A Dissertation

entitled

Development of Parallel Architectures for Radar/Video Signal Processing Applications

by

Amin Jarrah

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the

Doctor of Philosophy Degree in Engineering

Dr. Mohsin M. Jamali, Committee Chair

Dr. Mohammad Samir Hefzy, Committee Member

Dr. Ezzatollah Salari, Committee Member

Dr. Devinder Kaur, Committee Member

Dr. Rashmi Jha, Committee Member

Dr. Patricia R. Komuniecki, Dean
College of Graduate Studies

The University of Toledo

December 2014

Copyright 2014, Amin Jarrah

This document is copyrighted material. Under copyright law, no parts of this document may be reproduced without the expressed permission of the author.

An Abstract of

Development of Parallel Architectures for Radar/Video Signal Processing Applications

by

Amin Jarrah

Submitted to the Graduate Faculty as partial fulfillment of the requirements for the

Doctor of Philosophy Degree in Engineering

The University of Toledo
December 2014

The applications of digital signal processing continue to expand and use in many different areas such as signal processing, radar tracking, image processing, medical imaging, video broadcasting, and control algorithms for sensor array processing. Most of the signal processing applications are intensive and may not achieve the real time requirements. However, the Multi-core phenomenon has been embraced by almost all processor manufacturers and the road to the future is through parallel processing. Now we have many parallel processing platforms that developed for high performance such as:

- Multi-Core/Many-Cores
- Graphic Processing Units (GPU)
- Field Programmable Gate Arrays (FPGA)

This research work involves developing optimized parallel architectures of many signal processing applications such as Extensive Cancellation Algorithm (ECA), Direct Data Domain (D^3), Block Compressive Sampling Matching Pursuit algorithm (BCoSaMP), video processing, Discrete Wavelet Transform (DWT), Particle Filter (PF), and Iterative Hard Thresholding (IHT) on different platforms such as Multi-core, FPGA and GPU. This is performed by exploring opportunities of any computation and storage that can be eliminated to achieve high performance and meet its real time requirements. Different techniques and ideas have also been derived from different advanced fields to increase the intelligibility and the usefulness of our research. A new innovative generalized method is proposed which can be very helpful for many researchers in various areas. Then, the applications have been moved higher ordering through implementing interfaces. This makes it adaptable by specifying all the input parameters of a certain application and fast prototyping through different performance evaluations.

We propose and exploit many parallelization methods and optimization techniques in order to improve the latency, hardware usage, power consumption, cost, and reliability. These parallelization methods predict the data path and the control unit of the application processes. Also, the applications examine into numerical algorithms approaches to provide a transition from the research theory to the practice and to enhance the computational and resource requirements by adapting the certain algorithm for high performance applications. We exploit techniques coupled with high level synthesis tools by enabling rapid development to generate efficient parallel codes from high-level problem descriptions. This will reduce the design time, increase the productivity, improve the reliability, and enable exploration of the design space. Approaches will include optimizations based on mathematical and/or statistical reasoning, set theory, logic, and auto-tuning techniques.

Hardware software co-design for these applications has been performed that pushes performance and energy efficiency while reducing cost, area, and overhead. This has been accomplished by developing a tool called Radar Signal Processing Tool (RSPT).

RSPT allows the designer to auto-generate fully optimized VHDL representation of any of these signal processing algorithms by specifying many user input parameters through Graphic User Interface (GUI). This will offer great flexibility in designing signal processing applications for a System on Chip (SoC) without having to write a single line of VHDL code. RSPT also communicates with Xilinx toolset to check for the available FPGA parts installed with the Xilinx toolset and for executing the VHDL synthesis command chain. Moreover, it utilizes optimization techniques such as pipelining, code in-lining, loop unrolling, loops merging, and dataflow techniques by allowing the concurrent execution of operations to improve throughput and latency. Finally, RSPT provides the designer a feedback on various performance parameters such as occupied slices, maximum frequency, and dynamic range. This offers the designer the ability to make any adjustments to the algorithm component until the desired performance of the overall SoC is achieved.

Parallel approach of IR Video processing is also proposed as it widely used in many numerous processing applications and not achieve the real time requirements. Analysis and assessment of the energy dissipation for heterogeneous Network on Chip (NoC) based Multiprocessor System on Chip (MPSoC) platform running a video application are performed. It identifies the latency, area, and energy bottlenecks of the entire heterogeneous platform including processors, interconnection wires, routers, memory, and caches etc. Also, we propose a new modeling and simulation approach regarding the channel width and buffer sizing which have a strong impact on the performance and the overhead of the chip. This approach monitors the state of each link in the NoC topology. Then, based on the congestion spot and the critical path we can optimize the design by

changing channel width and buffer size until achieving the desired performance.

إلى جناحيّ روحي .. أبي وأمي .. مدادُ قلبي وعمري

الى إخواني وأخواتي .. رفاقُ طفولتي وحُلمي

إليكِ حنيني .. إلى الطفلِ الذي ننتظرْ .. طريقُ جمالٍ وبداياتُ ربيعْ

Acknowledgments

بسم الله الرَّحْمَنِ الرَّحِيمِ

(إِنَّمَا يَخْشَى اللَّهَ مِنْ عِبَادِهِ الْعُلَمَاء)

(Only those fear Allah, from among His servants, who have knowledge)

The Noble Quran (Surat Fatir, Verse 28)

I would like to express my special appreciation and thanks to my advisor Prof. Mohsin Jamali, you have been a tremendous mentor for me. I would like to thank you for encouraging my research and for allowing me to grow as a research scientist. Your advice on both research as well as on my career have been priceless. I would also like to thank my committee members.

To the beautiful and special family I'm lucky to have, you guys enriched me with dreams, hope and the desire to be special and different , my great father Abdel Karim, my sweet mother Hijazia here is my life always for you. My brothers Nabil, Zeyad, Fouad, Yousef, Mohammad, Jalal, Jamal and my lovely sisters Sahar, Samar and Tamara. May allah bless you all.

My Haneen, the wonderful, beloved wife and friend I have. With you, I shared unforgettable moments of happiness, sadness, tiredness and madness as well. You are my support and I promise to be the same for you.

Table of Contents

List of Tables

List of Figures

List of Abbreviations

AMD	Advanced Micro Devices.
APIs	Application Programming Interfaces.
ARM	Advanced RISC Machines.
ASIC	Application Specific Integrated Circuit.
BCoSaMP	Block Compressive Sampling Matching Pursuit algorithm.
BDL	Behavioral Description Language.
CAD	Computer Aided Design.
CG	Conjugate Gradient.
CPU	Central Processing Unit.
CUDA	Compute Unified Device Architecure.
D^3	Direct Data Domain.
DEVS	Discrete Event System Specifications.
DSP	Digital Signal Processing.
DWT	Discrete Wavelet Transform.
ECA	Extensive Cancellation Algorithm.
EDA	Electronic Design Automation.
FIFO	First Input First Output.
FPGA	Field Programmable Gate Arrays.
fps	Frame Per Seconds.
GPU	General Processing Unit.
GUI	Geographic User Interface.
HDL	Hardware Description Languages.
HLS	High Level Synthesis.
HPC	High Performance Computing.
IHT	Iterative Hard Thresholding.
I/O	Input/output.
IP	Intellectual Properties.
IR	Infrared.
ISE	Integrated Software Environment.

MB-IHT	Model-Based Iterative Hard Thresholding.
MIMO	Multiple Input Multiple Output.
NoC	Network on Chip.
OpenC	Open Computing Language.
OTP	One Time Programmable.
PCBs	Printed Circuit Boards.
PF	Particle Filter.
RAM	Random Access Memory.
RGB	Red Green Blue.
RISC	Reduced Instruction Set Computer.
RTL	Register Transfer Level.
SIMD	Single Instruction Multiple Data.
SM	Streaming Multiprocessor.
SoC	System on Chip.
VHDL	VHSIC Hardware Description Language.
VLSI	Very Large Scale Integration.
WSR	Weather Surveillance Radar.
XSG	Xilinx System Generator.

Chapter 1

Introduction

1.1 Problem Statement

Semiconductor technology is approaching its limits, and one way to get future advances in computing is via parallel processing. There are many application areas that can benefit from the parallel computing such as signal processing [1-2], radar tracking [3-5], image processing [6], medical imaging [7-8], video broadcasting [9-11], control algorithms and sensor array processing [12-14]. Most of the signal processing applications are computationally and data intensive. So, efficient implementation is required to achieve high performance and meet their real time constraints.

Researchers in the literature concentrate either on algorithm implementation or on architecture development. Algorithm researchers perform the implementation based on their performance parameters such as correctness, efficiency, accuracy etc. while assuming the underlying architecture would be adequate. On the other hand, architecture researchers' interest in different performance parameters such as speed, hardware resources, communication architecture, memory and bus architecture etc. However, it is very difficult to master both the algorithm implementation and the architecture development at the same time. So, a unifying framework is required to unify the aspects of algorithms, architectures, and software.

Most processing platforms have a complete system [15-17] on the same chip which includes multi-cores, Digital Signal Processors (DSP), circuits, memory banks, send and receive units etc. These platforms become more complex and powerful since it has many heterogeneous components in their design and the tasks they perform. High Performance Computing (HPC) research interest to develop a platform that is capable to achieve high performance for real time applications by improving all the following elements:

- Number of processors
- Processor and memory architecture
- Bus architecture
- Communication with I/O modules
- Energy constrained
- Resilient programming techniques
- Parallel processing friendly environment

Embedded systems come in wide variety of processing elements, memory and other peripherals. The complexity of an embedded system varies from single core processors to multi-core and even many-core architectures, including a wide variety of possible peripherals. Multi-core architectures are relying less on instruction level parallelism as it has reached its limits. Although multi-core solutions offer more parallelism and look promising for general purpose processing, they might not be efficient in performance and flexible enough for certain specific tasks. However, the implementation of these computationally algorithms can be performed using the following parallel processing platforms where each platform has different trade-offs in terms of latency, area, power consumption, cost, and flexibility:

- Multi-core through NoC.
- Field Programmable Gate Arrays (FPGAs).
- Graphic Processing Units (GPUs).

There are many parallel processing approaches available in the literature that can be applied to a given algorithm. Some of the approaches that can be considered are:

- Loop level parallelism: it is a technique where different iterations of the same loop are executed in parallel on different processors. We can also use loop interchanges in the nested loops to maximize parallelism in the innermost loop.
- Data level parallelism can be handled by independently processing data in parallel.
- Function level parallelism can be exploited by dividing functions into various stages and executing them either in parallel or pipeline fashion.

- Pipelining can improve throughput of the function by allowing the concurrent execution of operations within a function.

- Dataflow technique that enables concurrency at the function level to improve throughput and latency.

- In-lining technique that removes all functions hierarchy to reduce the function call overhead.

- Reusing technique to minimize the area and power consumption. Functions and loops will iterate over the same hardware resources each time they execute to maximize the sharing.

- Optimal mapping of arrays on FPGA. FPGA memories have limited access capabilities (read ports and write ports). This imposes dependencies which prevents applying some parallelization techniques. For example, a dual-port RAM, or reconfigured RAM may allow more reads and writes in the same clock cycle.

- Bit-level parallelism where the amount of information of the processor that can be processed per cycle (word size) is doubled. This reduces the number of instructions to perform an operation on variables whose sizes are greater than the length of the available word size. For example, 8-bit processor must add two 16-bit integers by first adding the 8 lower-order bits from each integer, and then add the 8 higher-order bits. So, 8-bit processor requires two instructions to complete a single operation whereas a single instruction is required to complete the operation of a 16-bit processor.

- Instruction-level parallelism where the instruction of a program can be re-ordered and combined into groups to be executed in parallel without changing the result of the program. Recently, processors have multi-stage instruction pipelines where each stage has different action on that instruction in the stage. For example, processor with an N-stage pipeline can have up to N different instructions at different stages of completion. For example RISC processor has five stages: instruction fetch, decode, execute, memory access, and write back. Moreover, some processors can issue more than one instruction at a time which is called superscalar processors. However, this technique can be performed by grouping the instructions together if there is no data dependency between them.

- Task parallelism where the parallelization is performed when entirely different code sections are executed either on the same or different sets of data.

1.2 Parallel Processing Environment

There are many algorithms in different applications that require high level of parallelization due to their intensive computation requirements. Many parallel processing platforms have evolved to achieve higher computational speed such as FPGA architecture, multi-core, and GPU. In the past, an algorithm working on standard single processor may be too slow; we desired a high performance CPU that will be able to execute it in a more efficient way. Despite in the last few years the CPU architecture has been increasingly improved with many processors, providing a higher level of parallelism where a lot of operations could be executed in a tiny time, especially in research fields such as video surveillance [18-19] and for medical image analysis [20]. For this reason, parallel implementations of such applications have been developed to achieve high computational speed.

In order to use all the resources of the parallel processing platform and get high benefit, the application must be parallelized, mapped, and scheduled efficiently to achieve the following performance parameters:

- Minimum run time
- Minimum code size
- Minimum memory consumption
- Meet real time constraints
- Meet power requirements
- Maximum system throughput

Some of the parallel processing platforms are:

- **Multi-core computing:** this platform contains multiple cores on the same chip. This can execute multiple instructions from multiple instruction streams. Each core can be superscalar processor where multiple instructions can be issued from one instruction stream.

- **Symmetric multiprocessing:** This platform contains multiple identical processors that share memory and connect via a bus. However, the bus has a limitation where the contention prevents scaling. The number of processors that can be connected to the bus is dependent on electrical characteristics and delays due to the bus tolerated by the system. So, this type of architecture is not desirable as system needs to meet definite real time constraints. Higher bus bandwidth will also be desired.

- **Distributed computing:** this type of architecture contains multiple processing units. They are connected with a network and may have distributed memory. This is highly scalable compared with the symmetric multiprocessing.

- **Cluster computing:** this type of architecture contains a group of computers that work together closely. These computer clusters are also connected by a network. However, the computer in a cluster can be asymmetric where the load balancing will be more difficult.

- **Massive parallel processing:** this type of architecture contains single computer with many networked processors. It has the same characteristics as clusters where the massive parallel processing has specialized interconnect networks. Also, it can be larger than clusters.

- **Grid computing:** this type is similar to the distributed computing system. It makes use of computers communicating over the Internet to work on a given problem. These types of computing use middleware software layer with the operating system to manage network resources and standardize the software interface.

- **Reconfigurable computing with Field-Programmable Gate Arrays (FPGAs):** FPGAs have large amount of logic, memory, interconnection and other resources that can be programmed and re-configured for a given task using Hardware Description Languages (HDL) such as VHDL or Verilog. However, HDLs need long time for implementation and verification. So, several C to HDL conversion tools have been developed that attempt to emulate the syntax and/or semantics of the C programming language.

- **Graphics processing units (GPU):** GPU has multi-core architecture consisting of hundreds of cores. Each core contains a grids and each grid contains threads. There are threads, thread blocks, and grids of thread blocks that all differentiate themselves based on memory access and kernel execution. A thread block is a group of threads that have the ability to cooperate with each other and communicate via the per-Block shared memory. This type of architecture is attractive for offloading numerically intensive computations. The combination of high-bandwidth memories and hardware that performs floating point arithmetic at significantly higher rates than conventional CPUs makes graphic processors attractive targets for computational intensive algorithms.

In order to implement an algorithm efficiently on any parallel processing platform by utilizing all its features, the following steps must be considered:

- Perform behavioral simulation system using Matlab, C or other languages for a given algorithm.
- Perform detailed analysis and simulation to neglect any unnecessary computation task, storage requirement, and area.
- Implement different methods for a given algorithm on the parallel processing platform to reduce the highly required computation modules and hardware resources.
- Develop approaches to identify any inherent parallelism in various computational modules.
- Decompose computational modules into parallel tasks or processes to run them in parallel as threads.
- Explore ways to efficiently map the algorithm on the target machine.
- Consider the trade-offs between hardware resource utilization, power

consumption and execution time.

- Explore performance measurement based on computation time, code size, power consumption, area, and the throughput.

Nowadays, the researchers widely use FPGAs parallel processing platform [21-27] in implementing and parallelizing computational algorithms as it supports parallel and pipelined architecture. FPGAs are reconfigurable and provide option of rapid prototyping. So, Multi-core, GPU, and FPGA platforms are selected and used extensively in this work for implementing and parallelizing different signal processing algorithms. We also propose a new software tool called Radar Signal Processing Tool (RSPT) as a unifying framework to unify the aspects of algorithms, architectures, and software. It bridges the gap between the algorithm and architecture scientific communities. So, hardware software co-design has been performed that pushes performance and energy efficiency while reducing cost, area, and overhead.

1.3 Dissertation Outline

This dissertation is organized as follows:

- Chapter 2: this chapter provides an overview and background of parallel processing platforms such as GPU and FPGA, and extensive review of many High Level Synthesis Tools (HLSTs).

- Chapter 3: Parallel implementations of Extensive Cancellation Algorithm (ECA), Direct Data Domain (D^3), Block Compressive Sampling Matching

Pursuit algorithm (BCoSaMP), video processing, Discrete Wavelet Transform (DWT), Particle Filter (PF), and Iterative Hard Thresholding (IHT) have been performed. It discusses their efficient parallel implementations on FPGA and GPU platforms.

- Chapter 4: this chapter presents a new software tool called Radar Signal Processing Tool (RSPT) for VHDL auto-generation for any radar signal processing algorithm.

- Chapter 5: Behavior simulation of selected radar signal processing algorithms is provided. It also shows the synthesis and simulation results of their parallel computation, storage resources, area, and power consumption on both FPGA and GPU.

- Chapter 6: Conclusions and future work of this research are provided.

Chapter 2

Parallel Processing Platforms and Tools

Achieving real time requirements for complex applications require a parallel processing platform with multiple processing elements, memories, high bandwidth, caches, etc. Multi-core System on Chip (SoC), Field Programmable Gate Arrays (FPGAs), and Graphic Processing Units (GPUs) parallel processing platforms have been used in our work. They differ in the architecture, number and type of hardware resources, and in their programming environment. Following sections will provide an overview and background for each one.

2.1 Parallel Processing on Multiprocessor Systems-on-Chips (MPSoC)

Single processor may be sufficient for low-performance applications that are typical of early microcontrollers but the increasing number of extensive applications requires multiprocessors to meet their performance goals. Multiprocessor Systems-on-Chips (MPSoC) are one of the key applications of VLSI technology today. It is a parallel processing platform to build complex integrated system. A certain algorithm can be executed on multiple processors simultaneously through two types of interconnections as shown in Figures 2-1 and 2-2:

- Processors connected via bus.
- Processors connected via network.

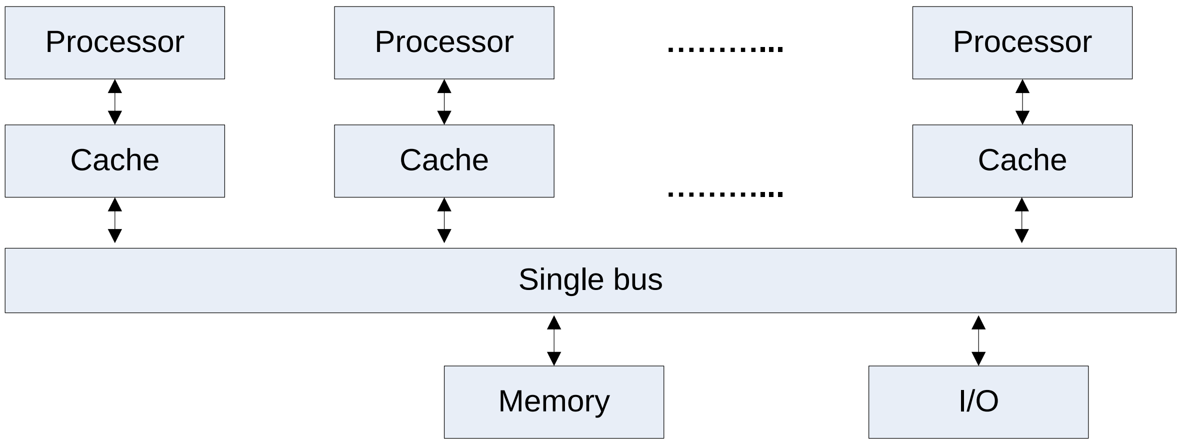


Figure 2-1: Multiprocessors connected by a single bus.

