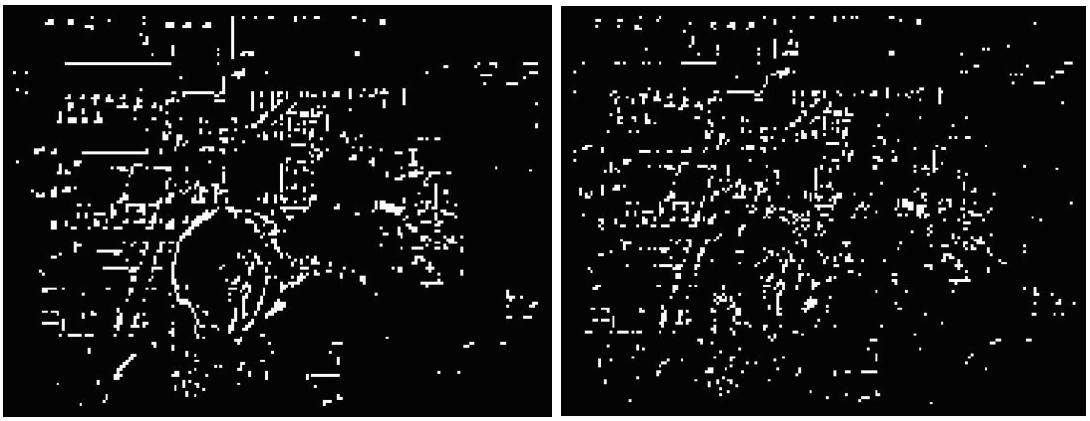


Figure 5.4: Match for window-size 8

Figure 5.5: Disparity map from HIM algorithm

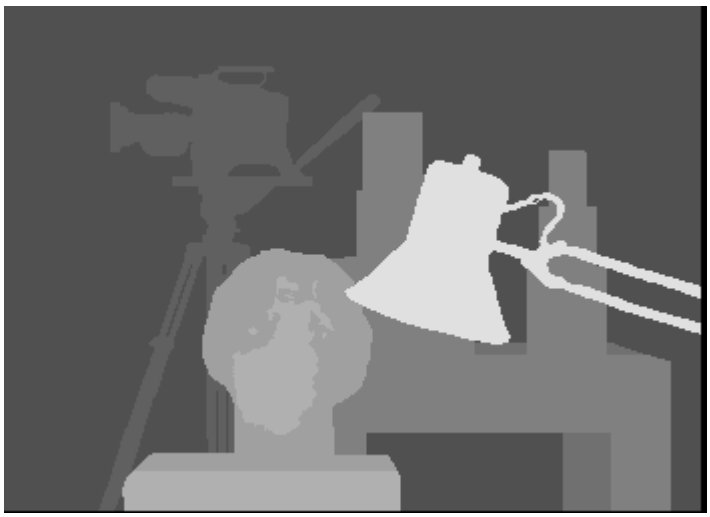


Figure 5.6: Ground truth disparity.

5.5.2. Results of Implementation in FPGA

A detail of stereo image matching using Hierarchical Image Matching (HIM) algorithm is discussed in sections 4.2.1, 4.2.2, and 4.2.3. Implementation of these algorithms on FPGA as in form of Square Root calculation unit, Correlation unit, and is discussed in section 4.3.1 and section 4.3.2. Handel-C code for each of the unit can be found in Appendix-I. Development of Hierarchical Algorithm for size of 61x61 image pair on FPGA is done in Handel-C. Code for Hierarchical Algorithm can be found in Appendix-I.

Following Table 5.6 shows the statistics involved in HIM algorithm. Here, HIM is performed with different size of correlation window on image pair stored in BlockRAM. It can be observed that as the window size of correlation increases computations increases for correlation, which results into delay in results. After some threshold of window size, if further window size is increased, execution latency starts decreasing. Reason behind this scenario is that as the size of window increases the possible candidate pixels for matching decreases due to neighbouring constraints for calculations (Correlation requires same dimensions of both windows). So, ***As the size of window increases, candidate pixels for matching decreases***. Table 5.8 shows statistics for the HIM algorithm using variable size of search windows. Here two variations are made, one with search window of 32x32 size and another with 16x16 size. Here for both of the variations results of matching are verified with MATLAB results of same algorithm. From the statistics from the table, one can observe that if search window size is reduced by half, computations are theoretically reduced by 4 times.