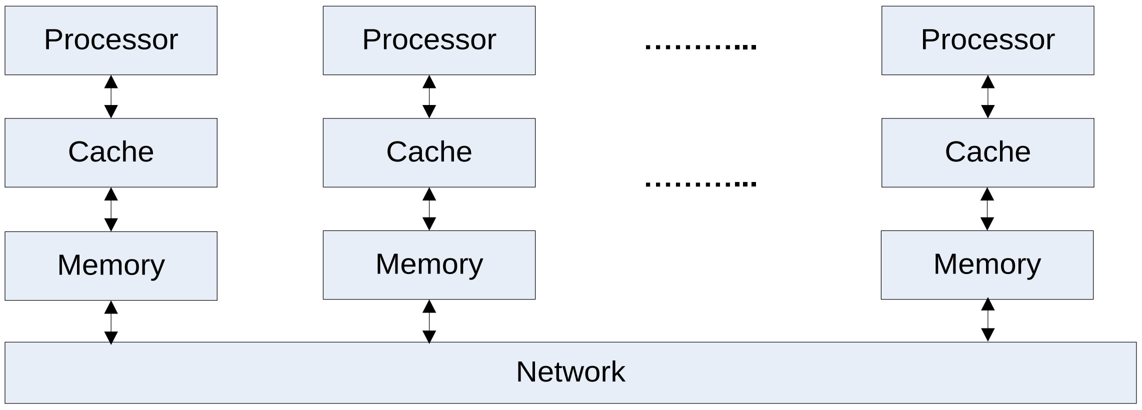


Figure 2-2: Multiprocessors connected by a network

MPSoC has two general communication modes:

- **Single address**: it offers the programmer a single memory address space that all processors share. Processors communicate through shared variables in memory where all processors capable of accessing any memory location via load and store operations.

- **Message passing**: Multiple processors communicate with each other by explicitly sending and receiving messages.

MPSoC requires high bandwidth interconnection between the processors to reduce the execution time as the number of processor increases. However, increasing the bandwidth by adding more channels improves the performance but also increases the total energy dissipation and the chip area. Balancing between reduction in the energy dissipation and the performance trade-off is a serious issue in multi-core systems. Technology advances in the handheld devices with enormous processing capacity required for the multimedia applications impose heavy constraints on the energy dissipation. The energy consumption is becoming a limiting factor for future handheld devices. Most of the emerging MPSoC platforms nowadays are heterogeneous in nature. In most of the MPSoC research, processors are organized around a shared bus, but researchers have launched NoCs communication infrastructure which can be designed to deal with growing system complexity [28-31].

Manufacturing network on chip is an expensive process requiring a thorough analysis and optimization before actually fabricating and outputting the product to the market. So, there is a need for energy analysis for heterogeneous NoC platforms and their memories for real time applications. A comprehensive method is essential to know bottlenecks of the energy dissipation in the handheld devices while running a multimedia application in real time. Generally, research groups have focused on on-chip communication energy consumption for interconnection architecture and homogenous NoC.

2.2 Parallel Processing on a GPU Platform

A Graphic Processing Unit (GPU) is an electronic circuit designed to perform rapid mathematical calculations especially for the purpose of rendering images. GPU has many usages such as mobile phones, personal computers, and game consoles. GPU is a very efficient platform for processing graphics where a high parallel structure of large blocks of data can be performed concurrently. The first GPU is introduced by NVIDIA (GeForce 256) which is capable of processing millions of operations per second. Intel and AMD also provide their own graphic processors. GPUs produced by different manufacturers where each one differs in the following features:

- Streaming Multiprocessor (SM) represents the number and the architecture of available cores.
- Thread global scheduler to manage the context switches between threads during execution.
- Host interface represents the connection between the GPU and the CPU.
- Memory structure and sizes
- Cache levels and sizes
- Clock frequency

- Global memory clock
- Cooling of chip mechanism
- Warranties
- Programming environments and tools

GPU platforms support massive computational power for applications requiring several orders of magnitude higher performance than a conventional CPU. Also, they can achieve high throughput of some computations that exhibit high level of data parallelism.

GPU has emerged as a new paradigm which is evolved into a highly parallel multiprocessor with high computation power. GPU is a collection of many processors with multiple processing units as shown in Figure 2-3. It has high bandwidth memories and hardware resources. GPU is capable of performing floating point operations at high speed. However, the performance gains in GPU architectures depend on effective application parallelization. An efficient implementation on GPU platforms faces two key challenges: the first challenge is that parallel tasks must be identified and extracted from the sequential algorithm. The second challenge is that there must be an excellent match between the extracted tasks and the architecture resources because any mismatch will lead in performance loss and a decrease of resource utilization.

NVIDIA GForce GTX 260 [32] GPU was used in our work. It contains eight thread blocks with 512 concurrent threads in each block. Each thread has separate access to individual memory, counters, registers, etc. It runs in Single Instruction Multiple Data (SIMD) fashion such that all thread blocks execute the same instruction but operate on different data. Threads communicate with each other via the per-block shared memory as shown in Figure 2-3.

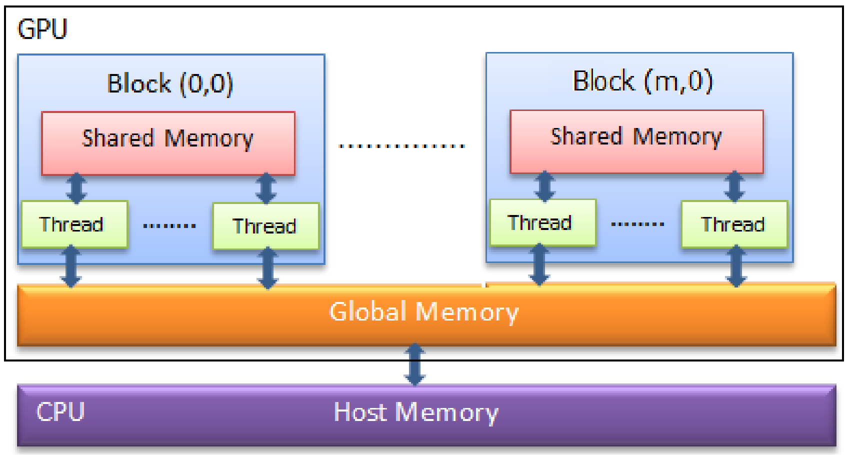


Figure 2-3: GPU architecture

The memory hierarchy of GPU consists mainly of five memory levels. Each one has different properties and uses. So, they must be used efficiently to achieve high performance implementations. The GPU memories are:

- **Constant memory:** is a memory that can be accessed from any thread in any block and grid of the GPU. It is used for only read operation which usually holds the arguments and small data of the kernel functions. It is very limited device resource (few kilobytes), very fast, and off-chip location.

- **Global memory:** is a memory that can be accessed from any thread in the GPU device. This memory is used for both read/write operation. It is used for transferring the data between CPU and GPU. So, it can’t transfer data directly from host to shared memory, registers, or local memory. However, the location of global memory is off-chip and therefore it is slower than shared memory. It has long delays and need a synchronization scheme between threads accesses to ensure correct result.

- **Shared memory:** is a memory that can be accessed by any thread in the same block. This memory is used for both read/write operation where it can't be accessed by the host (CPU). However, it resides on GPU chip and therefore it is very fast but it has a limitation regarding its size.

- **Register file:** is a memory that can be accessed by only one thread. It's used for both read/write operation where all local variables of a thread reside in these registers. However, these registers have a limited resource. Therefore, some of the variables and vectors that don't have enough space in the registers will be moved to the local memory. Table 1.1 summarizes the comparison between different types of memory hierarchy of GPU device.

Table 1.1: GPU memories hierarchy comparison

Memory	Location (on/off chip)	Access	Scope	Lifetime
Constant	Off	R	Grid	Application
Global	Off	R/W	Grid	Application
Shared	On	R/W	Block	Kernel
Local	Off	R/W	Thread	Kernel
Register	On	R/W	Thread	Kernel

CPU and GPU are connected as shown in Figure 2-4. The limitation in the connection is the low speed of transferring data between the host (CPU) and the device (GPU). The data transfer inside the GPU device has high speed but still represents a bottleneck for high performance applications. So, placing data at different levels of memory in the GPU architecture affects the performance.

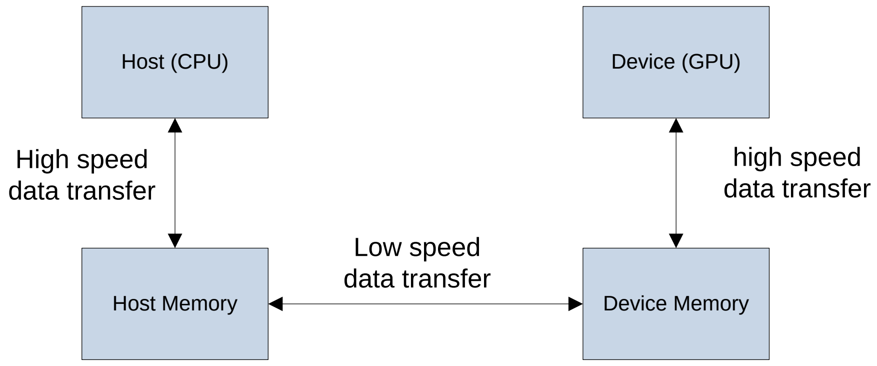


Figure 2-4: CPU-GPU connection.

After moving the data from the host (CPU) to the device (GPU), it is very beneficial to move the data from global memory to shared memory on the GPU part as shared memory resides on the GPU chip. This will help in keeping the needed data for each block in its own shared memory as shown in the following code:

Used shared memory for data storing

```
__global__ void  Function (int m, float *Mat_d, float *V_d)

{  __shared__ float Mat_shared [1024];      //Declared shared memory
   index ←  Thread index
   Mat_shared [index] =Mat_d [index];         //Copy from global to shared
   V_d [index] =Mat_shared [index] /m;}       //Execute the task
```

The shared memory is declared inside the body of the kernel. Then, the data is moved from the global memory to the allocated storage on shared memory for further processing. This will reduce the latency as the data will be available to all the threads in the block inside the GPU chip. This optimization strategy is applied in our implementation.

NVIDIA supports Compute Unified Device Architecture platform (CUDA) and Open Computing Language (OpenCL). CUDA is an Application Programming Interfaces (APIs) extension to the C programming language which is developed specifically for NVIDIA GPUs. OpenCL is a framework for writing programs that execute across heterogeneous platforms such as CPU, GPU, DSPs, and FPGAs. However, CUDA and OpenCL allow specified functions from a C program to run on the GPU's stream processors. This makes C programs to be executed while taking the advantages of GPUs to operate on large matrices in parallel, while still making use of the CPU when appropriate.

OpenCL includes a language based on C programming language for writing functions that execute on OpenCL devices. It also includes Application Programming Interfaces (APIs) that are used to define and control the platforms. Moreover, it provides parallel computing using task-based and data-based parallelism. OpenCL has been adopted by many companies such as Apple, Qualcomm, Advanced Micro Devices (AMD), NVIDIA, Altera, and Samsung. It has many uses where it gives the ability to access the GPU for running programs and to automatically compile OpenCL programs into application-specific processors running on FPGAs. The following steps must be performed to use the OpenCL in different applications:

- **Set up environment:** Declare and create the OpenCL context and a command queue.

- **Declare buffers & move data:** Declare the needed buffers and the transfer input data to the device.

- **Run the program:** set the kernel arguments and the work group size and then enqueue kernel into the command queue to be executed on the device.

- **Get result to host:** after the program completes its execution, the result will return back from device buffer to the host memory.

Compute Unified Device Architecture platform (CUDA) [33-35] is a software development kit. It consists of a library that allows the programmer to develop programs for GPU utilization. It is developed based on the notion of kernel function which can be called simultaneously across many threads instances. Each function is identified to be executed either using CPU or GPU. The programmer must allocate the amount of memory storage area for each variable in a GPU function. The threads in the kernels differentiate themselves and work on separate parts of a data set since the kernels have special thread identification variables. The execution of any algorithm on GPU consists of four basic steps as shown in Figure 2-5: memory storage allocation on the GPU; copies the data from the CPU to the GPU, specifies a routine that executes on the GPU processing elements, finally, copies the data back to the CPU.

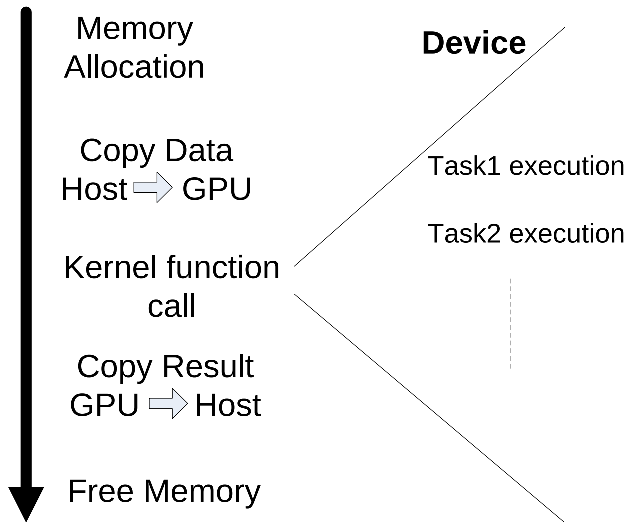


Figure 2-5: Basic steps of the algorithm execution on GPU.

2.3 Parallel Processing on FPGA Platform

FPGA is an integrated circuit designed to be configured by the designer where the configuration is specified using a Hardware Description Language (HDL). It contains large resources of logic gates and RAM blocks to implement complex digital computations. It supports very fast I/O and bidirectional data buses. The functions that are implemented on the FPGA can be re-configured with low non-recurring costs as FPGA contains reconfigurable interconnects that allow the logic blocks to be wired together. This makes FPGA platforms suitable for high performance applications.

Field Programmability of FPGA refers to the ability to change the operation at any time. This makes FPGA very interesting for hardware implementation. Designers can reprogram it after it’s manufactured rather than limited to unchangeable hardware function. FPGAs may be different from each other since they differ in their size and internal architecture. However, all FPGAs contain the following basic logic blocks:

- Logic elements.
- Lookup table.
- Memory resources.
- Routing resources.
- Clock.
- Configurable I/O.

Nowadays, FPGA becomes more complex where these basic logic blocks are combined with arithmetic and control structures such as multipliers, microcontrollers and others. Also, some of them contain specialized logic blocks that provide programmable input and output capabilities. FPGAs may differ with each other in terms of following elements:

- Logic size
- Logic structure
- Speed
- Memory size

- Power consumption

Recent research interests in a complete system on a programmable chip where the logic blocks and interconnects of FPGA are combined with embedded microprocessors. This design is accomplished by Xilinx and Altera. Xilinx Inc. integrates ARM and other processors into FPGA device which enables system architects and embedded software developers to use a combination of serial and parallel processing in their system designs. This integration helps to reduce the power consumption with higher reliability since most failures in modern electronics occur on Printed Circuit Boards (PCBs) in the connections between chips.

There are many techniques used to implement the DSP algorithms in the past such as Application Specific Integrated Circuit (ASIC). However, the ASIC has many drawbacks including high cost, low flexibility, long time to handle the mapping, routing, placement, and timing. However, FPGA based hardware implementation can effectively bridge the gap between software programmable systems and application-specific platforms based on custom hardware functions. Advances in FPGA lead to implementation of rapid hardware-accelerated algorithm. It eliminates the complex and time-consuming process of placing, routing, and timing analysis beside its low cost. FPGA are generally slower and consume more power than the same applications implemented in custom ASICs. However, the lowered risk and cost of development of FPGAs have made them good alternatives to custom ICs.

FPGA based systems are reconfigurable and provide option of rapid prototyping. It includes a large hardware area, memory resources and multiplier blocks surrounded by a programmable routing fabric that allows blocks to be programmable interconnected as shown in Figure 2-6. It also has programmable input/output blocks to connect the chip to the outside world. A large enough collection of gates can be used to implement any digital circuit including I/O, communication bus interfaces, and even entire microprocessors.

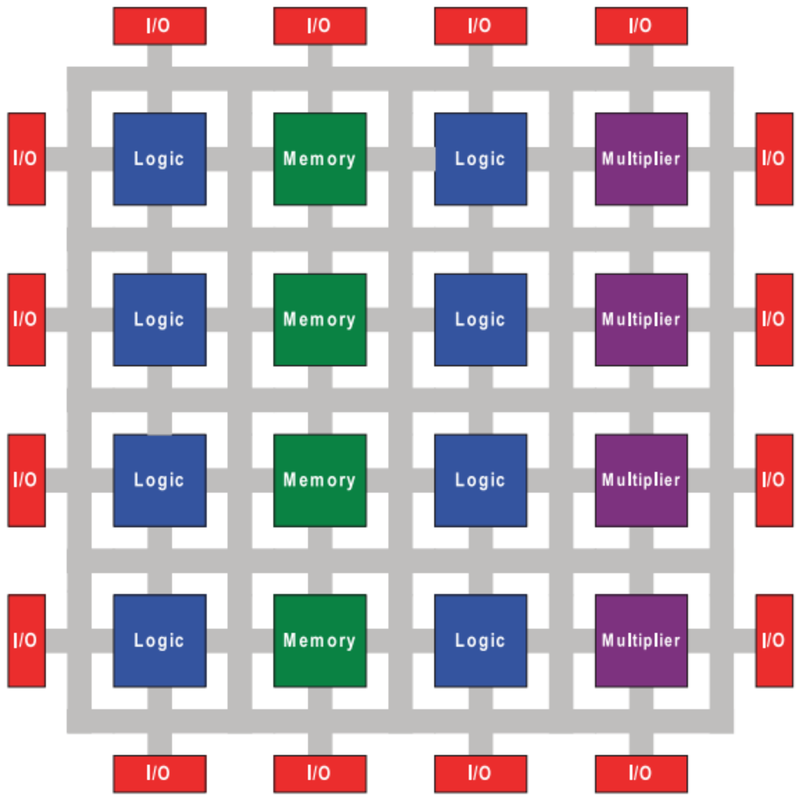


Figure 2-6: Conceptual FPGA architecture [36]

FPGAs use dedicated hardware for processing logic and do not have an operating system. Different operations do not have to compete for the same processing resources as the processing paths are parallel. This means speeds can be very fast, and multiple control loops can run on a single FPGA device at different rates. Also, the re-configurability of

FPGAs can provide designers with almost limitless flexibility. FPGA-based systems can literally rewire their internal circuitry to allow configuration. The implementation using FPGAs not only include a larger hardware area, but also embedded processors and memory resources. This option offers versatility in running diverse software applications on embedded processors, while taking advantage of reconfigurable hardware resources, all on the same chip package. Therefore, a hardware chip implementation is more desirable than software based implementation. The design flow process of FPGA for any application follows five steps as shown in Figure 2-7.

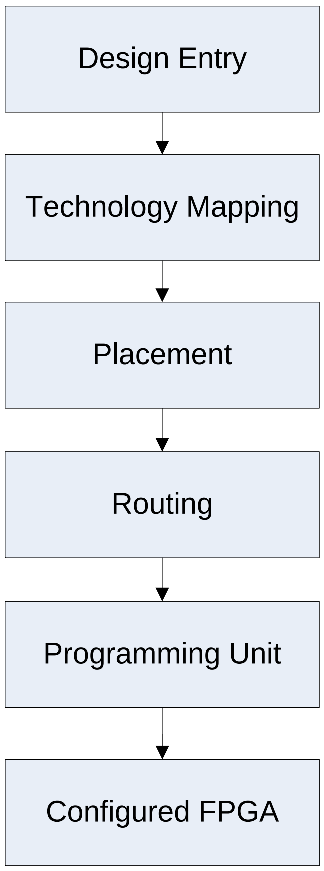


Figure 2-7: Design flow process diagram

FPGA architectures can be classified into two categories:

- **Fuse-based FPGAs**: it's called One Time Programmable (OTP) as it can't be modified once it's programmed.

- **SRAM technology-based FPGAs**: it supports design changes and updates throughout the cycle of a product until it's delivered to the customer.

In order to achieve high performance and meet real time requirements for complex signal processing applications, FPGAs can be interconnected to each other as shown in Figure 2-8 to design a complete Multi-FPGA system [37-38]. This may be necessary as some applications need very high computational and storage resources.

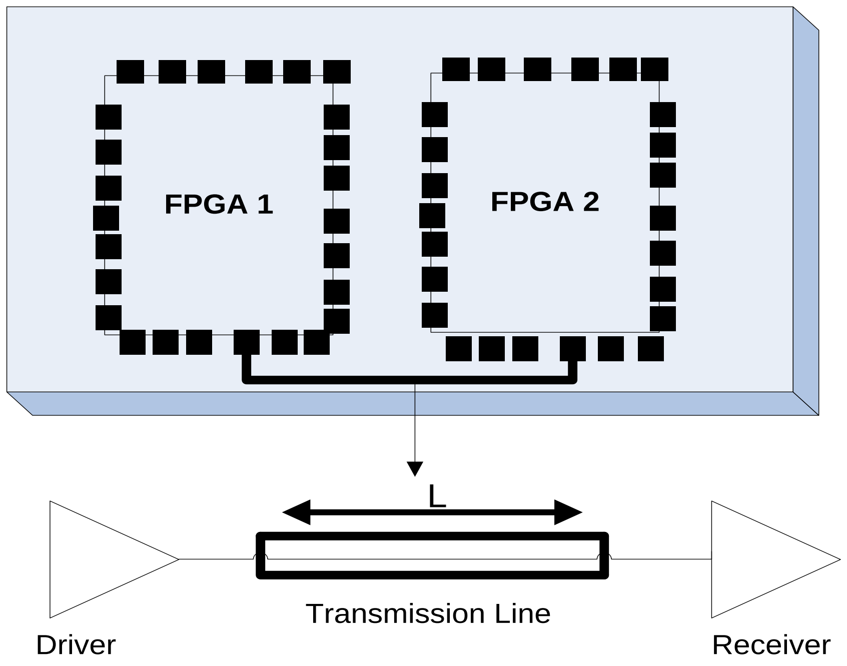


Figure 2-8: Multiple FPGA interconnection model.

VHDL and Verilog are two popular methods of Hardware Description Languages (HDLs) for FPGA programming. They are powerful but require high levels of expertise to program. One of the major problems with reconfigurable computing platforms is the required design time to get the desired efficiency and performance. Increasingly sophisticated tools are enabling embedded control system designers to more quickly create and more easily adapt FPGA-based applications. The design time and the VHDL programing difficulty can be reduced by the acceptance of High Level Synthesis (HLS) [39], a compiler for reconfigurable computing platforms.

Many steps must be considered in the design process of FPGA to achieve the desired performance such as:

- **Architectural design:** this can be accomplished by writing a pseudo-C code of the application, then translates it to Verilog or VHDL.
- **Compiling:** this process transfers the data into registers by compiling the VHDL or Verilog into Register Transfer Logic (RTL) netlists.
- **Synthesis:** this step is required to produce bits to control gates and fill registers and memories on an FPGA. This level is called gate-level logic as it describes the needed logical gates of the application.
- **Placing and Routing:** this step places the synthesized subsystems into FPGA locations and makes necessary connections (FPGA routes) between these subsystems.

- **Loading:** Finally, after FPGA programs have been compiled, synthesized, placed, and routed, it must be embedded in the physical FPGAs. So, the programmer downloads the programming file bits into the FPGA hardware to implement the gates of the system.

- **Debugging programs**: this step uses debuggers, simulators, and emulators in FPGA for verification purposes. These tools enable us to go through the program execution and observe the effects on flags, register contents, and memory locations.

2.4 High-Level Synthesis Tools

FPGA technology is an interesting choice to achieve high performance and meet real time constraints. However, implementing complex applications using a low-level HDL is a discouraging task for regular DSP or microprocessor user. For this reason, high-level synthesis tools (HLSTs) [40-43] arise as an alternative to HDLs when using FPGAs.

HLSTs have been evolved since the 1980s. Based on [40], three generations of HLSTs can be distinguished. The first one was developed during the 1980s and was mainly a research-oriented one with no or little impact in industry [44]. The lack of a real need, obscure input languages, and a problematic performance limited the adoption of these tools [45]. The second generation started in mid 1990s and was fostered by the major Computer Aided Design (CAD) companies present in the market, i.e. Cadence, Synopsys, and Mentor Graphics [45]. However, the commercial tools from these companies lack of good performance and reduce learning curve, making it not interesting for current designers [45]. Finally, the third generation of HLSTs started at the beginning of 2000s. This generation of tools currently offers improved performance and user interface leading to a significant reduction in the design times and in the learning process [45]. Also, most of them have adopted high level languages such as Matlab or C. There are many HLSTs developed during 2000s such as Vivado HLS from Xilinx, Synphony C Compiler from Synopsys, Catapult from Mentor Graphics, and CtoS from Cadence.

Raising the abstraction level from low level languages to high level languages such as C has enabled software engineers to develop more complex applications with improving productivity. This makes the design process easier. High level synthesis tools also known as Electronic Design Automation (EDA) tools try to bridge the hardware/software gap by supporting automatic transformations from high level programming models to RTL hardware descriptions. This is performed by redirecting the time consuming HDL work to the compiler instead of the programmer.

HLSTs simplify and accelerate the design process for complex algorithms and ease the migration of some designs from DSP to FPGAs [46]. There are many DSP design tools such as DSP Builder and System Generator from Altera and Xilinx, respectively, which enable the use of Simulink for FPGA design [47]. However, both of them need the implementation of the control logic that controls the scheduling of the operations which is one of the most time-consuming tasks, and they are limited to the available library elements.

HLSTs support some directives and constraints defined by the user to optimize the design according to performance and/or area criteria. This allows obtaining and comparing different implementations in a few minutes. As a consequence, the design space exploration is significantly simplified and further optimization can be achieved. Moreover, the C-based input can include a test bench to verify the output RTL, therefore improving designer productivity by removing the need to create RTL test benches for RTL verification. The HLST also creates the required scripts to verify the generated RTL through co-simulation with the original test bench and a variety of RTL simulators.

Many HLS tools are developed to raise the abstraction level for designing digital logic. A full comparison between twelve tools regarding their capabilities and limitations are performed in the following paragraphs [39]. The comparison will be performed based on many metrics such as:

- Source language and ease of implementation regarding the abstraction level.
- Tool complexity regarding the documentation and the user interface.
- Support for data types.
- Design exploration capabilities.
- Verification and correctness.
- Generated design regarding the size, latency, and resource usage.

1. Xilinx AccelDSP

AccelDSP is a high level synthesis tool from Xilinx. It provides the designer the following capabilities:

- Transforms Matlab floating-point data to hardware description language that can be implemented in Xilinx FPGA.
- Explores design trade-offs of an algorithm for target FPGA architectures.
- Creates a synthesizable RTL HDL model.
- Creates an HDL test bench for verification purposes.
- Provides automatic conversion from floating-point to fixed-point.
- Provides automatic calling an HDL simulator to run test bench.
- Provides automatic calling of ISE tool to place and route the design.
- A GUI that eases the use of integrated environment with Matlab and Xilinx ISE.

However, AccelDSP has many limitations such as:

- Works only on streaming functions for image and signal processing applications which reduces the domain applications.
- A limited part of Matlab code is supported.

- Based on [39], AccelDSP is a powerful tool for only streaming applications and more directives are still needed.

2. Agility compiler

Agility compiler is a tool of electronic system level for SystemC which was acquired by Mentor Graphics. It provides the designer the following capabilities:

- Automatically generates register transfer language from SystemC code.
- Explores complex algorithms and architectures early in the design.
- Automatically generates a code which is supported for Actel, Altera and Xilinx FPGAs.
- The generated code is performed by separating the details of the communication from the implementation modules.
- It contains black boxes where HDL IP can be imported.
- Automatic generations of control and data flow graphs are supported.

However, Agility compiler has many limitations such as:

- It requires writing of certain hardware processes manually.
- A test bench must be implemented manually for verification purposes.

3. Vivado HLS (AutoPiolt)

Vivado HLS tool is acquired by Xilinx where it compiles HDL from a variety of C-based input languages such as ANSI C, C++ and SystemC. It automatically generates the RTL description of FPGA for a given application using a high-level language. It offers many advantages:

- Eliminates use of low-level HDL which is time consuming and prone to errors.
- Provides various directives and constraints that will be useful for an optimized design.
- Simplifies and accelerates the design process and eases the migration of some designs from DSP to FPGAs.
- Allows development of optimized hardware-software co-designs and allows easy comparison of different hardware implementations of the same algorithm in terms of performance, resources, and estimated power.
- Increases the productivity, improves the reliability, and enables design exploration to achieve the most efficient implementation.
- Includes a test bench to verify the output RTL through co-simulation with a variety of RTL simulators.
- Automatically generates the data-path and the control unit.
- Provides a report about clock period, latency, resource estimation, and power dissipation.

- Allows reuse of c-code test bench for RTL simulation reducing the verification time.

Although HLST gives the user a set of constructs through C/C++ and SystemC to design a specific application. However, it has many restrictions and limitations:

- Dynamic memory allocation is not supported and a fully specified of all needed resources is required.

- It has limitations during synthesis when using pointers as well as the size of the structure pointed to must be known.

4. BlueSpec

BlueSpec is a tool that was developed by BlueSpec Inc. It is Verilog based language where rules called Guarded Atomic Actions (GAA) must be used in order to implement the design modules. It offers many advantages:

- Handles the scheduling and dependencies implicitly.

- Pragmas can be performed for rule scheduling.

- Supports a GUI and command line interface for compilation purposes.

- Provides clear readability and traceability of the generated code since it preserves the signal names in the design.

However, BlueSpec has many restrictions and limitations:

- Requires re-implementation of the algorithms where no pre-existing code can be used.
- Needs long port maps and connection lists since all its modules implemented separately.
- Has complex code for basic operation since it uses server-client interfacing approach.
- Difficulty in learning since it uses uncommon rules programming paradigm.
- Doesn’t support any design exploration capabilities.
- Needs to write the testbench manually.

5. Catapult C

Catapult C is a tool that was acquired by Calypto Design Systems. It provides a large subset of C-based languages. It offers many advantages:

- It’s easy to use as it has a powerful manual and a very clear guide through HLS process.
- Supports options to select target technology, clock rate, area and latency constraints, interfaces etc.
- Supports a GUI and command line interface for compilation purposes.
- Optimization techniques are supported to enhance the design.

- Includes a test bench to verify the output RTL through Modelsim simulator.
- Provides arbitrary bit width data types.
- Provides an overview of the performance in terms of area, latency, and throughput in the form of tables and plots.

However, Catapult C has many restrictions and limitations:

- Memory accesses are not optimized.
- A modification of the source code must be performed to enable local buffering for memory purposes.

6. Compaan

Compaan is a tool developed by Compaan Design that uses Eclipse Software Development Kit (SDK) tool. It’s not considered as high level synthesis since it doesn’t generate the processing elements of the application. This tool is mainly designed for streaming application for heterogeneous multi-core platforms. It has following advantages:

- Supports some pragmas for auto-parallelization purposes.
- Generates the entire communication infrastructure for an application.
- A FIFO structure is supported for data exchange between different nodes of multi-core processors.

Compaan has many disadvantages and restriction such as:

- Doesn’t support a wide range of applications since it is designed mainly for streaming applications.
- The mapping process on the resources must be performed manually by the designer.
- A modification of the source code must be performed by the designer to change the number of generated nodes for design exploration.
- Writing of functional IP using external tools must be performed by the designer.
- Supports only uni-directional FIFO interface.

7. C-to-Silicon (CtoS)

CtoS is a tool developed by Cadence where it uses SystemC as a design language. It accepts several programming languages such as C, C++, and TLM 1.0. Offering many advantages:

- Supports GUI for compilation and implementation purposes.
- Provides arbitrary bit width and fixed-point data types.
- Supports optimization techniques for both memory mapping and loop optimization.

- Generates RTL models for verification purposes.

CtoS has many disadvantages and restriction such as:

- Developed mainly for ASIC design not for FPGA design.
- Doesn’t support powerful materials and tutorials leaving many questions unanswered.
- Complicated GUI.
- Difficult to learn since defaults are missing.
- Difficulty in understanding of some options and constraints for some optimizations.
- The verification process must be performed only through Cadence simulators.

8. CyberWorkBench (CWB)

CWB is a high level synthesis tool based on ANSI C. It accepts a variety of input languages such as C, SystemC and Behavioral Description Language (BDL). This tool provides many advantages:

- Supports both GUI and command line for parsing and compilation purposes.
- Supports an interface for both function parameters and global variables.
- Arbitrary bit width and integer data types are supported.

- Supports both automatic and manual scheduling.
- A number of optimization techniques such as unrolling and folding are supported through GUI.
- Pragmas can be set in the source code for certain loops or variables.
- Provides an option where the designer can restrict the number of resources such as multipliers and adders to balance the design between area and latency.
- Generates various performance diagrams in terms of area/latency for many combinations of settings to obtain the optimal design.
- Automatically generates test bench with many simulation scenarios for verification purposes.
- Provides features where some external simulators can be called from CWB.
- The resulting output is written in both VHDL and Verilog languages as well as generating scripts for RTL synthesis.

CyberWorkBench has many disadvantages and restriction such as:

- Some constructs and functions are non-synthesizable where the designer must remove it manually.
- The input data need to be converted by the user into a separate file for each port.

- There are some difficulties in understanding the numerous tool options.

9. DK Design Suite

DK Design Suite is a high level synthesis tool developed by Mentor Graphics in 2009. It is appeared before Catapult C tool. This tool accepts Handel-C language as input language which is a subset of C language. It offers many advantages:

- Supports some extensions of interfaces, arbitrary precision data types, and parallelism.
- Supports a straightforward GUI for compilation purposes in addition to a command line interface.
- An excellent Handel-C manual through online help system is supported.
- Generates the RTL output in both VHDL and Verilog languages.

DK Design Suite has many drawbacks and restriction such as:

- Causes difficulties of the designer to explicitly specify parallelism for generating efficient code from the Handel-C languages.
- DK is mainly designed for FPGA which impede the designer to port it on different platforms.
- The designer must perform some modifications on the source code such as adding macros and explicitly defining the parallelism for design space exploration purposes.

- The loops nests must be optimized manually by the designer where the correctness of such optimizations must be validated through simulations.

- Validation of the generated RTL is difficult since DK has limitations to expose the data ports on the top level.

10. Impulse CoDeveloper

Impulse CoDeveloper is a high level synthesis tool developed by Impulse Accelerated Technologies (IAT). It comes with its own IDE where it generates RTL design from C languages. This tool offers many advantages:

- Supports a wide range of FPGA platforms such as Altera, Nallatech, Pico, and Xilinx.

- Supports many tutorials to get started and learn its features easily.

- The generated RTL can be combined with IP to build a complete system.

- Provides processor acceleration as the generated hardware accelerators can be connected to an embedded FPGA processor.

- Supports many ways for communication between processes such as streams, registers, and signals where the designer can adjust the depth of the streams.

- Supports several libraries and bus interfaces to create interface between the FPGA and the embedded processor.

- Provides various ways for design optimization to generate an efficient design.

- Provides various simulations, verification, and debugging tools such as stage master debugger and ModelSim simulator.

Impulse CoDeveloper has many drawbacks and restrictions such as:

- Some modifications must be performed by the designer since ImpulseC is based on ANSI C language with some extensions.
- Doesn't support any IP to the designer through implementation.
- The generated HDL code files contain large number of lines making difficulties in reading and modifications.

11. ROCCC

Riverside Optimizing Compiler for Configurable Computing (ROCCC) is a high level synthesis tool from Jacquard Computing Inc. It can be run under eclipse environment. It offers some advantages such as:

- Supports arbitrary bit width data type.
- Allows the designer to select some options and constraints to generate powerful design.
- Supports smart buffers where the fetched data from memory can be re-used in a subsequent iteration if they have the same data elements.
- Supports a powerful design for streaming and sliding window applications.

ROCCC has many drawbacks and restriction such as:

- Accepts a very restrictive subset of C language which restricts the implementation of the desired design.
- Doesn’t support overflow checking such as out of bound memory size.
- The generated RTL code contains huge file of VHDL making difficulties in modifications and reading.
- The transfer rate is limited to a single word in every two clock cycles.
- Difficulty in verifying RTL output since the memory interface contains both synchronous and asynchronous behaviors concurrently.

12. Synphony C Compiler

Synphony C Compiler is a high level synthesis tool from Synopsys which is acquired from Synfora in 2010. It accepts ANSI C and C++ as input language. This tool offers some advantages such as:

- Supports many optimization techniques which can be inserted into the source code through pragma feature.
- Supports an excellent manual which illustrate how to write a code for optimal design.
- Provides options to include clock rate and some constraints on the input file to generate the desired design for a certain application.

- Provides a GUI and a script for compilation purposes.
- Provides streaming interfaces.
- Automatically generates the test bench for verification purposes.
- Various HDL simulators can be called from the tool.

Synphony C compiler has many drawbacks and restrictions such as:

- The generated output is huge in terms of area leading to a bad design where corrected settings must be properly assigned.

A summary of the main characteristics and comparisons between the tools is shown in Figure 2-9. The comparison is performed in terms of different criteria such as:

- The ease of implementation.
- Abstraction level.
- Learning curve.
- Documentation.
- Data types supporting.
- Exploration.
- Verification.