Window Size	Clock Cycles		Clock Savings %
	Without Parallel Execution	With Parallel Execution	
4x4	15,954,847	9,097,592	42.98
6x6	20,174,627	11,414,380	43.42
8x8	25,522,887	14,211,936	44.32
12x12	29,454,673	16,138,222	45.21
16x16	27,139,314	14,760,600	45.61

Table 5.6 Matching Algorithm on FPGA Statistics with and without Parallel Execution

	Without Parallel Execution	With Parallel Execution	Gate Reduction %
Gate Utilization	3,351,412	3,159,054	5.74

Table 5.7: Gate Utilization for HIM.

From the Table 5.7 it can be seen. Here practically also the computations are decreasing by that much amount. But preprocessing of image pair like calculation of average map for image pair, candidate pixel identification for matching and other overhead of calculations remains same for the any search window size. These overhead calculations restrict to achieve 4 times faster execution because they are independent from the search window size and always require to calculate for any window size. Table 5.7 gives the statistics for gate utilization of HIM on FGPA with/without parallel execution. Here, it can be observed from the Table 5.8 that for reduction in cycles in percentage is increasing the as the window size increasing, which also supports the observations and conclusions from the Table 5.6.

Window Size	Search Window		Reduction of Cycles %
	Window=32	Window=16	
4x4	9,097,592	3,551,136	60.97
6x6	11,414,380	3,894,968	65.88
8x8	14,211,936	4,195,376	70.48
12x12	16,138,222	3,097,126	80.81
16x16	14,760,600	1,497,720	89.85

Table 5.8 Matching Algorithm on FPGA Statistics with Search Window of size 32 and 16

So from Table 5.8 data it can be observed that if the possible match is highly predictable (maximum pixel disparity can be predicted from the previous results or can be predictable using mathematical calculations using camera parameters and epipolar geometry) then one can finally decide optimum window size that requires less computations and faster matching of candidate pixels is possible.

6. CONCLUSION

It is a challenging task to develop complex computer vision algorithm with constrains and meet real time requirement of the applications on FPGA. It is necessary to implement the efficient customized architecture rather than simply porting software implementation on hardware.

As the aim of the dissertation, *Design of FPGA based Co-Processor for Stereo Imaging Algorithms Using Handel-C*, hierarchical image matching algorithm has been studied and implemented on FPGA. For implementation on FPGA various basic building block like fixed-point square root unit, complex multiplication unit for fixed point and floating point arithmetic are implemented with pipeline to increase throughput. For image matching, normalized cross correlation function is implemented as similarity measure. Finally all this building blocks are used for implementation of *Hierarchical matching algorithm* on stereo image pair. Fixed-Point complex multiplication unit (TI's C64xx) and Floating-Point complex multiplication unit (TI's C67xx) is designed for FFT calculation on DSP-FPGA hybrid architecture.

Selection between Floating point calculations and Fixed point calculations is dependant on the various parameters like accuracy, execution latency, resources available, range of number that can be represented for requirement of applications. Selection is always a trade-off among these parameters.

Programming hardware using high level language Handel-C provides many advantages like rapid hardware design, faster debugging, less programming efforts then languages like VHDL.

REFERENCES

Books

[1] R. Gonzales, P. Wintz, Woods, *Digital Image Processing*, Addison-Wesley, 1987.

[2] Richard Hartley, Andrew Zisserman, *Multiple View Geometry in Vision*, 2002.

[3] Ian H. Witten, Eibe Frank, *The Geometry from Multiple Images,* MIT Press, 2000.

[4] Handel-C Language Reference Manual, Celoxica Ltd., 2004.

[5] Floating Point Reference Manual, Celoxica Ltd., 2004.

[6] Fixed Point Reference Manual, Celoxica Ltd., 2004.

Papers

[7] B. Gopala Krishna, "Conjugate Point Identification and Image Matching Methods", *Tutorial on "Satellite Photometry and Surveying ",INCA GB and SAC-ISRO, Ahmedabad, Dec 2000*

[8] Stephen D. Brown, R.J. Francis, J. Rose, Z.G. Vranesic, "Filed Programmable Gate Arrays", 1992.

[9] Moore, M "A DSP-based real time image processing system*", In the Proceedings of the 6th International conference on signal processing applications and technology*, Boston MA, August 1995.

[10] Texas Instruments, Inc.: "C64xx Floating-Point Benchmarks," *Houston, TX, 2000.*

[11] Bruce Draper, Walid Najjar, Wim Bohm, "Compiling and Optimizing Image Processing Algorithms for FPGA's", *In IEEE International Conference on Application-specific Architectures and Processors*. 2000. Boston.