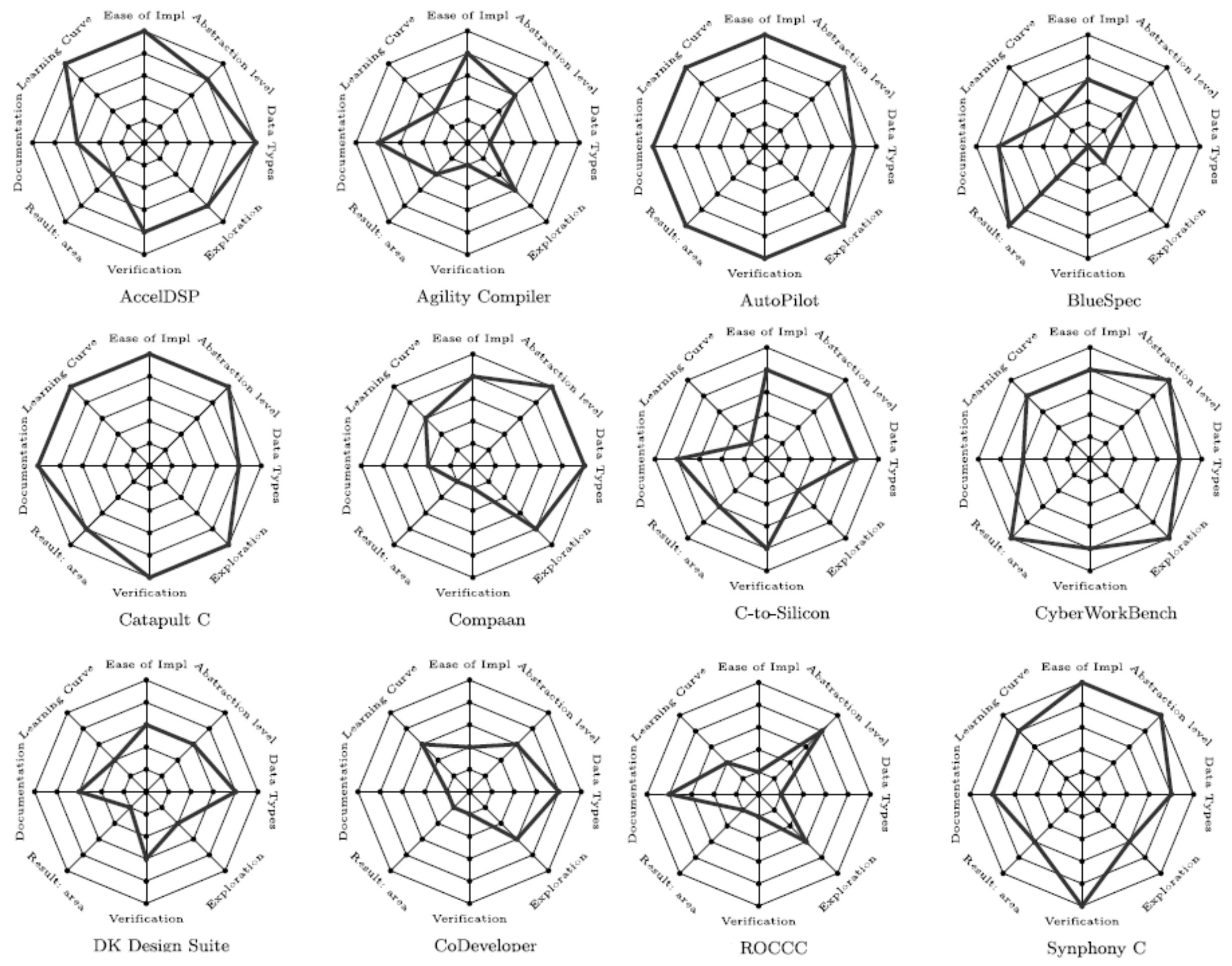


Figure 2-9: Tools comparisons [39].

These HLS tools have been developed to move the design effort to higher abstraction levels. This reduces the design time for FPGA based hardware implementation and speed up the verification process. Also, it improves the performance and the design exploration by adopting many optimization techniques. Based on the evaluation in Fig. 2-9, Vivado HLS tool is chosen since it offers the best performance based on their metrics evaluation. Moreover Xilinx provided Vivado tool under their university support program. Therefore, it made easier for us to use it in this work. However, all tools lack exploitation of any available opportunities of data locality and reduction of memory bandwidth requirements.

Also, in all HLS tools [39], even the best ones, exploration and optimization options are still application specific. It is the responsibility of the designer.

Figure 2-10 shows the Vivado HLST-based implementation procedure. Vivado HLS allows the user to automatically generate the hardware description code such as Verilog and VHDL from high level languages such as C, C++, and SystemC. This will be highly beneficial since the low level language needs strong knowledge of hardware, requires long time for coding and validation, and is prone to errors. The generated VHDL code from HLS can be easily converted to bitstream files using Xilinx ISE. Moreover, HLS provides many optimization options which can help in producing an optimized design quickly and efficiently that meets the desired performance.

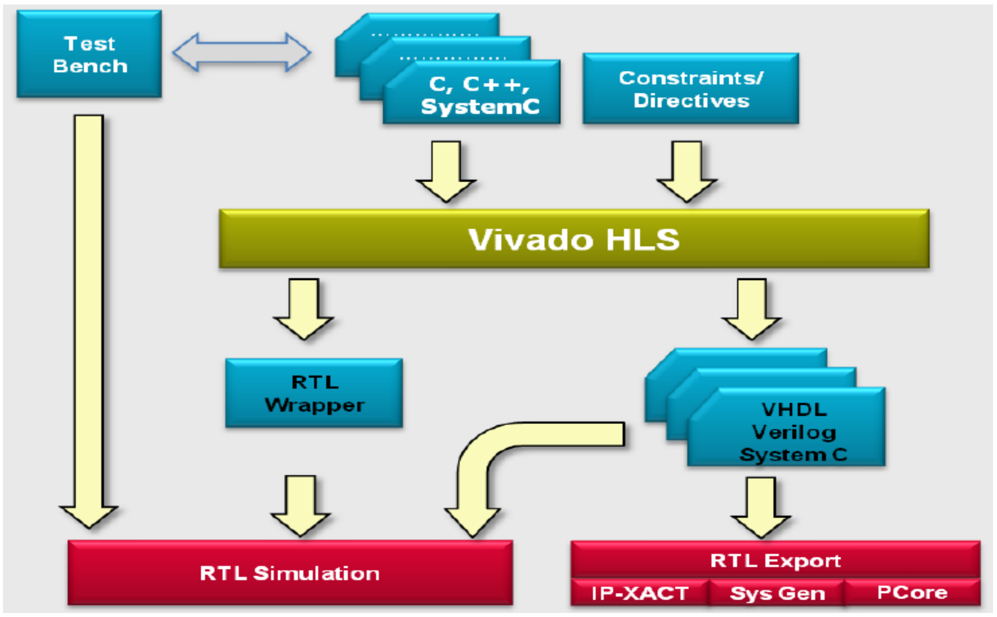
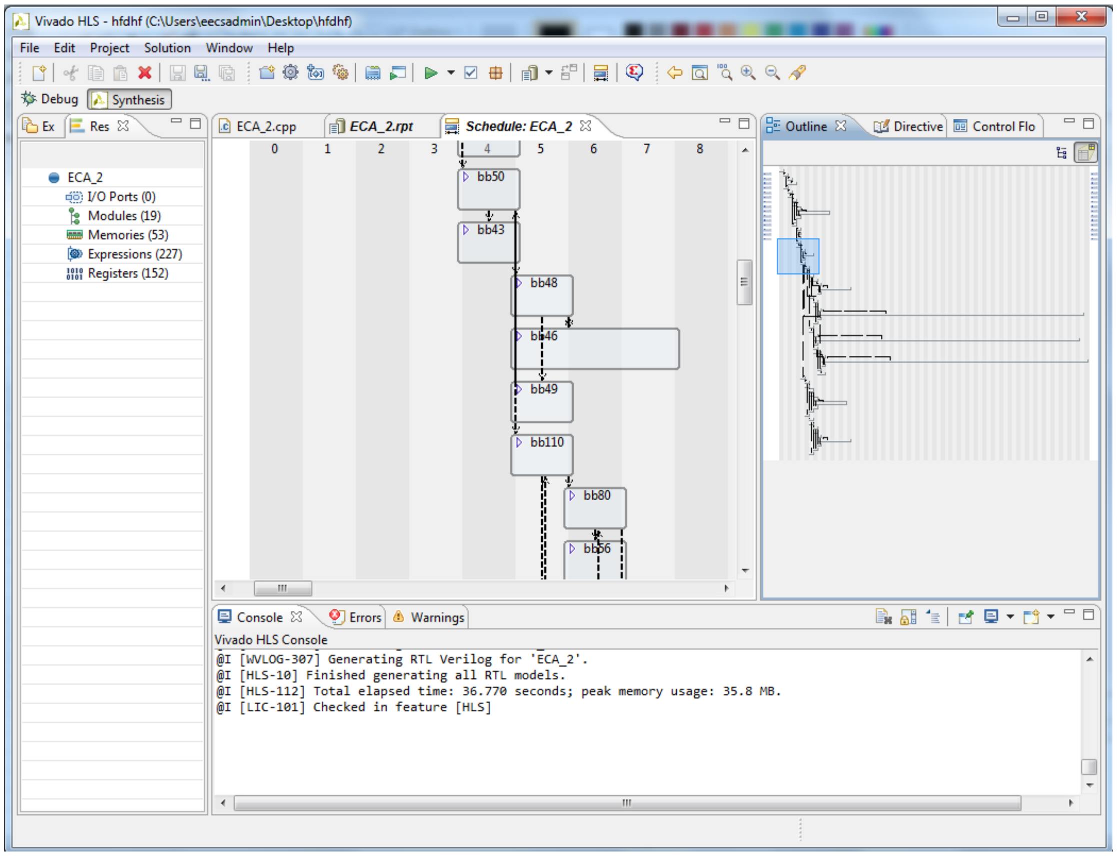


Figure 2-10: HLST-based implementation procedure [48].

Vivado HLS also allows the user to implement the same algorithm in different ways by applying different directives, constraints, and offers quick comparisons in terms of performance, hardware resources, and estimated power consumption. It offers many advantages and flexibility to the designer such as:

- Simplifies the design and simulation tasks.
- Reduces the design time.
- Increases the productivity.
- Improves the reliability.
- Enables exploration of the design space.
- Includes a test bench for verification of generated RTL.
- Generates the data-path and the control unit for micro-scheduling operations.
- Supports arbitrary bit-width operations.
- Supports automatic array that can be partitioned, mapped, streamed, or reshaped to increase the bandwidth.

Vivado HLS also informs the designer about the limiting factors in the design such as multiple BRAM accesses which needs to be spread out across multiple clock cycles due to limited resources. Figure 2-11 shows the scheduling graph for a given algorithm as it is shown in the Vivado HLS GUI. It shows the design of the control unit of a deep pipelined data-path which is a complex problem.



```
@I [WVLOG-307] Generating RTL Verilog for 'ECA_2'.
@I [HLS-10] Finished generating all RTL models.
@I [HLS-112] Total elapsed time: 36.770 seconds; peak memory usage: 35.8 MB.
@I [LIC-101] Checked in feature [HLS]
```

Figure 2-11: Vivado HLS graphical interface (scheduling graph).

Vivado HLS gives the user a set of constructs through C/C++ and SystemC for the design of a specific application. However, it has many limitations and results in poor performance. Vivado HLS does not support dynamic memory allocations, recursion, and has some restrictions on pointers. The program needs to be modified to express arbitrary bit-width operations as they are not supported in C/C++ specifications. The support for floating point is also dependent on the HLS tool and the target technology.

Moreover, Vivado HLS does not support certain trigonometric and other mathematical functions and requires the development of customized functions. It is not able to extract all the possible parallelism from the sequential program and requires explicit expression of parallelism. However, the coding style of the input program can drastically influence the end result of the generated design. The efficiency of the design is also affected by packing the data elements into wider vectors or distributing them across multiple storage elements, streaming/interface support, and the use of arbitrary precision types. So, the designer still needs to be aware of the underlying hardware and use the proper coding style to arrive at an optimized architecture.

2.5 Conclusion

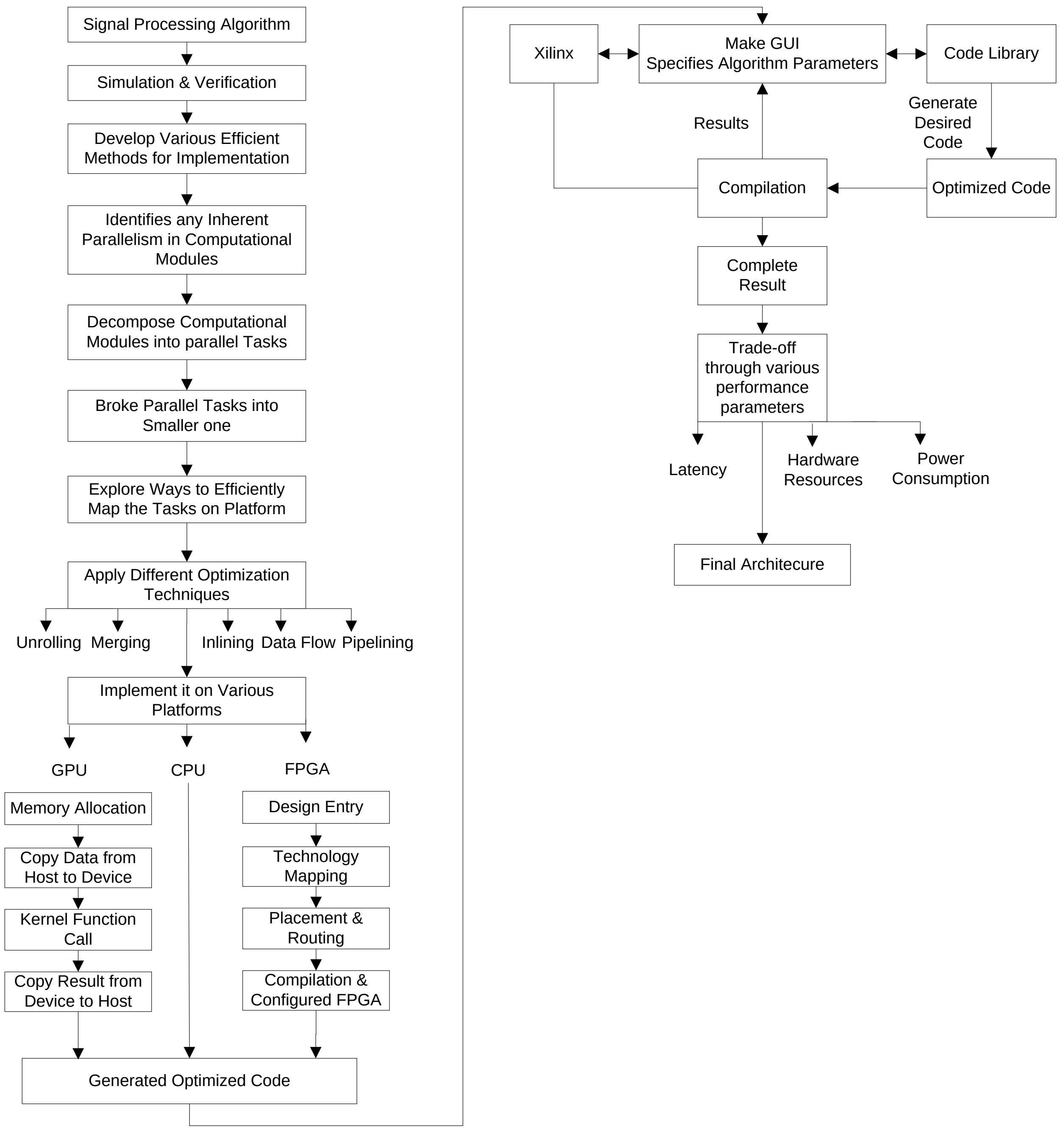
An overview and background on parallel processing platforms and high level synthesis tools are discussed. Parallel processing platforms such as multi-core, FPGA, and GPU can run a complex algorithm in a tiny time as they provide hundreds of cores and resources. Xilinx Vivado HLS tool can significantly reduce the design time of FPGA-based hardware by avoiding VHDL code. It enables design exploration, and, therefore, should be considered as a new paradigm in FPGA design for complex applications.

Chapter 3

Parallelization of Signal Processing Algorithms

Signal processing algorithms are computationally and data intensive. These algorithms will benefit with the availability of parallel processing platforms to meet their real-time requirements. Some of the signals processing algorithms that are being investigated in this thesis are IR video processing, Direct Data Domain (D^3), Extensive Cancellation Algorithm (ECA), Block Compressive Sampling Matching Pursuit algorithm (BCoSaMP), Discrete Wavelet Transform (DWT), Particle Filter (PF), and Iterative Hard Thresholding (IHT). They will be first transformed into highly parallel algorithms. Then, they will be mapped and scheduled on different platforms such as CPUs, multi-cores, MPSoCs, GPUs, and FPGAs. Following sections will describe parallelization, mapping and scheduling of these algorithms on various platforms.

A generalized approach for parallelizing a target algorithm has been adopted as shown in Figure 3-1. It is accomplished by creating a methodology for various processes such as evaluation of data dependencies, exploring parallel processing opportunities, and improving the data locality. So, this exploits any opportunity of optimization to minimize computation time, storage resources, area, power dissipation, and cost.



Signal Processing Algorithm
Simulation & Verification
Develop Various Efficient Methods for Implementation
Identifies any Inherent Parallelism in Computational Modules
Decompose Computational Modules into parallel Tasks
Broke Parallel Tasks into Smaller one
Explore Ways to Efficiently Map the Tasks on Platform
Apply Different Optimization Techniques
Unrolling
Merging
Inlining
Data Flow
Pipelining
Implement it on Various Platforms
GPU
CPU
FPGA
Memory Allocation
Copy Data from Host to Device
Kernel Function Call
Copy Result from Device to Host
Design Entry
Technology Mapping
Placement & Routing
Compilation & Configured FPGA
Generated Optimized Code
Xilinx
Make GUI Specifies Algorithm Parameters
Code Library
Results
Generate Desired Code
Compilation
Optimized Code
Complete Result
Trade-off through various performance parameters
Latency
Hardware Resources
Power Consumption
Final Architecure

Figure 3-1: Generalized parallel approach for signal processing applications.

3.1 Parallelization of IR Video Processing on a GPU

Video processing is applied in numerous processing applications such as security for shopping malls, detection for military targets to detect suspicious activity, and monitoring for birds and bats activity to minimize impact of wind turbines on birds and bats [49-51]. The IR camera provides excellent night visibility and situational awareness, even in absolute darkness. The IR data processing is different from the normal video processing techniques as it has some unique features that can be exploited. IR camera usually captures video with less background information and highlights objects with higher temperature. This helps in subtracting objects from the simple background model. IR camera is less sensitive to the changes in illumination and makes its background model more stable. However, the IR camera data are only grayscale images (although they are recorded in RGB format); the color information is not very useful for identifying different objects in details.

There are three essential parts for IR video processing: (1) background subtraction (2) noise filtering and (3) connected component labeling. Background subtraction segments out image pixels that correspond to moving objects. A statistical background model [52] is used which takes the mean of the previous n-frames as shown in Equation (3.1) and subtracted from the current frame. The background model is updated every n-frames. At the beginning of the video (before the 5th frame), it takes the constant background as model. The updating starts from the 6th frame to the end of the video. It is useful when number of targets is not very large. A background model is assembled by observing the temporal history of each pixel.

$$B(x, y, t) = \frac{1}{n}\sum_{i=0}^{n-1} I(x, y, t-i). \quad (3.1)$$

Where, $B(x, y, t)$ is the background model, $I(x, y, t-i)$ is the image for pixel (x, y).

Noise is the unnecessary information of the data/source signal that needs to be removed. The quantity of the noise is a main factor that affects the quality of the outputs. By applying morphological operations such as erosion, dilation, closing, and opening, most salt and pepper noise can be removed and the object can be reshaped. Noise removal is also helpful for the connected component labeling. Connected component labeling in IR video processing is used to isolate, measure and identify potential object regions in an image by assigning a unique number to all foreground pixels that belong to the same connected component of the image [53]. As a result of the labeling, individual components can be extracted from the image for further analysis [54-55].