[12] Woodfill, J. and B.V. Herzen. "Real-Time Stereo Vision on the PARTS Reconfigurable Computer", *In IEEE Symposium on Field-Programmable Custom Computing Machines*. 1997. Napa, CA: IEEE Press.

[13] P. M. Athanas and A. L. Abbott “Processing images in real time on a custom computing platform”, *In IEEE International Conference on Application-specific Architectures and Processors*. 1998..

[14] R. W. Hartenstein and M. Z. Servit, “Field-Programmable Logic Architectures, Synthesis, and Applications*”, In IEEE Symposium on Field-Programmable Custom Computing Machines*, pages 156– 167. Springer-Verlag, Berlin, 1994.

[15] Downton, A. and Crookes, D., “Parallel Architectures for Image Processing”, *Electronics& Communication Engineering Journal*, vol. 10, pp. 139-151, June 1998.

[16] Venkateshwar Rao Daggu. “Design and Implementation of an Efficient Reconfigurable Architecture for Image Processing Algorithms using Handel-C”. Masters Thesis, UNLV, 2003.

[17] J. Bannur and A. Varma. “A VLSI implementation of a square root algorithm”, *In IEEE Symposium on Comp. Arithmetic, pages 159{165. IEEE Computer Society Press*, Washington D.C., 1985.

[18] D. Scharstein and R. Szeliski, “A taxonomy and evaluation of dense two frame stereo correspondence algorithms”, International Journal of Computer Vision, vol. 47, no 1, pp-7-42, April 2002.

[19] Olivier Faugeras, Bernard Hotz, Herve Mathieu, Thierry Vieville, Zhengyou Zhang, “Real time correlation based Stereo: Algorithm, Implementations and Applications*”, The International Journal of Computer Vision, 4(3):180-195, INRIA*, January 1993.

Web Sites

[20] SystemC. SystemC homepage. www.systemc.org/.

[21] Celoxica Ltd. www.celoxica.com

[22] Altera. www.altera.com

[23] Mentor Graphics Ltd. www.mentor.com

[24] Handel-C Forum, http://www.doc.ic.ac.uk/~akf/handel-c/cgi-bin/forum.cgi

[25] Verilog www.verilog.com

[26] Cite Seer :http://citeseer.ist.psu.edu/

[27] www.middleburry.edu

APPENDIX-A
HANDEL-C SOURCE CODE

1. Floating Point Complex Pipelined Multiplication Unit

```
#include<stdlib.hch>
#include<fixed.hch>

set family= XilinxSpartan3;
set clock = external "P35";


#define N 512
#define M 32


void main()
{

FS P1_R1[2],P1_R2[2],P1_I1[2],P1_I2[2];
FS P2_R1[2],P2_R2[2],P2_I1[2],P2_I2[2];
FS FloatRR[2],FloatII[2],FloatRI[2],FloatIR[2];
FS ResultImag[M],ResultReal[M];
int 16 Real[M],Imag[M];

unsigned int 1 FUP,FUPI,FM,FMI;
unsigned int 9 k;
unsigned int 10 RI;

/*static unsigned 10 ML = FloatPipeMultCycles;
static unsigned 10 SL =N+FloatPipeSubCycles;*/

static mpram Fred
{
        ram <unsigned 32> P1[N*2];
   ram <unsigned 32> P2[N*2];

}Result with {block="BlockRAM"};

static mpram Fred
   {
   ram <unsigned 32> P1[N];
   ram <unsigned 32> P2[N];
   }BRAM_R1={
   0x40156543,0x400CCCCD,0x40833333,0x40866666,0x42306666,
/*RAM can be initlized or written witrh 32 bit Hex data */
0x40166666,0x400CCCCD,0x40833333,0x40866666,0x42306666,0x4230CCCC}with
{block="BlockRAM"};
 static mpram Fred
   {
   ram <unsigned 32> P1[N];
   ram <unsigned 32> P2[N];
   }BRAM_I1={
0x40833333,0x40866666,0x42306666,0x4230CCCD,0x43CA0CCD,0x43CA199A,... /*RAM can be
initlized or written witrh 32 bit Hex data */

 ......,0x43CA0CCD,0x43CA199A,0x40166666,0x400CCCCD}with {block="BlockRAM"};
 static mpram Fred
   {
   ram <unsigned 32> P1[N];
   ram <unsigned 32> P2[N];
   }BRAM_R2={
   0x40866666,0x42306666,0x4230CCCD,0x43CA0CCD,0x43CA199A,0x40166666, /*RAM can be
initlized or written witrh 32 bit Hex data */
0x43CA0CCD,0x43CA199A,0x43CA0CCD,0x43CA199A,0x40166666,0x400CCCCD}with
{block="BlockRAM"};
 static mpram Fred
   {
   ram <unsigned 32> P1[N];
```

```
   ram <unsigned 32> P2[N];
   }BRAM_I2={
   0x42306666,0x4230CCCD,0x43CA0CCD,0x43CA199A,0x40166666,0x400CCCCD, /*RAM can be
initlized or written witrh 32 bit Hex data */
,0x40166666,0x400CCCCD,0x40833333,0x40866666}with {block="BlockRAM"};
par
{
FUP=0;
FUPI=1;
FM=0;
FMI=1;
k=0;
RI=0;
}
do
{
   par
      {
        FUP++;
        FUPI++;
        FM++;
        FMI++;
          k++;
          RI++;
        P1_R1[FUP]=FloatUnpackFromInt32(BRAM_R1.P1[k]);
        P1_R2[FUP]=FloatUnpackFromInt32(BRAM_R2.P1[k]);
        P1_I1[FUP]=FloatUnpackFromInt32(BRAM_I1.P1[k]);
          P1_I2[FUP]=FloatUnpackFromInt32(BRAM_I2.P1[k]);

          P2_R1[FUP]=FloatUnpackFromInt32(BRAM_R1.P1[k]);
          P2_R2[FUP]=FloatUnpackFromInt32(BRAM_R2.P1[k]);
          P2_I1[FUP]=FloatUnpackFromInt32(BRAM_I1.P1[k]);
          P2_I2[FUP]=FloatUnpackFromInt32(BRAM_I2.P1[k]);
        if (ML)
         {
            ML--;
         }
         else
         {
         delay;
         }
         if (SL!=0 && ML==0)
         {
            SL--;
         }
         else
         {
         delay;
         }
                  /*R1*R2*/
FloatPipeMult (P1_R1[FUPI],P1_R2[FUPI],&FloatRR[FM],3,&P1_R1[FUPI],
&P1_R2[FUPI],__clock,FloatPipeChipTypeGeneric);
                  /*I1*I2*/
FloatPipeMult (P1_I1[FUPI],P1_I2[FUPI],&FloatII[FM],3,&P1_I1[FUPI],
&P1_I2[FUPI],__clock,FloatPipeChipTypeGeneric);
                  /*R1*I2*/
FloatPipeMult (P2_R1[FUPI],P2_I2[FUPI],&FloatRI[FM],3,&P2_R1[FUPI],
&P2_I2[FUPI],__clock,FloatPipeChipTypeGeneric);
                  /*R2*I1*/
FloatPipeMult (P2_R2[FUPI],P2_I1[FUPI],&FloatIR[FM],3,&P2_R2[FUPI],
&P2_I1[FUPI],__clock,FloatPipeChipTypeGeneric);
        /*(R1*R2)-(I1*I2)*/
FloatPipeSub(FloatRR[FMI],FloatII[FMI],&ResultReal[(k<-5)-1],3,&FloatRR[FMI], &FloatIR[FMI]);
         /*(R1*I2)+(I1*R2)*/
FloatPipeAdd(FloatRI[FMI],FloatIR[FMI],&ResultImag[(k<-5)-1],3,&FloatRI[FMI], &FloatIR[FMI]);
         Result.P1[RI+0]=(unsigned)FloatPackInInt32(ResultReal[k<-5]);
         Result.P2[RI+N]=(unsigned)FloatPackInInt32(ResultImag[k<-5]);
                  }

}while (ML+SL!=0);
```

```
/*Block Main End*/

/*k=0;
do
{
   par
   {
   Real[k<-5]= FloatPackInInt32(ResultReal[k<-5]);
   Imag[k<-5]= FloatPackInInt32(ResultImag[k<-5]);
   k++;
   }
}while(k<32);*/
delay;

}
```