3.1.1 Parallelization of Each Step of Video Processing Method

In order to effectively take advantage of GPU processing, we must extract all the parallel tasks and exploit all the resources on a GPU. So, fine-grained parallelism within each video frame must be exploited. IR video processing algorithm will have following structure.

```
/* Main program*/
    -   Background subtraction
    -   Noise filtering
    -   Connected component labeling
    -   End
```

Following sections describe parallelization of three essential phases of video processing.

Parallelization of Background Subtraction

The objective of the background subtraction is to extract pixels corresponding to moving objects. In order to estimate the background subtraction, we divide the algorithm into three parts: 1) Create the background model which estimates the mean of the previous n-frames to serve as the background for the next n-frames. 2) Background model is subtracted pixel by pixel from the current frame. 3) Image thresholding is performed. Mapping of all three parts of the background subtraction algorithm is performed in the following fashion.

An m*m image will have m^2 pixels. Operations can be performed in parallel on each pixel giving us very fine grain parallelism. This work utilizes IR images of size 704*480. Each column is then mapped on thread block (512 threads/block, 8 blocks/core). Each thread calculates the mean of the corresponding pixel by using pixels from n-frames. If there are n-images in a video then each core will receive ten images (five images for background model estimation and other five images for background subtraction).

All created threads calculate in parallel the mean value of the pixels in the first column for the five frames and store the result as background model as shown in Figure 3-2. All threads subtract in parallel the value of each pixel in the column from the background model and store the result as shown in Figure 3-3. A pseudo code of the background subtraction is as follows:

```
Function Background Subtraction
For i ⟵ 0 to Num_Frames
   { F(i) ⟵ Read Frames;}
Declare host variables (CPU);
Declare integer k, k1;
Declare device variables (GPU);
  k1=0;
For k ⟵ 0 to (Num_Frames-5), K+=5
{If (k=0)
         {For i  ⟵ 0 to Rows
           {For j ⟵ 0 to Cols
             {F1=F(k);
                 F2=F(k+1);
                 F3=F(k+2);
                 F4=F(k+3);
                 F5=F(k+4);}}
Copy the frames F1, F2, F3, F4, F5 to the device}
Do in parallel on the device:
{Call BackgroundModel_Kernel (attributes)}
/*Execute on host following code */
For i ⟵ 0 to Rows
           {For j ⟵ 0 to Cols
              {F1=F(k1+5);
                 F2=F(k1+6);
                 F3=F(k1+7);
                 F4=F(k1+8);
                 F5=F(k1+9);}}
Copy the frames F1, F2, F3, F4, F5 to the device
Do in parallel on the device:
Call BackgroundSub_Kernel (attributes)
K1+=5;}

/* functions to be executed in parallel */
/* code for background model estimation */
BackgroundModel_Kernel
Declare X ⟵ index for the thread in x-dimension;
Declare Y ⟵ index for the thread in y-dimension;
Average= mean value of the five frames
BackgroundModel [Y*Rows+X] =Average;

/* code for background subtraction*/
BackgroundSub_Kernel
Declare X ⟵index for the thread in x-dimension;
Declare Y ⟵index for the thread in y-dimension;
BackgroundSub [Y*Rows+X] =the current five Frame – BackgroundModel
```

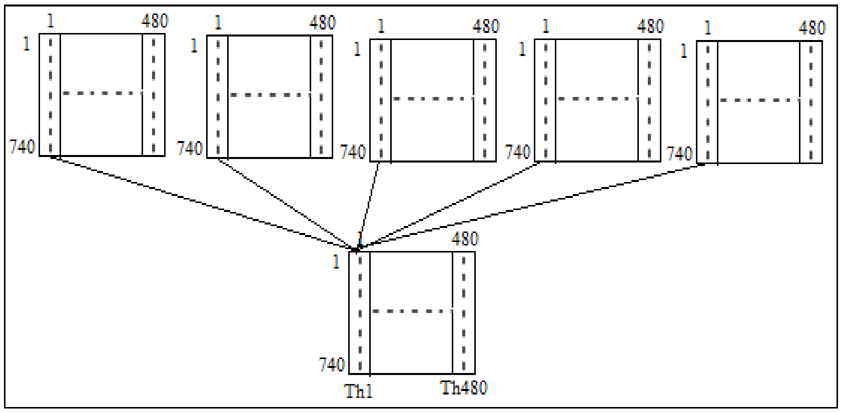


Figure 3-2: Parallelization of background model estimation

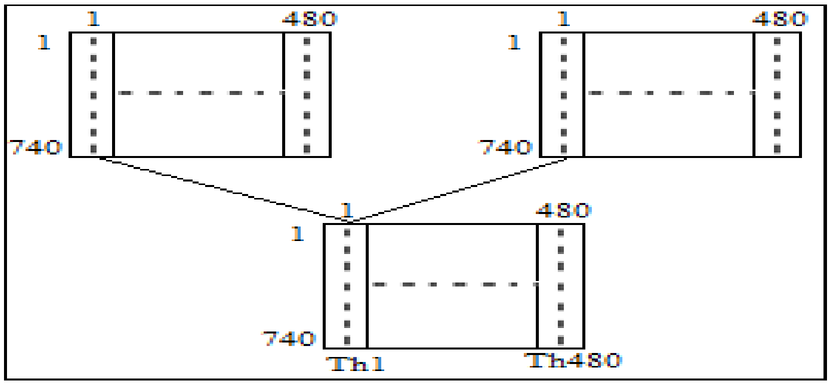


Figure 3-3: Parallelization of background subtraction using previously created background model

<u>Parallelization of Noise Filtering</u>

Noise filtering is also mapped similar to the approach shown in Figure 3-2. It also uses column-wise parallelism. There is no inter thread dependencies and the operations on a pixel are independent of the operations on other pixels in that frame. Following is the parallel pseudo code for dilation and erosion algorithms.

```
  Function Dilation (GPU)
Declare X ← index for the thread in x-dimension;
Declare Y ← index for the thread in y-dimension;
{For (X, Y) in frame
        For i ← 0 to KerX {
        For j ← 0 to KerY
```

```
                {**If** (frame[x-i][y-j] =255 &&ker [i][i]=255)
                        {Value=255; break ;}
**Else** {Value= 0 ;}}}
Output_Dilation [X*Cols+Y] ⟵ Value}}
```

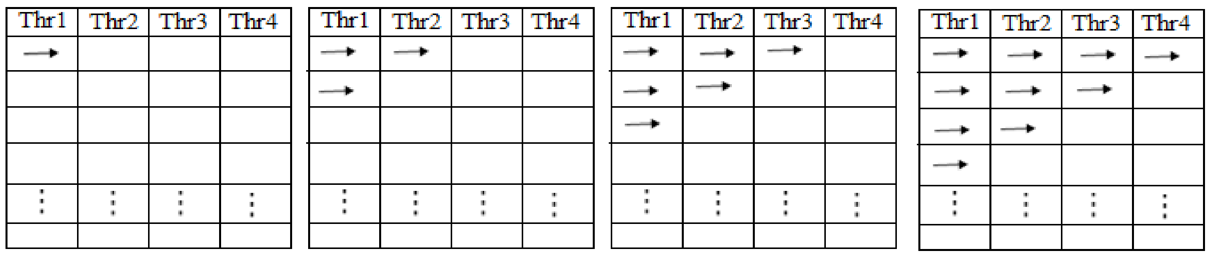
```
  **Function** Erosion (GPU)
**Declare** X ⟵ index for the thread in x-dimension;
**Declare** Y ⟵ index for the thread in y-dimension;
{**For** (X, Y) in frame
    **For** i ⟵ 0 to KerX {
    **For** j ⟵ 0 to KerY
       {**If** (frame[X-i][Y-j] =0 &&ker [i][i]=0)
          {Value=0; break ;}
        **Else** {value=255 ;}}}
Output_Erosion [X*Cols+Y] ⟵ Value;}
```

Parallelization of Connected Component Labeling

Connected component labelling involves scanning an image, pixel-by-pixel (from top to bottom and left to right) in order to identify connected pixel regions by moving along a row until it comes to a point p (pixel to be labelled). When this is foreground object then it examines the neighbours of p which have already been encountered in the scan. After completing the scan, the equivalent label pairs are sorted into equivalence classes and a unique label is assigned to each class. As a final step, a second scan is made through the image, during which each label is replaced by the label assigned to its equivalence classes. In this work, 8-connected neighbours are used for labelling objects in the image.

In order to perform labelling operation in parallel, image is processed row-wise. Each row is divided into four threads as shown in Figure 3-4. This process creates a data dependency as the second thread cannot start processing until first one is done. A synchronization scheme has been devised and uses a synchronization point in the form of a flag.

In order to show the method clearly, suppose there are four threads (Th1 to Th4) and the image in this case has 704 rows and 480 columns assuming an IR image of 704*480. So we divide the number of row pixels between four threads. First 120 pixels in the first row will be labelled by Th1 and the next 120 pixels by Th2. It can be seen that Th2 can't start labelling in the same row as Th1 until Th1 is completed its labelling. But, after Th1 completed labelling of the first 120 pixels in the first row it will start with the second row. Simultaneously Th2 can start labelling the second half of the first row. Threads will start executing in parallel.



(a) (b) (c) (d)

a) Th1 labeling its portion b) Th1 and Th2 are labeling concurrently c) Th1, Th2 and Thr3 are labeling concurrently d) The four threads labeling together.

Figure 3-4: Example of connected component labelling parallelization method

Following is the pseudo code of the connected component labeling.

```
Function Component Labeling (GPU)
Declare Offset = row/ number of threads;
For row in data  {
K= calculate the start index for this thread;
While (condition of this thread is 0; then wait)
   For K in row to K+Offset
     {Label the pixel;
```

```
 If (all the neighbors = 0), then a new label is assigned
Else if (only one neighbor =1), then assign its label to it;
Else if (more than one neighbor =1), then assigns one of the label to the pixel and
make a     note of the equivalence ;}
 Set the condition of the next thread to 1;}
```

3.2 Parallel Implementation for IR Video Processing on MPSoC Platform

Networks-on-chip has been seen as an interconnect solution for complex system but the performance and the energy dissipation still represent limiting factors for Multi-Processors Systems-on-Chip (MPSoC). The future handheld devices must support multimedia applications for long battery life but this type of application imposes heavy constraints in terms of energy and forces the designers to optimize all parts of the platform to achieve the desirable goals. So, extensive energy analysis for heterogeneous NoC platform and a comprehensive method are essential to know bottlenecks of the energy dissipation in the handheld devices while running a video application in real time.

Generally, research groups have focused on on-chip communication energy consumption for interconnection architecture and homogenous NoC. The objective of this work is to analyze and assess the energy dissipation for heterogeneous NoC-based MPSoC platform running a video application. It identifies bottlenecks for the entire platform. Also, we propose a new modeling and simulation approach regarding the channel width and buffer sizing which have a strong impact on the performance and the overhead of the chip. We showed that there are some hot spots in the system regarding the channel width and buffer size must be optimized to get a better performance.

Simulation is performed by running a video processing algorithm. It includes the reading of video data into memory and storing it. The video processing algorithm is then partitioned, parallelized, mapped and scheduled on multi-core processors. This video chain is mapped on a heterogeneous NoC platform based on mesh topology.

3.2.1 DEVS Formalism

The Discrete Event System Specification (DEVS) formalism provides a means of specifying a system mathematically. DEVS has also been applied in many areas such as in computer [56], manufacturing [57], transportation traffic [58], command and control [59] and networking [60]. It provides a rich environment in which any phenomena could be modeled by producing a mathematical model which in turn can be simulated under the DEVS simulation environment [61]. Some input events can occur from users or from other running tasks which affect the application behavior. An application running on one intellectual property generates output for other interconnected intellectual properties. In addition, the application transition from one state to another can be described by some functions or algorithms. Hence, an application can be modeled as a discrete event system with some specific parameters that needs to be computed by observing the application under consideration. DEVS allows variation of different parameters and performance evaluation that can be performed by exploring as many NoCs architectures as possible until an optimal architecture can be found [61].

DEVS modelling and simulation define four variables: desired application, model, simulator and the experimental environment as shown in Figure 3-5. A video processing application is used to model and optimize the NoC architecture for best bandwidth, low power dissipation and low latency. The model has mathematical relations and instructions that produce traffic properties observed in the real application. The behavior of a model is the set of all possible input/output combinations. The simulator executes the model in order to emulate the real system for comparisons, evaluations and analysis. Finally, the experimental environment defines the constraints and conditions under which the system was observed to collect its output behavior. For example, the application might be running on individual processing cores, two levels of caches L1 and L2, and a shared memory between different cores.

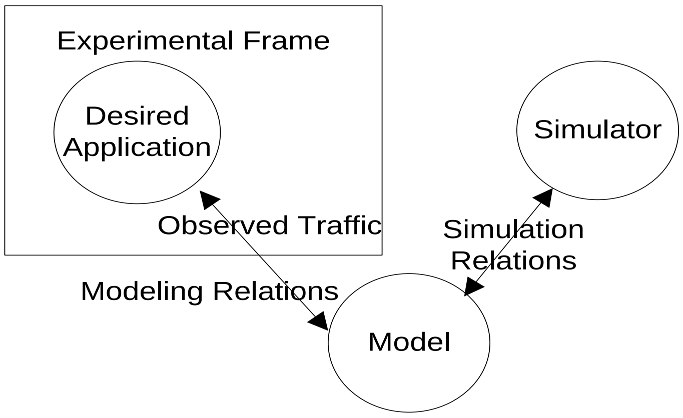


Figure 3-5: Basic relations in On-Chip Traffic modeling and simulations.

3.2.2 Architecture Description

The proposed heterogeneous NoC-based MPSoC platform is shown in Figure 3-6. It uses a 3x3 mesh NoC. The platform employs a 2-D mesh topology which contains a Master Processing Element (Master PE) which is placed in the center of the chip. It is responsible for reading of the video and distributing partitioned tasks to other nodes. Master PE partitions, parallelizes, maps and schedules on Multi-Core (slave cores). The Master PE contains a large memory to store a series of video images. When the execution starts, the Master PE allocates a subset of images to the Slave Processing Elements (Slave PEs) for processing and remaining frames are stored in the memory. This memory keeps all task codes necessary in any instance of the applications execution. During execution, tasks are dynamically loaded from the memory of the Master PE to Slave PEs. When the Slave PE finishes execution, it returns the result back to the Master PE and requests for another job.

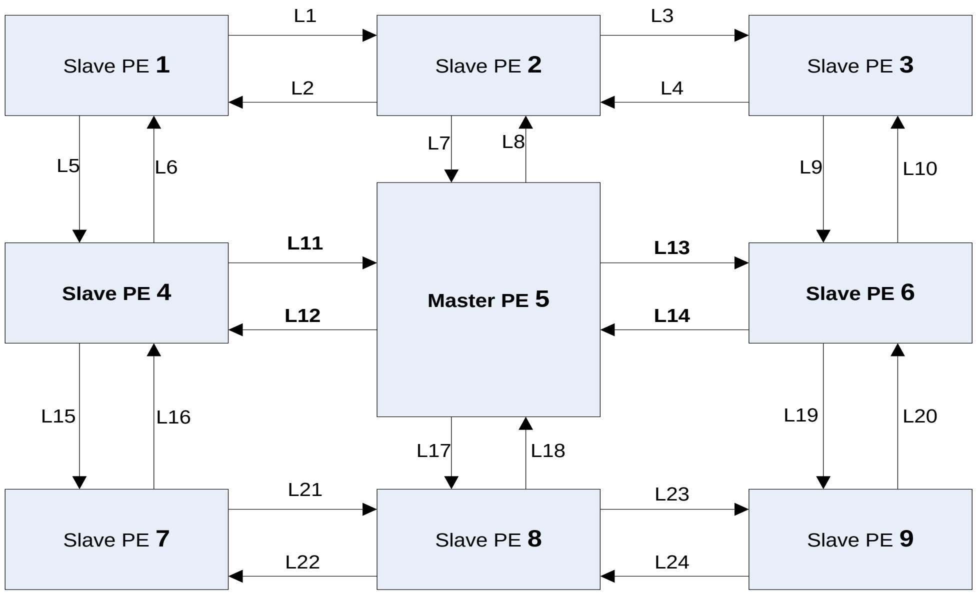


Figure 3-6: Mesh platform architecture.

Each Processing Element (PE) in Figure 3-6 has a router that has five bidirectional ports: north, west, east, south and local as shown in Figure 3-7. The local port is connected to the PE and others are connected to neighboring routers. Each router has a buffer to receive data from the PE. A single round-robin arbitration schedules grants access to incoming packets, and a deterministic distributed XY routing algorithm determines the path between source and target.

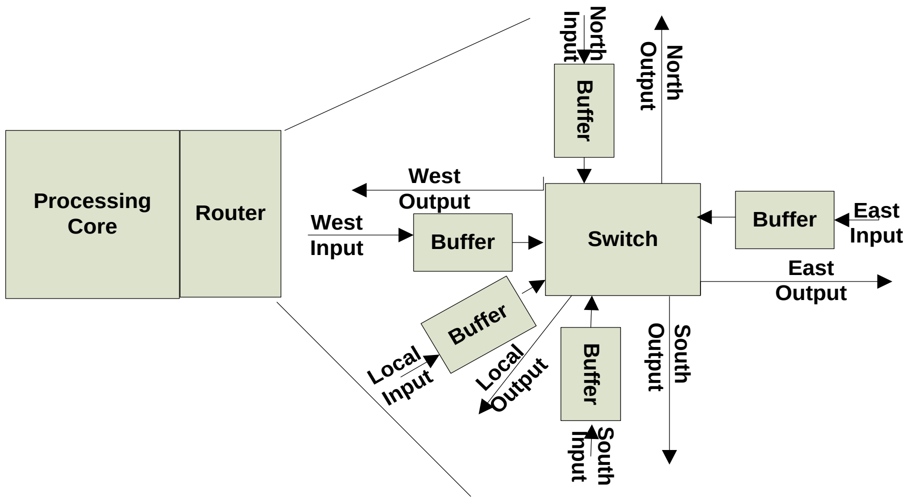


Figure 3-7: Structure of a core and the on-chip router.

3.2.3 Energy Modelling

Energy modelling for NoC architectures became one of the critical issues in NoC. There are many energy models that have been proposed [29-30]. Ye et al. [29] proposed a model for energy consumption of network routers. The bit energy (E_{bit}) metric is defined as the dynamic energy consumed when one bit of data is transported through the router:

$$E_{bit} = E_{Sbit} + E_{Bbit} + E_{Wbit} \tag{3.2}$$

Where E_{Sbit}, E_{Bbit} and E_{Wbit} represent the dynamic energy consumed by the switch, buffering and interconnection wires inside the switching fabric, respectively.

Since in Equation (3.2), E_{Wbit} is the dynamic energy consumed on wires inside the switch fabric for NoC, the dynamic energy consumed on the links between tiles (E_{Lbit})

should also be included. Thus, the average dynamic energy consumed in sending one bit of data from a tile to its neighbouring tile can be calculated as:

$$E_{bit} = E_{Sbit} + E_{Bbit} + E_{Wbit} + E_{Lbit} \tag{3.3}$$

Let E_{Rbit} to be the average energy consumption of transferring one bit of data through a router, that is:

$$E_{Rbit} = E_{Sbit} + E_{Bbit} + E_{Wbit} \tag{3.4}$$

Consequently, the average energy consumption of sending one bit of data from tile ti to tile tj is:

$$E_{bit}^{ti,tj} = n_{hops} \times E_{Rbit} + (n_{hops} - 1) \times E_{Lbit} \tag{3.5}$$

Where n_{hops} is the number of routers a given bit needs to pass through; similar energy models have also been used extensively in other works in high-level/system-level NoC [30-31].

3.2.4 Analysis and Optimization Techniques

An IR video processing application is used for simulation purposes since it provides excellent night visibility, situational awareness, less sensitive to the changes in illumination, and makes the background model more stable. The energy consumption of various components was modeled similar to the Ebit model [62] and Noxim simulator [63] using DEVS. The configuration and modeling of energy dissipation for memory and caches is based on Ayala [64]. Ayala provides in-depth analysis of the energy consumption of memory hierarchy. It measures the energy dissipation from the write access, read access, input address bus, output address bus, output data bus, pre-charge and comparators.

There are twenty-four links (unidirectional) in mesh topology as shown in Figure 3-8. The packets are routed based on XY routing algorithm. Packets are first routed in the X direction, until reaching the Y coordinate. Then they are forwarded in the Y direction. Eight links (L2, L3, L5, L9, L16, L20, L22 and L23) have not been used due to mesh topology and properties of XY routing algorithm. Therefore, these links will be removed to decrease the static energy dissipation, area cost and the total wire length. Relative traffic load for IR video processing on all remaining links (16 Links) of the mesh is shown in Figure 3-8.

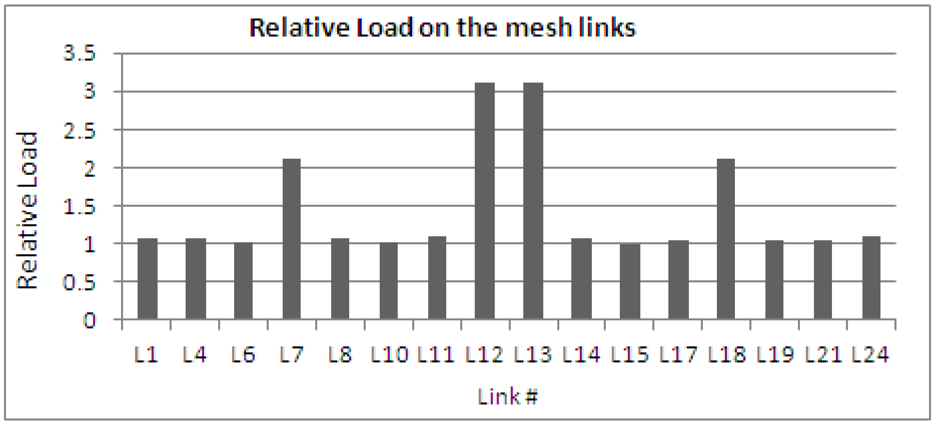


Figure 3-8: Relative load on mesh links.

Link 15 has the smallest load in the system and is denoted by one unit. Other links loads are measured relative to the load on Link 15. The highest relative load in the mesh topology is on links L12 and L13 as they are directly connected to the Master PE. Master PE distributes frames for nodes 1, 3, 4, 6, 7 and 9 via these two links. It can also be seen that links 7 and 18 have relatively moderate load. This is due to that Mater PE receives result on Link 7 from PEs 1 to 3. Similarly Link 18 has a moderate load due to return results from PEs 7 to 9.

Energy dissipation on all remaining links (16 Links) and routers of the mesh is shown in Figures 3-9 & 3-10 respectively. The highest energy dissipation in the mesh is on the links L12 and L13; because they are directly connected to the Master PE (the utilization of these links is very high). Similarly the router number 5 (Master PE) has the highest energy dissipation due to its high utilization.

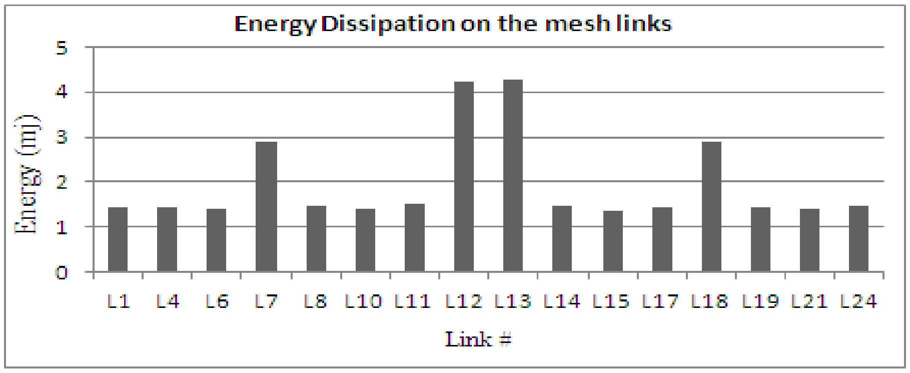


Figure 3-9: Energy dissipation on the mesh links.

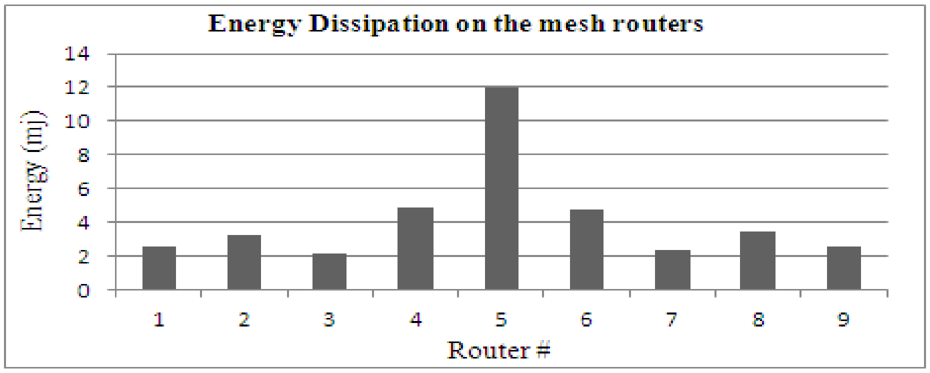


Figure 3-10: Energy dissipation on the mesh routers.

It can be seen from Figure 3-8 that the physical network model needs different bandwidths and different buffer sizes for links and routers. Since there is a lot of congestion on the master core links and its router, therefore the bandwidth of the links connected to the master core (L12 and L13) is increased by a factor of two by adding more virtual channel. The buffer sizes of the router 4, 5 and 6 are increased by a factor of two. A flowchart for these optimizations is shown in Figure 3-11.

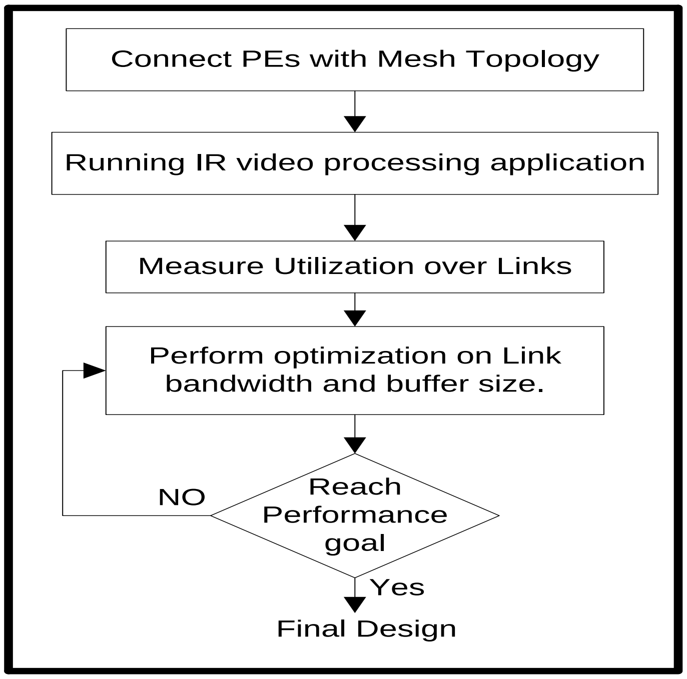

Connect PEs with Mesh Topology
Running IR video processing application
Measure Utilization over Links
Perform optimization on Link bandwidth and buffer size.
NO
Reach Performance goal
Yes
Final Design

Figure 3-11: Flow chart of NoC simulation and design.

3.3 Parallelization of Direct Data Domain (D^3) for Space Time Adaptive Processing (STAP)

3.3.1 Introduction

Space Time Adaptive Processing (STAP) is a signal processing technique used to suppress the effects of co-channel interference, Inter-Symbol Interference (ISI), and jammers in communications systems. STAP algorithms contribute in achieving greater capacity and communication quality [65]. Many variations of STAP algorithms have been proposed to effectively detect a moving target in the presence of cluttered environment [66-70].

The fundamental principle in all STAP algorithms involves the use of multiple receive antennas on the receiving platform. The incoming digitized received signals are adaptively weighted using a variety of algorithms in order to steer the antenna gain towards the desired signals while nulling the signals from unwanted noise and interference. Figure 3-12 shows the baseline setup for most STAP implementations.

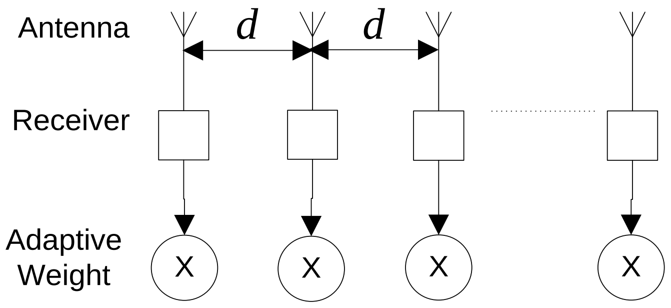


Figure 3-12: Baseline STAP receiver configuration.

Due to the fast-changing clutter scenario, the received data may not be stationary and statistically-based methods such as covariance matrix suffer from performance degradation and fail if the interference scenario ever becomes heterogeneous [71]. Direct Data Domain (D^3) method does not assume the stationary nature of the data and can also deal with heterogeneous interference. It can effectively suppress the clutter [72-75].

3.3.2 STAP Algorithms

Two algorithms for STAP are examined and discussed in this section; interference covariance matrix estimation (a statistical method) and Direct Data Domain (D^3) processing approach (a non-statistical method).

3.3.2.1 Interference Covariance Matrix Estimation

Interference covariance matrix estimation is a well-known classical approach to

STAP implementation. It utilizes tapped delay lines and adaptive weights as shown in Figure 3-13.

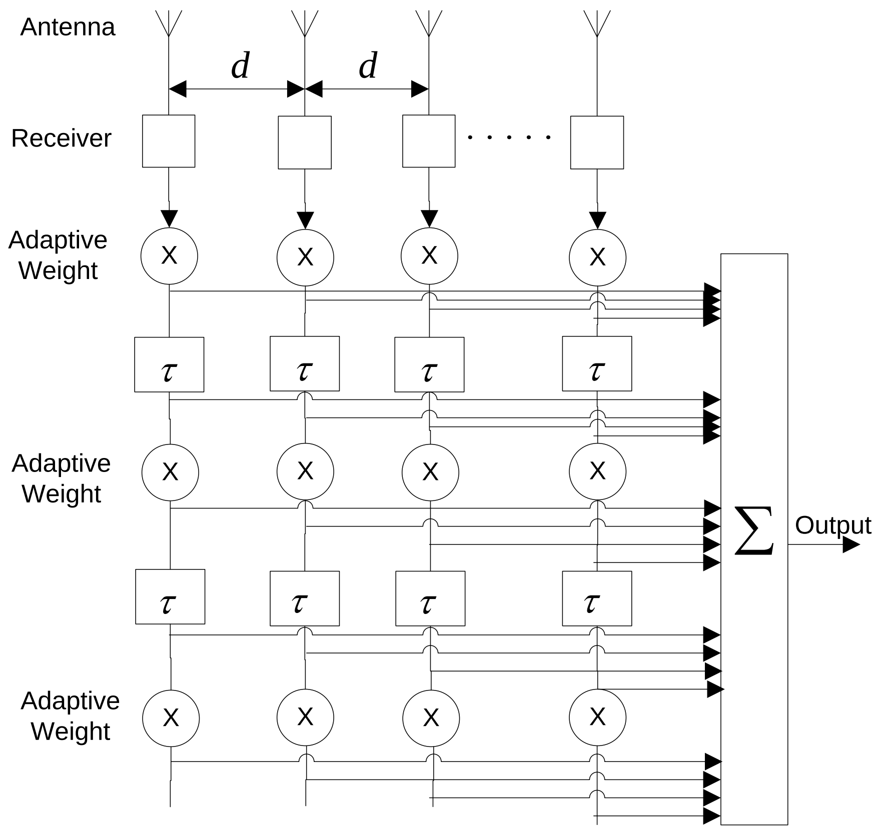


Figure 3-13: N-element antenna array with M taps.

Each of the tapped delay signals is adaptively weighted that minimizes interference and noise while preserving or enhancing the desired signal. The final output signal is the sum of all the weighted taps. The classical interference covariance estimation approach has been proven to have excellent interference cancellation performance in homogeneous correlated interference scenarios. However, if the interference scenario ever becomes heterogeneous, statistical methods will fail [71]. This lends non-statistical methods such as D^3 as the algorithm of choice with inferior homogeneous performance as a tradeoff.

3.3.2.2 Direct Data Domain (D^3) Approach

The Direct Data Domain (D^3) approach uses no tapped delay lines following the weight on each receiver element; therefore, the voltages are then summed and the output $Y(t)$ is given as shown in Figure 3-14.

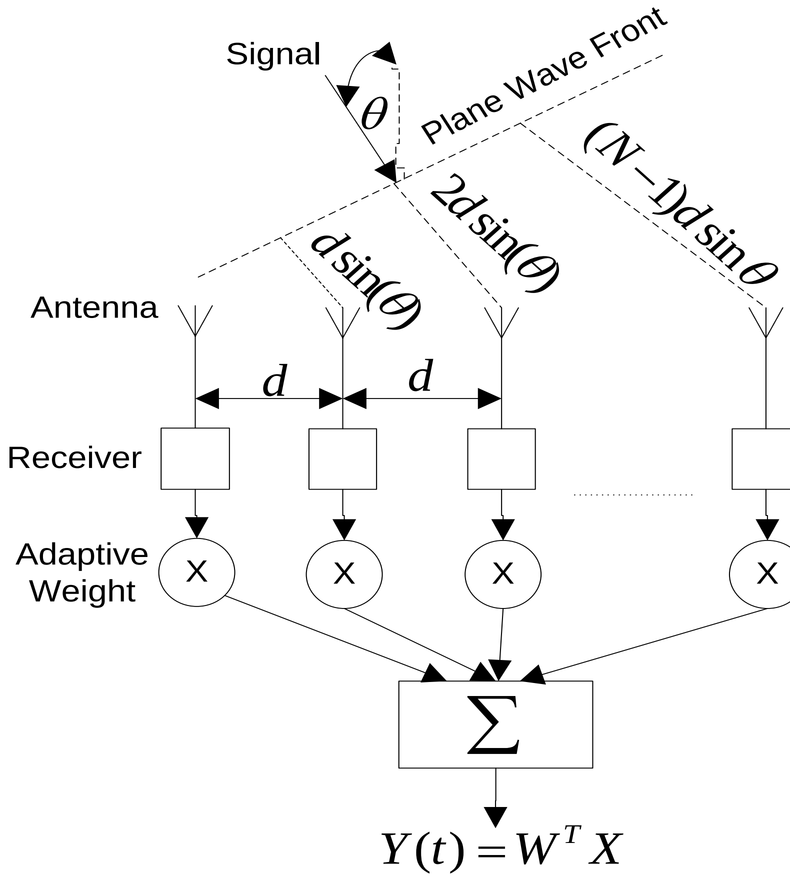


Figure 3-14: N-element antenna array.

The signal arrives at each antenna at different times depending upon the direction of arrival of the target and the geometry of the array. Let $S(t)$ denote the desired signal. The received signal voltage on each antenna element is the sum of the desired signal plus noise and interference [75-76]. Hence:

$$X_k(t) = S_k(t) + N_k(t) \qquad k = 1, 2, ...N \tag{3.6}$$

Where $x_k(t)$ is the received voltage on an antenna and $S_k(t) \,\&\, N_k(t)$ are the desired signal and noise contributions on the antenna respectively. If θ is the direction of the desired signal, then the received signal on each antenna element can be modeled [75-76] by:

$$S_k(t) = S(t)\; e^{(j2\pi kd/\lambda)\sin\theta} \tag{3.7}$$

Where d is the distance between antenna elements and λ is the wavelength of the desired signal [75-76]. The received signal after weighting then becomes:

$$Y(t) = \sum_{k=1}^{M} W_k X_k(t) \tag{3.8}$$

Or in a matrix form:

$$Y(t) = W^T X \tag{3.9}$$

Where (T) denotes transpose and

$$W^T = \begin{bmatrix} W_1 \; W_2 \; W_3 \; \cdots\cdots W_M \end{bmatrix} \tag{3.10}$$

$$X^T = \begin{bmatrix} X_1 \; X_2 \; X_3 \; \cdots\cdots X_M \end{bmatrix} \tag{3.11}$$

Where M is the number of degrees of freedom set by $M = (N+1)/2$.

If the actual received voltages are written as a vector X and the modeled received desired signals are written as a vector S, then the difference contains just the noise and interference which is desired to be eliminated [75-76]. This can also be written as:

$$\bar{X} - \alpha\bar{S} \tag{3.12}$$

$$\begin{bmatrix} X_1 & X_2 & \cdots & X_M \\ X_2 & X_3 & \cdots & X_{M+1} \\ \vdots & \vdots & \ddots & \vdots \\ X_M & X_{M+1} & \cdots & X_N \end{bmatrix} - \alpha \begin{bmatrix} S_1 & S_2 & \cdots & S_M \\ S_2 & S_3 & \cdots & S_{M+1} \\ \vdots & \vdots & \ddots & \vdots \\ S_M & S_{M+1} & \cdots & S_N \end{bmatrix}$$

Where α is the complex amplitude of the desired signal.

So, the goal of D^3 is to find an adaptive weighting vector that minimizes this interference output and gets the target desired signal without distortion. Calculating the weight vector requires a method to solve the generalized eigenvalue problem as shown in Equation (3.13).

$$[X-\alpha S][W]=[0] \tag{3.13}$$

The most time-consuming part of the D^3 algorithm is the computation of the generalized eigenvalue problem. Its computation consists of a sequence of complex operations such as matrix multiplication, matrix inversion, division, additions and subtractions. Also, the development of solving matrix inversion and finding the parameters of linear algebraic equations require a large number of complex computations.

Therefore, in this work, a combination of inexact inverse iteration and Conjugate Gradient (CG) methods is proposed for solving the generalized eigenvalue problem required in D^3 algorithm. Transformed algorithms are parallelized and have been implemented on both FPGA and GPU.

3.3.3 Inexact Inverse Iteration for Generalized Eigen Value Problem

There are many algorithms that are available to solve the generalized eigenvalue problem such as subspace iterations, the Lanczos algorithm, Arnoldi algorithm, and rational Krylov subspace method [77-78]. Many of these algorithms require inverting and factorizing a shift matrix and are limited to problems of moderate size. Inexact inverse iteration has been proven to converge linearly if the inner thresholds satisfy certain conditions [79-81]. This method converges linearly at essentially at the same rate as the exact case [82]. The standard inverse iteration algorithm to solve the generalized eigenvalue problems can be described by the following pseudo code:

Standard Inverse Iteration Algorithm

```
Given $x_0$;
For k ⟵ 0 to till convergence
$y_{k+1} = X^{-1} S w_k$;
$w_{k+1} = y_{k+1} \div \|y_{k+1}\|$
End
```

This method uses X^{-1} which may not be available because of its singularity property. Moreover, it can be computationally intensive. So, an iterative method can be used to solve $Xy_{k+1} = Sw_k$, this is called inner iterations [83]. The inverse iteration method used in this work is called the outer iteration. So, for each step of outer iteration, we seek an approximate solution y_{k+1} that satisfies $Xy_{k+1} = Sw_k + q_k$, where q_k is called residual that satisfies certain termination criterion in the inner iteration ($\|q_k\| < \varepsilon_k \|y_{k+1}\|$) where $\varepsilon_k = r^k$.