2. Square Root Calculation Unit & Correlation Calculation Unit

```
#include<fixed.hch>
#include<stdlib.hch>
set clock = external "P35";
#define N 16
#define CW 4
#define INT 32
#define FRAC 8
typedef FIXED_SIGNED(INT,FRAC) FS;
typedef FIXED_SIGNED(INT*2+1,FRAC) FSS;
ram <unsigned int 8> Left[N][N]={
   139,49,11,210,196,28,0,21,141,177,27,45,14,45,152,254,...
/* RAM can be initialized or can be written */
   ...,248,67,50,213,23,252,170,53,137,21,131,114,26,47,26,211};
ram <unsigned int 8> Right[N][N]={
   86,199,174,93,199,110,15,25,142,41,108,174,206,203,201,6,
   /* RAM can be initialized or can be written */
   16,240,22,39,193,156,34,180,97,189,27,9,235,208,36,163};

//SQRT//
void SQRT(FSS* F)
{
FIXED_UNSIGNED(INT*2+1,8)  FXA,Base,Y,FXY,AA;
unsigned int 6 i;

par
{
FXA=FixedCastSigned(FIXED_ISUNSIGNED,INT*2+1,FRAC,*F);
Base = FixedLiteral(FIXED_ISUNSIGNED,INT*2+1,FRAC,4294967296.0);
Y    = FixedLiteral(FIXED_ISUNSIGNED,INT*2+1,FRAC,0.0);
i=1;
}
do
          {

        Y=FixedAdd(Y,Base);
        AA=FixedMultUnsigned(Y, Y);

        par
        {
           if  ( FixedGT(AA, FXA) )
           {
              Y=FixedSub(Y, Base);  // base should not have been added, so we substract again
           }
           Base=FixedRightShift(Base,1);      // shift 1 digit to the right = divide by 2
           i++;
        }
     } while(i<=41);

   *F=FixedCastUnsigned(FIXED_ISSIGNED,INT*2+1,FRAC,Y);


//return (Y);
```

```
}

//SQRT//
void main()
{
unsigned int 5 i,j;
unsigned int 1 Index,IndexI;
FS NumSUM,Den1,Den2,SumL,SumR,L[2],R[2],AvgL,AvgR,Num,FP1,FP2,FPMUL1,FPMUL2,T1,T2;
FSS Denom,Corr,D1,D2,Numerator;
par
{
NumSUM = FixedLiteral(FIXED_ISSIGNED,INT,FRAC,0.0);
Den1   = FixedLiteral(FIXED_ISSIGNED,INT,FRAC,0.0);
Den2   = FixedLiteral(FIXED_ISSIGNED,INT,FRAC,0.0);
Num    = FixedLiteral(FIXED_ISSIGNED,INT,FRAC,CW*CW);
i=0;
j=0;
SumL=FixedLiteral(FIXED_ISSIGNED,INT,FRAC,0.0);
SumR=FixedLiteral(FIXED_ISSIGNED,INT,FRAC,0.0);
}
//Calculation AVG
do
{
   do
   {
        par
      {
      T1.FixedIntBits=(signed)(0@Left[i<-4][j<-4]);
      T2.FixedIntBits=(signed)(0@Right[i<-4][j<-4]);
      }
      par
      {
      SumL=FixedAdd(SumL,T1);
      SumR=FixedAdd(SumR,T2);
      j++;
      }
   }while(j<=(CW-1) );
   par
   {
   i++;
   j=0;
   }
}while(i<=(CW-1));
par
{

AvgL=FixedDivSigned(AvgL,Num);
AvgR=FixedDivSigned(AvgR,Num);
//Calculation Correlation

   i=0;
   j=0;
}
do
{
   do
   {

      par
      {

         FP1.FixedIntBits =(signed)(0@Left[i<-4][j<-4]);
         FP2.FixedIntBits =(signed)(0@Right[i<-4][j<-4]);
      }
      par
      {
         FP1 =FixedSub(FP1,AvgL);
         FP2 =FixedSub(FP2,AvgR);
      }
      FPMUL1=FixedMultSigned(FP1,FP2);
```

```
      NumSUM = FixedAdd(NumSUM,FPMUL1);
      par
      {
      FPMUL1=FixedMultSigned(FP1,FP1);
      FPMUL2=FixedMultSigned(FP2,FP2);
      }
      par
      {
      Den1   = FixedAdd(Den1,FPMUL1);
      Den2   = FixedAdd(Den2,FPMUL2);
      j++;
      }
   }while(j<=(CW-1));
   par{
   i++;
   j=0;
   }
}while(i<=(CW-1) );
Numerator=FixedCastSigned(FIXED_ISSIGNED, INT*2+1, 8, NumSUM);
D1=FixedCastSigned(FIXED_ISSIGNED, INT*2+1, 8, Den1);
D2=FixedCastSigned(FIXED_ISSIGNED, INT*2+1, 8, Den2);
Denom=FixedMultSigned(D1,D2);
SQRT(&Denom);
Corr =FixedDivSigned(Numerator,Denom);
delay;
}
```