If we use y_k as an initial approximation to y_{k+1}, we aim at solving:

$$X\delta_k = Sw_k - Xy_k + q_k \text{ where } y_{k+1} = y_k + \delta_k \tag{3.14}$$

So, we consider the following version of inexact inverse iterations [83] to solve the generalized eigenvalue problem:

Inexact Inverse Iteration Algorithm

```
Given $w_0$; set $y_0$=0.
For k ⟵ 0 to till convergence
$r_k = Sw_k - Xy_k$;
$X\delta_k = r_k + q_k$ (Solved using both CG and LU decomposition)
```

```
        Stopping criterion with $\|q_k\|$ satisfying $\|q_k\| < \varepsilon_k \|y_{k+1}\|$
    $y_{k+1} = y_k + \delta_k$
    $\delta_{k+1} = \max(y_{k+1})$
    $w_{k+1} = y_{k+1} \div \delta_{k+1}$
End
```

In order to find the weights, we need to find an inverse of matrix X ($X\delta_k = r_k + q_k$) which requires a complex computation. Therefore, two methods are implemented and parallelized for solving the matrix inversion problem; Conjugate Gradient (CG) method [84] and LU-Decomposition method [85].

3.3.4 Analysis and Optimization Techniques

Parallel processing in both FPGA and GPU has been used to achieve the real-time constraints. The following sections describe some of the optimizations that have been incorporated.

3.3.4.1 Parallel implementation on FPGA

FPGAs are appropriate to provide increased computation time as they provide flexibility in allocating needed resources. Figure 3-15 shows the computational steps to implement the D^3 algorithm using Inexact Inverse Iteration Algorithm (IIIA) and Conjugate Gradient (CG). It also shows data dependency between various operations and their computational sequence. It is proposed that the input should be stored in a DDR3 SDRAM. Generally, SDRAM is used for performance critical applications where its architecture supports high bandwidths, low power dissipation, high densities, and fast access times [86-88]. So, it has the ability to transfer the data at high rate while allowing greater capacities [87].

Xilinx has released a memory interface for DDR3 which handles everything for communication between the system and memory [87]. DDR3 provides two burst modes for both reading and writing and a prefetch buffer [86]. DDR3 memory has increased the bandwidth and the Column Access Strobe (CAS) time is very close to 0 ns [87].

In order to set up the interface between the FPGA and DDR3 SDRAM external memory, Memory Interface Generator (MIG) must be used to generate the RTL code of the memory interface as shown in Figure 3-16. MIG is a tool under the core generator of Xilinx software which supports memory interface for FPGA. Memory interface from MIG is composed of many modules which make both communication and testing of the memory more easily.

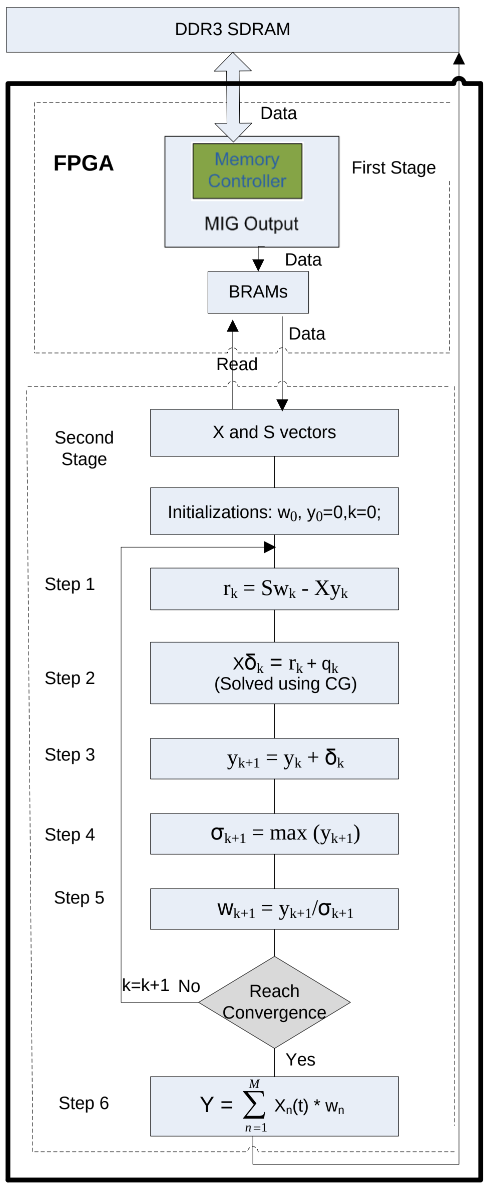


Figure 3-15: D^3 computational steps

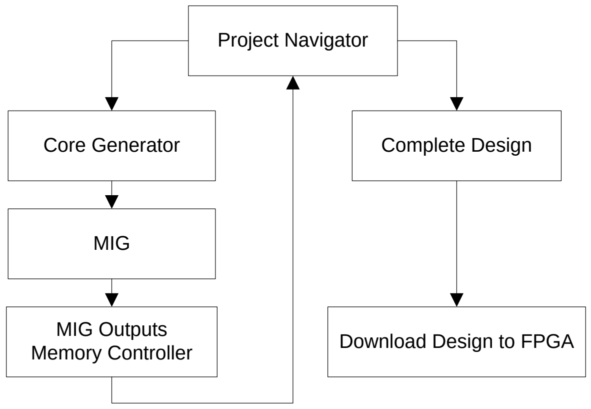


Figure 3-16: Memory interface RTL code generation using MIG tool.

The received and desired signals have a vector structure with dimension N based on Equations (3.6) and (3.7). However, step one of D^3 algorithm in Figure 3-15 needs to construct and build the matrices X and S based on Equation (3.12). These matrices will be used in many steps throughout the algorithm. The building of these matrices requires additional resources for storage and computation due to following reasons:

- Building of X and S matrices each with a size (M*M) requiring two for-loops with a time complexity of O (M^2).
- $2M^2$ registers are required for matrices storage.

However, the rows of matrices X and S are the same with only shifting by one element. These matrices are only used through the algorithm by multiplying them with a vector. So, this structure can be exploited to implement a new code for matrix-vector multiplication. This new code contributes to the following advantages:

- Eliminating construction of matrices X and S.
- Storage areas for these matrices are eliminated.

- Accessing the data in the memory is more efficient as we deal with vector instead of a matrix where all the data are contiguous. This contributes in reducing the miss rate and it will be very helpful for prefetching technique to hide memory latency by overlapping the execution with memory access.

Following code is used to implement the matrix-vector multiplication in steps one and six of the algorithm in Figure 3-15.

Serial Matrix-Vector Multiplication

```
Declare ←  X[N], Res[M], sum=0, index=0;
Loop1: While (index<M){
Loop2:   For i← 0 to M
 {sum=sum+X[index+i]* vector[i];}
Res[index]=sum; sum=0; index++}
```

Moreover, our new code can be performed in parallel since there is no dependency. Loop unrolling and loop pipelining are two effective types of parallelization methods in FPGA. In order to choose the best parallelization method, both of them have been experimented via simulation. The performance parameters in terms of latency, memory usage, and power dissipation are obtained as shown in Table 3.1. One interesting observation is that applying loop unrolling to the outer loop doesn’t achieve performance enhancement as with the inner loop. This is because the inner loop performs the operation on every element in the array, whereas the outer loop only uses the first element of each row.

Loop unrolling was the fastest compared with other options when both inner and outer loops are fully unrolled. However, it causes excessive memory usage, area, and high power dissipation. On the other hand, the performance of the pipelining method for the outer loop was found to be approximately the same as the unrolling method for both loops in terms of latency. Also, pipelining technique worked better than loop unrolling when unrolled one of the loops. So, the Loop pipelining technique of the outer loop is chosen as a tradeoff since it performs in parallel with the reuse of the same hardware resources across different stages to achieve high speed with awareness of power, area, and cost.

Table 3.1: Latency and resource utilization of matrix-vector multiplication for N=100 and m=10 on Artix7 (XA7A100T CSG324) -1q

	No Optimization	Pipelining Outer Loop	Pipelining Inner Loop	Pipelining both Loops	Unrolling Outer Loop	Unrolling Inner Loop	Unrolling both Loops
Latency (cycles)	9201	513	1012	2001	9100	1701	511
Clock period (ns)	6.68	8	7.18	8	6.68	8	8
FF	255	1424	302	264	16370	1421	128018
LUT	215	1202	306	225	22878	1208	111037
DSP48E	4	40	5	4	400	40	4000
Power	46	264	59	46	3964	266	24304

Step two requires the solving of matrix inversion problem. Conjugate Gradient (CG) and LU-Decomposition are two methods which can be used for solving this constraint. Following sections will explain each method and how they are parallelized.

LU decomposition

LU decomposition method is a kind of exact solutions of system of linear algebraic equations and to find the inverse of a matrix which was introduced by mathematician

Alan Turing [89]. This method attempts to decompose the coefficient matrix into two lower and upper triangular matrices. Suppose that A can be factored into the product of a lower triangular Matrix L and an upper triangular matrix U such that $A = LU$, when this is possible, we say that A has an LU-decomposition (we assume that A is a nonsingular matrix). Then, to solve the system of $Ax = b$, it is enough to solve this problem in two stages using Equations (3.15) and (3.16) [85]:

$$Lz = b \text{ (Solve for } z\text{)} \tag{3.15}$$

$$Ux = z \text{ (Solve for } x\text{)} \tag{3.16}$$

To derive an algorithm for the LU-decomposition of A, we start with the formula for matrix multiplication using Equation (3.17) [85]:

$$a_{ij} = \sum_{s=1}^{n} l_{is} u_{sj} = \sum_{s=1}^{\min(i,j)} l_{is} u_{sj} \tag{3.17}$$

where $l_{is} = 0$ for $s > i$, and $u_{sj} = 0$ for $s < j$ is used.

Each step in this process determines one new row of U and one new column in L. At step k, it can be assumed that the rows $1, 2, ..., k-1$ have been computed in U and that columns $1, 2, ..., k-1$ have been computed in L. Putting $i = j = k$ in Equation (3.17), we obtain [78]:

$$a_{kk} = \sum_{s=1}^{k-1} l_{ks} u_{sk} + l_{kk} u_{kk} \tag{3.18}$$

If u_{kk} or l_{kk} has been specified, then Equation (3.18) can be used to determine the other coefficients. Equation (3.17) can be used to compute for the k^{th} row ($i = k$) and the k^{th} column ($j = k$), respectively [78].

$$a_{kj} = \sum_{s=1}^{k-1} l_{ks}u_{sj} + l_{kk}u_{kj} \; (k+1 \le j \le n) \tag{3.19}$$

$$a_{ik} = \sum_{s=1}^{k-1} l_{is}u_{sk} + l_{ik}u_{kk} \; (k+1 \le j \le n) \tag{3.20}$$

Equation (3.19) can be used to obtain the elements u_{kj} if $l_{kk} \neq 0$. Similarly, Equation (3.20) can be used to obtain the elements l_{ik} if $u_{kk} \neq 0$ [85].

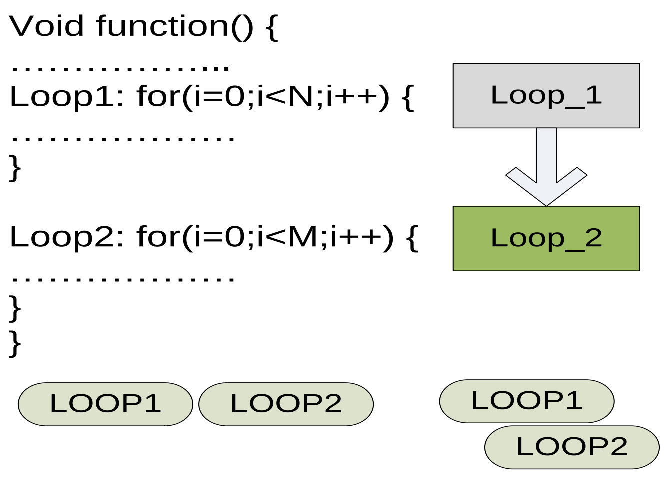
The pseudo code of the LU-factorization algorithm is presented as:

LU Decomposition for solving $X\delta_k = r_k + q_k$

```
$Given\ X[M][M]; r_k[M]; q_k[M]$
$Define\ N;$
$Define\ M = (N+1)/2;$
$Define\ \delta_k[M] = \{0\};$
$Define\ b = r_k + q_{k;}$
$Define\ l[M][M] = \{0\}, u[M][M] = \{0\}, sum, z[M] = \{0\};$
For $k \leftarrow$ 1 to $M$
  { $u[k][k] = 1;$
Loop1: For $i \leftarrow k$ to $M$
      { $sum = 0;$
For $p \leftarrow$ 1 to $k$-1
 $sum += l[i][p]*u[p][k];$
 $l[i][k] = X[i][k] - sum;$
      }
Loop2: For $j \leftarrow k+1$ to $M$
      { $sum = 0;$
For $p \leftarrow$ 1 to $k$-1
 $sum += l[k][p]*u[p][j];$
 $u[k][j] = (X[k][j] - sum)/l[k][k];$
      } }
//*FINDING Z; LZ=b by forward substitution */
For $i \leftarrow$ 1 to $M$ ;
 { $sum = 0;$
For $p \leftarrow$ 1 to $i$
$sum += l[i][p] * z[p];$
$z[i] = (b[i] - sum) / l[i][i];$
   }
//**********FINDING X; UX=Z***********//
For $i \leftarrow M$ to $0$ ; $i > 0; i--;$
   { $sum = 0;$
For $p \leftarrow M$ to $i$ ; $p > i; p--;$
$sum += u[i][p] * \delta[p]$ ;
$\delta[i] = (z[i] - sum) / u[i][i];$   }
```

LU-Decomposition must be implemented and parallelized in an efficient way to get benefits from the resources of parallel processing platforms. LU decomposition has two loops (Loop1 and Loop2) that depend on each other. So, they can't be executed simultaneously due to the dependency between them. Loop2 needs the value of l to complete the computation and Loop1 needs the value of u from Loop2. However, Loop 2 can begin execution after completion of the first iteration of the Loop1. Loop1 has an additional iteration than Loop2. So, the dataflow pipelining optimization techniques between these loops are very effective since it takes a sequential loop and creates parallel processing architecture. Without dataflow pipelining, Loop1 must execute and complete all its iterations before Loop2 can begin execution. Loop2 will then execute and will provide the value of u to Loop1. It will reduce the overall throughput and increase the latency. However, with dataflow pipelining, these loops can be allowed to operate in parallel in such a way that when Loop1 finishes the first iteration then it forwards the value l to Loop2. Loop2 will supply the value of u to Loop1 after completing its all iterations.

Dataflow mechanism inserts channels between the loops to insure that the data can flow asynchronously from the first Loop to the next one as shown in Figure 3-17. This improves both the throughput and the latency.



(a) Without Dataflow (b) With Dataflow

Figure 3-17: Loop dataflow pipelining technique.

<u>Conjugate Gradient</u>

The computation of step two requires following computation and storages:

- Computation of matrix inversion $Z = (X)^{-1}$.
- Matrix-vector multiplication $F = Z * y$ where y = rk + qk
- Storage for Z and F.

However, the computation of vector (δ) in step two in Figure 3-15 can be performed in an efficient way using the CG method. CG is a good option for this type of computation due to the following reasons:

- It can deal with different dimensions where many other inversion algorithms require a square matrix.

- Eliminates performing the computation of matrix inversion and the matrix-vector multiplication separately.
- It eliminates matrix inversion storage requirements (Z).

The pseudo code of the CG method is presented as:

Serial Conjugate Gradient Method (CG)

```
Declare ← X[M];
For c ← 1 to M {                          While (i<M){
For d ← 0 to N                            For d ← 0 to N
{ sum=sum+X[c][d]* δ[d];}                 {sum+=X[i+d]* δ[d];
r[c]=y[c]-sum; sum=0; }

For d ← 0 to M {
        {p[d]=r[d]; rsold=rsold+r[d]*r[d];}

For index ← 1 to converge {
                                           While (i<M){
For c ← 1 to M {                          For d ← 0 to N
     For d ← 0 to N {          Region 1   {sum+=X[i+d]*p[d];
       Sum+=X[c][d]*p[d];}                Ap[i]=sum;sum=0;i++
Ap[c]=sum;  sum=0;}

  For d ← 0 to M {
tem=tem+p[d]*Ap[d];}           Region 2
alpha=rsold/tem;

  For d ← 0 to M {
δ[d]= δ0[d]+alpha*p[d];        Region 3
r[d]=r[d]-alpha*Ap[d];}

  For d ← 0 to M {
rsnew=rsnew+r[d]*r[d];}        Region 4

  For d ← 0 to M {
p[d]=r[d]+(rsnew/rsold)*p[d];} Region 5
rsold=rsnew;}
```

Since we treat X as its original structure (vector), two steps of CG method need to be modified to match our new code as shown in its pseudo code. Full parallelization of CG can’t be achieved due to data dependencies as the next computational step depends on the result of the current step. However, the CG is divided into five computational regions where each region can be performed in parallel by applying pipelining technique.

Moreover, it is not necessary for region 2 to wait until region 1 completes all its iterations. So, region 2 can start execution after the first iteration of region 1 is completed. This can be exploited by applying the dataflow technique between these regions where the data can flow asynchronously from the first region to the next one. So, regions 1 and 2 execute in a pipelined fashion until all iterations are completed. Region 1 forwards the value of the current iteration to region 2 and begins with the next iteration at the same time region 2 can start execution. Similar approach is applied to regions 3 and 4.

Third and fifth steps of the algorithm in Figure 3-15 require vector-vector addition and division respectively. These steps can be fully parallelized by applying loop unrolling technique where each step is performed by one clock cycle instead of M clock cycles. Step four requires finding the maximum element in a vector which is parallelized by applying pipelining technique where the latency is improved as shown in Table 3.2.

Table 3.2: Latency and resource utilization of finding the maximum element with vector size 100 (N=100) on Artix7 (XA7A100T CSG324) -1q

	No Optimization	Pipelining Optimization
Latency (cycles)	200	102
Clock period (ns)	6.24	6.24
FF	46	49
LUT	124	131
Power	15	16

3.3.4.2 Parallel implementation on GPU.

D^3 algorithm is also implemented and parallelized on GPU architecture to get benefits from the use of parallel processing to achieve real time requirements. Also, the result of HLS tool on FPGA implementation can be compared against the outcomes of GPU architecture to draw fair comparisons.

Step one of the algorithm in Figure 3-15 requires 2M vector-vector multiplications and M vector-vector subtractions. However, these operations can be fully parallelized by 2M threads where each thread multiplies one vector-vector multiplication as shown in the following code:

Parallel Matrix-Vector Multiplication

```
int index ⟵ Thread index;
Fori ⟵ 0 to M
{ Res[index] + =X[index+i]* vector[i];}
```

Also, vector-vector subtraction is accomplished by M threads where each thread subtracts one subtraction operation. Following paragraphs describe the parallelization method for each region in the CG method as shown in Figure 3-18.