3. Hierarchical Image Matching Algorithm of FPGA

```
#include<stdlib.hch>
#include<fixed.hch>

#define X 61                                    // Image width
#define Y 61                                    // Image height
#define CorrWin 6                                 // Window size
#define ss 32                                   // Search space size
#define OffSet CorrWin/2                            // Dynamic offset
#define Thresold 20                               // Ground point threshold
#define N 16
#define CW 6
#define INT_SQR 32                                 // Number of integer bits in fixed point number
#define FRAC_SQR 8                                  // Number of fraction bits in fixed point number

typedef FIXED_SIGNED(INT_SQR*2+1,FRAC_SQR) FSS;             // Typedefinition of fixed signed
number  - (64,8)
typedef FIXED_SIGNED(INT_SQR,FRAC_SQR) FS;                  // Typedefinition of fixed signed
number  - (32,8)
typedef FIXED_UNSIGNED(16,8) FU;                        // Typedefinition of fixed unsigned number
- (16,8)


set clock =external "P1";                               // Operating clock given on pin - 1

extern macro expr abs(a);                                // Absolute macro definition

                                                // Left Image data
ram <unsigned 8>  ArrLeft[X][Y]
={104,95,93,102,115,128,122,119,117,117,118,117,118,129,.....
/* RAM can be initialize using 8 bit unsigned numbers or writing RAM */
.....,81,86,87,86,88,84,80,82,105,133,145,153,156,157};

                                                // Right Image data
ram <unsigned 8> ArrRight[X][Y]
={92,97,96,94,100,109,125,126,120,118,119,120,118,117,130,.....
/* RAM can be initialize using 8 bit unsigned numbers or writing RAM */
.....,82,83,88,88,87,89,84,80,85,113,137,149,155,159};

ram <unsigned 12> IV_Point[X][Y];                   // Interest point array
ram <unsigned 40> AvgMapL[X][Y];                    // Average map for left image
```

```
ram <unsigned 40> AvgMapR[X][Y];              // Average map for right image
ram <unsigned 8> I_J_REFI_REFJ[256][4];       // Correlation array

void SQRT(FSS* F)                              // Square root function definition
{
unsigned int 6 i;
FIXED_UNSIGNED(INT_SQR*2+1,8)  FXA,Base,YY,FXY,AA;

par{
FXA=FixedCastSigned(FIXED_ISUNSIGNED,INT_SQR*2+1,FRAC_SQR,*F);
Base = FixedLiteral(FIXED_ISUNSIGNED,INT_SQR*2+1,FRAC_SQR,4294967296.0);
YY    = FixedLiteral(FIXED_ISUNSIGNED,INT_SQR*2+1,FRAC_SQR,0.0);
i=1;
}

do
{

   YY=FixedAdd(YY, Base);
   AA=FixedMultUnsigned(YY, YY);
   par{
        if( FixedGT(AA, FXA) )
             {
                  YY=FixedSub(YY, Base);  // base restoration
             }

             Base=FixedRightShift(Base,1);      // shift 1 digit to the right = divide by 2
             i++;
        }
} while(i<=41);

*F=FixedCastUnsigned(FIXED_ISSIGNED,INT_SQR*2+1,FRAC_SQR,YY);
}




void main()
{

int 32 SumL, SumR;
int 16 P;
int  8 P1,P2;
signed   int 8 condition_i,condition_j,condition_ii,condition_jj;

unsigned int 6 i,j,ii,jj;
unsigned int 11 Index,Count;
unsigned int 8 temp,XX,YY,GP[2][512],ti,tj;
unsigned int 12 temp1,temp2;
unsigned int 3 count;
unsigned int 12 Mask[3][3];
unsigned int 24 SQR;
unsigned int 8 k;
unsigned int 6 var1,var2,var3;
unsigned int 6 II,JJ,IV_counter;
unsigned 1 suppresed,flag;
unsigned 6 refi,refj,refi_start,refi_end,refj_start,refj_end,mi,mj;

FS FP1,FP2,Avg,FPMUL1,FPMUL2,CorrSum,Den1,Den2,AvgLS,AvgRS,TempSQR,AvgL,AvgR,Num;
FSS Denom,FinalCorr,D1,D2,Numerator,GlobalCorr;
par
{
   Num = FixedLiteral(FIXED_ISSIGNED,INT_SQR,FRAC_SQR,49.0);
   GlobalCorr= FixedLiteral(FIXED_ISSIGNED,INT_SQR*2+1,FRAC_SQR,-1.0);
   k=0;
   Index=0;
   suppresed = 1;
   flag=0;
   SumL=0;
   SumR=0;
```

```
   IV_counter=0;
}
for (i=OffSet;i<61-OffSet;i++)                          // Ground point calculation start
{
   for (j=OffSet;j<61-OffSet;j++)
   {
      Mask[0][0]=0@ArrLeft[i-1][j-1];
      Mask[0][1]=0@ArrLeft[i-1][j];
      Mask[0][2]=0@ArrLeft[i-1][j+1];

      Mask[1][0]=0@ArrLeft[i][j-1];
      Mask[1][1]=0@ArrLeft[i][j];
      Mask[1][2]=0@ArrLeft[i][j+1];

      Mask[2][0]=0@ArrLeft[i+1][j-1];
      Mask[2][1]=0@ArrLeft[i+1][j];
      Mask[2][2]=0@ArrLeft[i+1][j+1];
                  if(i==6 && j==9)
                  {
                        delay;
                  }

IV_Point[i][j]=abs(Mask[1][1]-Mask[0][0])+abs(Mask[1][1]-Mask[0][1])+ abs(Mask[1][1]-
Mask[0][2]) +abs(Mask[1][1]-Mask[1][0]) +abs(Mask[1][1]-Mask[1][2]) +abs(Mask[1][1]-
Mask[2][0]) +abs(Mask[1][1]-Mask[2][1]) +abs(Mask[1][1]-Mask[2][2]);

                  if(IV_Point[i][j] == 0)
                  {
                        delay;
                  }
         par{
               SumL=0;
               SumR=0;
           }
         for (ii=0;ii<=6;ii++)
         {
            for(jj=0;jj<=6;jj++)
            {
               par
               {
                  SumL+=(signed)(0@ArrLeft[i+ii-3][j+jj-3]);
                  SumR+=(signed)(0@ArrRight[i+ii-3][j+jj-3]);
               }
            }
         }
         par
         {
            AvgL.FixedIntBits=SumL;
            AvgR.FixedIntBits=SumR;
         }
         par{
            AvgL=FixedDivSigned(AvgL,Num);
            AvgR=FixedDivSigned(AvgR,Num);
           }
         par{
            AvgMapL[i][j]=((unsigned)(AvgL.FixedIntBits))@((unsigned)(AvgL.FixedFracBits));
            AvgMapR[i][j]=((unsigned)AvgR.FixedIntBits)@((unsigned)AvgR.FixedFracBits);
           }
   }

}                                                       // Ground point calculation over
for (i=OffSet;i<61-OffSet-1;i++)                        // Interest point calculation start
{
   for (j=OffSet;j<61-OffSet-1;j++)
   {
                  par{
         count =0;
         temp1=0@ArrLeft[i][j];
         }
      temp2=0@ArrLeft[i-1][j];
      if (abs(temp1-temp2)>Thresold)
```

```
        {
        count++;
        }
      temp2=0@ArrLeft[i+1][j];
      if (abs(temp1-temp2)>Thresold)
        {
        count++;
        }
      temp2=0@ArrLeft[i][j-1];
      if (abs(temp1-temp2)>Thresold)
        {
        count++;
        }

      temp2=0@ArrLeft[i][j+1];
      if (abs(temp1-temp2)>Thresold)
        {
        count++;
        }



      if(count>=2)
      {
        par
        {
        GP[0][Index<-9]=0@i;
        GP[1][Index<-9]=0@j;
        Index++;
        suppresed = 0;
        }
        Mask[0][0]=IV_Point[i-1][j-1];
              Mask[0][1]=IV_Point[i-1][j];
              Mask[0][2]=IV_Point[i-1][j+1];
              Mask[1][0]=IV_Point[i][j-1];
              Mask[1][1]=IV_Point[i][j];
              Mask[1][2]=IV_Point[i][j+1];

              Mask[2][0]=IV_Point[i+1][j-1];
              Mask[2][1]=IV_Point[i+1][j];
              Mask[2][2]=IV_Point[i+1][j+1];

                        if( (Mask[0][0] > Mask[1][1]) ||  (Mask[0][1] > Mask[1][1]) ||
(Mask[0][2] > Mask[1][1]) || (Mask[1][0] > Mask[1][1]) || (Mask[1][2] > Mask[1][1]) ||
(Mask[2][0] > Mask[1][1]) || (Mask[2][1] > Mask[1][1]) || (Mask[2][2] > Mask[1][1]))
                        {
                suppresed = 1;
                        }

                        else
                        {
                        delay;
                        }


        if(suppresed == 0)
        {
                        par
          {
            GlobalCorr= FixedLiteral(FIXED_ISSIGNED,INT_SQR*2+1,FRAC_SQR,-1.0);
                        IV_counter++;
            flag=1;
            II=i;
            JJ=j;
          }

           par
           {
           condition_ii = (signed )(0@II)-(ss/2);
                         condition_jj = (signed)(0@JJ)-(ss/2);
           condition_i = (signed)(0@II)+(ss/2);
```

```
condition_j = (signed)(0@JJ)+(ss/2);
}
                par
{
     if(condition_ii <= 1)
                {
                        refi_start = (1)+(CorrWin/2);
                }
                else
                {
                                refi_start =abs(i-(ss/2))+(CorrWin/2);
                }
                if(condition_jj <= 1)
                {
                        refj_start = (1)+(CorrWin/2);
                }
                else
                {
          refj_start= abs(j-(ss/2))+(CorrWin/2);
                }

                if(condition_i > X)
                {
          refi_end=X-(CorrWin/2);
                }
                else
                {
                        par
          {
             var1 = (ss/2);
             var2 = (CorrWin/2);
          }
             var3 = abs(var1-var2);
             refi_end = i+var3;
                         }

                if(condition_j > Y)
                {
                        var2 = (CorrWin/2);
          refj_end = (Y)-(CorrWin/2);
                }
                else
                {
                        par
          {
             var1 = (ss/2);
             var2 = (CorrWin/2);
          }
             var3 = abs(var1-var2);
             refj_end = j+var3;
     }
   }

    for (refi=refi_start;refi<=refi_end;refi++)
       {
         for (refj=refj_start;refj<=refj_end;refj++)
         {

              par
              {
                   CorrSum= FixedLiteral(FIXED_ISSIGNED,INT_SQR,FRAC_SQR,0.0);
                   Den1    = FixedLiteral(FIXED_ISSIGNED,INT_SQR,FRAC_SQR,0.0);
                   Den2    = FixedLiteral(FIXED_ISSIGNED,INT_SQR,FRAC_SQR,0.0);
              }
              for(mi=0;mi<=CorrWin;mi++)
              {
                 for(mj=0;mj<=CorrWin;mj++)
                 {
                    par
                    {
                       FP1.FixedIntBits =0@((signed)ArrLeft[i+mi][j+mj]);
```

```
                                FP2.FixedIntBits =0@((signed)ArrRight[refi+mi][refj+mj]);
                                AvgL.FixedIntBits=(signed)(AvgMapL[i][j]\\8);
                                AvgR.FixedIntBits=(signed)(AvgMapR[refi][refj]\\8);
                             }
                             par
                             {
                                AvgL.FixedFracBits=(signed)(AvgMapL[i][j]<-8);
                                AvgR.FixedFracBits=(signed)(AvgMapR[refi][refj]<-8);
                             }

                             par
                             {
                                FP1 =FixedSub(FP1,AvgL);
                                FP2 =FixedSub(FP2,AvgR);
                             }

                                FPMUL1=FixedMultSigned(FP1,FP2);
                                CorrSum = FixedAdd(CorrSum,FPMUL1);

                             par
                             {
                                FPMUL1=FixedMultSigned(FP1,FP1);
                                FPMUL2=FixedMultSigned(FP2,FP2);
                             }
                             par
                             {
                                Den1   = FixedAdd(Den1,FPMUL1);
                                Den2   = FixedAdd(Den2,FPMUL2);
                             }

                                                                  }

                          }


par
{
Numerator=FixedCastSigned(FIXED_ISSIGNED, INT_SQR*2+1, FRAC_SQR, CorrSum);
D1=FixedCastSigned(FIXED_ISSIGNED, INT_SQR*2+1, FRAC_SQR, Den1);
D2=FixedCastSigned(FIXED_ISSIGNED, INT_SQR*2+1, FRAC_SQR, Den2);
}
Denom=FixedMultSigned(D1,D2);
SQRT(&Denom);
 FinalCorr =FixedDivSigned(Numerator,Denom);

                          if (FixedLT(GlobalCorr,FinalCorr))
                          {
                             par
                             {
                                GlobalCorr=FinalCorr;
                                ti=0@refi;
                                tj=0@refj;
                             }
                          }
                          else
                          {
                          delay;
                          }
                       }
                    }
      par
        {
           I_J_REFI_REFJ[k][0]=0@i;
           I_J_REFI_REFJ[k][1]=0@j;
           I_J_REFI_REFJ[k][2]=ti;
           I_J_REFI_REFJ[k][3]=tj;
           k++;
        }
       }
        else
        {
```

```
          delay;
          }
       }
    }
}                                          // Interest point calculation ends

delay;
}
```