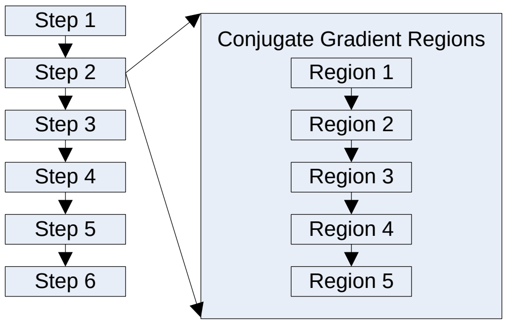


Figure 3-18: D^3 computational steps with Conjugate Gradient (CG) computation regions.

Region 1 of CG is parallelized using the same technique as in step one where each thread works on one vector-vector multiplication. Regions 3 and 5 in CG are fully parallelized by M threads where each loop is executed by one clock cycle instead of M clock cycles as shown in the following pseudo code:

Parallel Method for **Region 3**

```
int index ⟵ Thread index;
δ[index] = δ0[index]+alpha* p[index];
r[index] =r[index]-alpha*Ap[index];
```

Regions 2 and 4 of CG require vector-vector multiplication. So, a synchronization scheme is required since the global accumulation variable is shared between all the threads of the multiplied vector elements that must be protected to get correct results. Full parallelization of these regions by M threads adds an overhead more than the desired computation itself because of synchronization scheme. Therefore, we run these regions under the GPU platform with different number of threads with a tradeoff with synchronization scheme to choose the optimal number of threads where each thread multiplies ten elements and save a result into its private variable. A pseudo code of parallel vector-vector multiplication for regions 2 and 4 is as follows:

Parallel Vector-Vector Multiplication

```
int index ⟵ Thread index;
Tile_Size=10;.
x=index*Tile_Size.
y=(index+1)*Tile_Size.
For i⟵  x to y {
Local_sum+=Vec[i]*Vec[i];}
__syncthreads();
Global_sum+=Local_sum;
Unlock_suncronization();
```

Third and fifth steps of the algorithm in Figure 3-15 require vector-vector addition and division respectively. These steps can be fully parallelized where each step is performed by M threads that are executed in one clock cycle instead of M clock cycles.

Step four in Figure 3-15 requires finding the maximum element in a vector. In order to parallelize this operation efficiently, we have divided the vector by multiple threads where each thread finds its local maximum element on its own vector part. Then the global maximum element is found by comparing these local maximum elements. However, this technique requires a lock and barrier synchronizations to ensure correct results since there is a data shared by all the threads. Following code is used to find the maximum element in a vector.

Parallel of Finding the Maximum element in a Vector

```
Declare ⟵ Y[N], Max;
Void max (int P){
intlocal_max=Y[P*N/M];
        For i ⟵ P*(N/M) to (P+1)*(N/M)
            If(Y[i]>local_max) {local_max=Y[i];}
Lock_Synchronization ( );
If(local_max>Max) {Max=local_max;}
Unlock_Synchronization ( ); Barrier_Synchronization ( );
```

3.4 Parallelization of Extensive Cancellation Algorithm (ECA) for Passive Bistatic Radar (PBR)

3.4.1 Introduction

Bistatic radars have the transmitter and the receiver antennas at separate locations [90-91]. On the other hand, conventional monostatic radars have the transmitter and the receiver at the same location. Passive radars have only a receiver and they simply listen to transmitted data from other radars or electromagnetic emitters. Passive radars are able to utilize signals of opportunity available in the environment. These signals may be broadcast FM radio, TV signals, mobile phones, and others. They can be very useful in detecting targets without emitting any Radio Frequency (RF) of their own. The scattered RF signal can be received by one antenna called surveillance antenna and compared with the received signal from another antenna which is called a reference antenna.

Recently, Passive Bistatic Radars (PBR) have received great interest among radar researchers. PBRs have low cost, reduces electromagnetic pollution and the interference with other necessary sources. They also do not need dedicated transmitter and frequency allocation. One drawback of PBR is that transmitted signals are not under the control of the radar designer. The PBR then deals with unknown transmitted RF signals and has a variable structure of the ambiguity function. So, Passive radars do not have luxury of having appropriate ambiguity function and narrow peaks in both range and Doppler [92]. Therefore, PBR requires the use of two correlated passive antennas to collect RF signals in order to detect the desired target. The surveillance antenna steers towards the area that needs to be surveyed and the reference antenna steers towards the transmitter antenna [92]. In order to get a good signal to noise ratio, PBR requires a long integration time for surveillance since the received RF signals are continuous waveforms [93].

PBR is based on the promise of use of unknown RF transmitted signal [94-95]. It contributes to the following:

- Strong clutter can mask some targets.
- A small fraction of the direct signal can mask target echoes.
- Strong target echo can mask other echoes from other targets.

A number of researchers [96-102] have used different techniques to overcome the above concerns. Colone et al. has proposed Extensive Cancellation Algorithm (ECA) [92] which is a very effective way to mitigate the direct signal, multipath and clutter echoes in PBR. Also, it is able to detect desired target accurately for the strong - clutter environment and long-range detection. However, ECA is a computationally intensive algorithm and may not be able to provide target information in real-time. ECA will benefit from the parallel processing to achieve real-time requirements. Parallel processing systems may utilize multi-core, Network on Chip (NOC), Field Programmable Gate Arrays (FPGAs), and Graphic Processing Units (GPUs). Parallel hardware should be efficient in terms of latency, area, power consumption, cost, and flexibility.

3.4.2 Signal Model and Reference Scenario

An example of a PBR geometry for detecting and locating the desired target is shown in Figure 3-19. ECA needs two separate antennas; the reference antenna steered toward the transmitter and the surveillance antenna looking in the direction of the surveyed area.

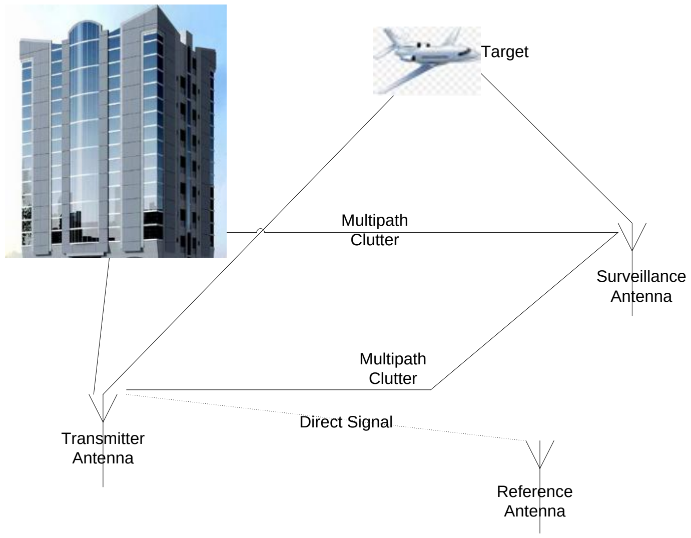


Figure 3-19: PCL geometry.

The total received signal in the surveillance antenna [92] is given by

$$s_{surv}(t) = A_{surv} d(t) + \sum_{m=1}^{N_T} a_m d(t-\tau_m) e^{j2\pi f_{dm} t} + \sum_{i=1}^{N_c} c_i d(t-\tau_{ci}) + n_{surv}(t), \quad 0 \le \mathrm{t} < \mathrm{T}_0 \tag{3.21}$$

where T_0 is the observation time; *d(t)* is the complex envelope of the direct signal; A_{surv} is the amplitude of the direct signal; a_m, τ_m, and f_{dm} are the amplitude, delay, and the Doppler frequency respectively of the mth target; c_i and τ_{ci} are the amplitude and the delay of the ith stationary ground scatter; $n_{surv}(t)$ is the thermal noise contribution.

The total received signal in the reference antenna [92] is given by

$$s_{ref}(t) = A_{ref} d(t) + n_{ref}(t) \tag{3.22}$$

where A_{ref} is the complex amplitude and $n_{ref}(t)$ is the thermal noise contribution.

Collected samples $s_{surv}[i]$ at the surveillance antenna are in the following vector form [92]:

$$s_{surv}[i]=[s_{surv}[0],s_{surv}[1],s_{surv}[2],.....,s_{surv}[N-1]]^{T} \tag{3.23}$$

With $i=0,......,N-1,\ t_i=i/f_s$

Where N is the number of samples to be integrated, f_s is the sampling frequency which satisfies the Nyquist theorem.

Similarly, the collected samples at the reference antenna are in the following vector form [92]:

$$s_{ref}=[s_{ref}[-R+1],......,s_{ref}[0],......,s_{ref}[N-1]]^{T} \tag{3.24}$$

Where $R-1$ is the number of additional samples included in integration time in order to achieve an acceptable signal to noise ratio.

3.4.3 Extensive Cancelation Algorithm (ECA)

ECA for PBR is developed based on the Least Square (LS) technique [102-103]. It exploits the signal model and finds the minimum residual signal power after cancellation of the direct signal and clutter [92], thus:

$$\min\left\{\| s_{surv}-X\alpha \|^{2}\right\} \tag{3.25}$$

$$X=B[\Lambda_{-p}S_{ref}.....\Lambda_{-1}S_{ref}S_{ref}\Lambda_{1}S_{ref}......\Lambda_{p}S_{ref}] \tag{3.26}$$

where B is an incidence matrix that selects only the last N rows of the following matrix [92]:

$$\begin{gathered} B=\left\{b_{ij}\right\}_{i=1,......,N\ j=1,......,N+R-1,} \\ b_{ij}=\begin{cases}1 & i=j-R+1\\ 0 & \text{otherwise}\end{cases} \end{gathered} \tag{3.27}$$

Λ_p is a diagonal matrix that applies the phase shift corresponding to the pth Doppler bin [92]:

$$\Lambda_p = \begin{pmatrix} 1 & 0 & \cdots & 0 \\ 0 & e^{j2\pi p} & \cdots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & e^{j2\pi p/(N+R-1)} \end{pmatrix} \tag{3.28}$$

$$S_{ref} = [s_{ref} \;\; Ds_{ref} \;\; D^2 s_{ref} \cdots D^{K-1} s_{ref}] \tag{3.29}$$

Where D is defined as

$$\begin{aligned} D &= \{d_{ij}\}_{i,j=1,.......,N+R-1,} \\ d_{ij} &= \begin{cases} 1 & i = j+1 \\ 0 & \text{otherwise} \end{cases} \end{aligned} \tag{3.30}$$

Solving Equation (3.25) yields

$$\alpha = (X^H X)^{-1} X^H s_{surv} \tag{3.31}$$

Therefore, the surveillance signal after cancellation becomes [92]:

$$s_{ECA} = s_{surv} - X\alpha = [I_N - X(X^H X)^{-1} X^H] s_{surv} \tag{3.32}$$

The two-dimensional Cross Correlation Function (2D-CCF) at the output of the cancellation filter becomes [92]:

$$\xi[l,p] = \sum_{i=0}^{N-1} s_{ECA}[i].s^*_{ref}[i-l].e^{-j2\pi pi/N} \tag{3.33}$$

Where $l = 0 \ldots R-1$ is the time bin representing the time delay respect to the direct signal. p is the Doppler bin representing the Doppler frequency of the backscattered echo from a target.

Equation (3.32) shows many complex operations such as complex matrix multiplication, Hermitian, inversion etc. The algorithm has a complexity of $O[NM^2 + M^2 \log M]$ where

M = (Number of Range bins × Number of Doppler bins) and N is the number of Data Samples [92].

The ECA can be divided mainly into two phases:

1. Building of clutter subspace matrix $\boldsymbol{X}$, which consists mainly of three steps:

- Building the reference signal vector based on Equations (3.22), and (3.24).
- Building the reference matrix based on Equations (3.24), (3.29), and (3.30). It has extensive computation steps and requires computation of D^K.
- Building the clutter subspace matrix X based on Equations (3.26), (3.27), and (3.28). It requires $(2P + 2)$ times matrix multiplication.

2. Calculation of the multiple matrix product $X(X^H X)^{-1} X^H s_{surv}$ which consists mainly of four steps:

- Building the Hermitian transpose X^H of the complex matrix X.
- The calculation of complex matrix multiplication $(X^H X)$.
- The calculation of complex matrix inversion $(X^H X)^{-1}$.
- Finally, the calculation of complex matrix multiplication $X(X^H X)^{-1} X^H s_{surv}$.

3.4.4 Analysis and Optimization Techniques

The ECA in Passive Bistatic Radar (PBR) application has proven to be a very effective way to mitigate the direct signal, multipath and noise. Also, it provides accuracy for desired target detection. However, it is difficult to achieve the real- time requirements due to its extensive computation requirement. It is sometimes desirable to increase the number of range bins in order to increase the surveillance area which also increases computation time. So, our goal is to exploit any possible opportunity to minimize computation time and storage resources. The ECA has been modified by exploring opportunities of any computation and storage that could be eliminated. Parallel processing in both FPGA and GPU has also been used to achieve the real-time constraints. The following paragraphs describe some of the optimizations that have been incorporated.

3.4.4.1 Parallel implementation on FPGA.

FPGAs support high flexibility by providing different hardware resources to reduce the computation time. However, the ECA algorithm involves computation and storage of large matrices. It is not possible to accommodate all required storage and computation hardware resources on FPGA. Therefore, use of external memories can provide additional resources and alleviate the burden of storage requirements. Figure 3-20 shows the computational steps for implementation of ECA algorithm and its needed memory. It also shows data dependency between various operations and their computational sequence. It is proposed that the input should be stored in a DDR3 SDRAM.

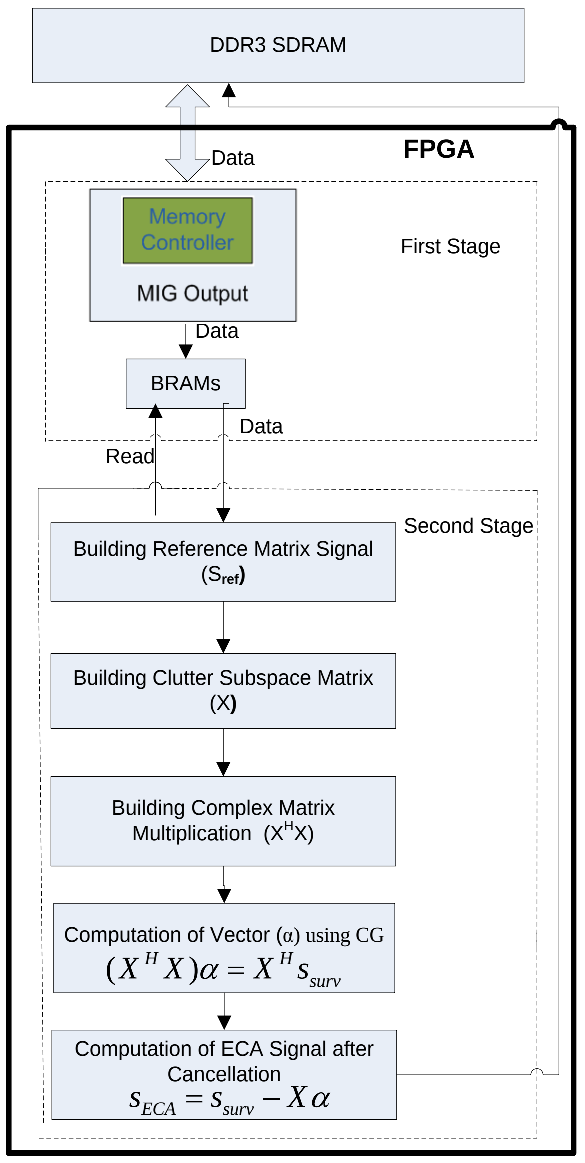

DDR3 SDRAM
Data
FPGA
Memory Controller
MIG Output
First Stage
Data
BRAMs
Data
Read
Second Stage
Building Reference Matrix Signal (S_{ref})
Building Clutter Subspace Matrix (X)
Building Complex Matrix Multiplication (X^HX)
Computation of Vector (α) using CG $(X^HX)\alpha = X^Hs_{surv}$
Computation of ECA Signal after Cancellation $s_{ECA} = s_{surv} - X\alpha$

Figure 3-20: ECA computational architecture

Firstly, the reference signal from the antenna is digitized and stored in FPGA memory instead of the external memory since it will be used many times to build the reference matrix in Equation (3.29). This step removes the communication overhead due to accessing of external memory and data transfer.

Matrices D, B and Λ_p have large dimensions of (N+R-1)*(N+R-1). They are involved in many matrix multiplications and constructions which consume large computation time and require high storage resources. However, these matrices can be called sparse matrices since they consist of few 1's and mostly zeroes that are used for shifting and construction purposes. Therefore, we have developed an efficient algorithm by eliminating all these sparse matrices. The new algorithm reduces the computation time and saves hardware resources.

The building of the reference matrix in Equation (3.29) requires extensive resources for storage and computation due to the following reasons:

- Building of (N+R-1)*(N+R-1) D matrix requires two for-loops with time complexity of O (N^2).
- Storage of D matrix and saving results of the D^K matrices.
- Computation of D^K power matrices requires time complexity of O ($N^2+N^3+2N^3+3N^3+\ldots..+ KN^3$).
- Multiplication of matrix D with the reference vector S_{ref} performing K-times matrix-vector multiplications.

However, the objective of matrix D is to shift the reference vector as shown in the Figure 3-21. Its sparse structure can be exploited to implement the reference matrix in Equation (3.29). This novel approach contributes in eliminating the construction of matrix D, computation of its power matrices and matrix-vector multiplication. Storage areas for storing powers of matrix are also eliminated. Following code is used to implement the reference matrix.

```
  Reference Matrix Implementation
Declare ⟵ index=1, j=0, Result [N+R-1] [K]={0};
For i ⟵ 1 to K {
  While (index < (N+R-1)) {
                Result [index][i] = Ref_vector[j];
  index = index+1; j=j+1;}
  index=i+1;j=0;}
```

$$\overset{D^K}{\begin{pmatrix} d_{11} & \cdots & d_{1n} \\ \vdots & \ddots & \vdots \\ d_{n1} & \cdots & d_{nn} \end{pmatrix}} \times \begin{bmatrix} sref_1 \\ sref_2 \\ sref_3 \\ \vdots \\ sref_{N-1} \\ sref_N \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ sref_1 \\ \vdots \\ sref_{N-K} \end{bmatrix}$$

Figure 3-21: Reference matrix implementation using the D matrix.

Our code for reference matrix implementation can be performed in parallel since there is no dependency. Such operations are accomplished by reordering and shifting of the reference vector. Loop unrolling and loop pipelining are two options that have been experimented via simulation of only this potentially parallelizable section. Performance parameters in terms of latency, memory usage and power dissipation are obtained as shown in Table 3.3. The loop pipelining technique is chosen as a tradeoff since it performs in parallel with the reuse of the same hardware resources across different stages. It can be seen from Table 3.3 the tradeoff between use of resources, power consumption and latency for different processing strategies. Our implementation goal is to achieve high speed with awareness of power, area and cost. Therefore, loop pipelining is selected to minimize high resource requirement, power and area.

Table 3.3: Latency and resource utilization of reference matrix implementation

	No Optimization	Loop Pipelining	Fully Parallel
Latency	16551	7940	3325
FF	57	49	206
LUT	56	120	9249
Power	9	15	944

Moreover, the reference matrix is stored in FPGA memory instead of external memory since it will be used many times to build the clutter subspace X in Equation (3.26). This will remove the communication overhead due to external memory data transfer.

Clutter subspace X needs the diagonal matrix Λ_p based on Equations (3.28) and (3.26). Matrix Λ_p has dimensions of (N+R-1)*(N+R-1) and is required (2P + 1) times. It needs to build matrices from – P to -1 and from 1 to P. Building of each matrix in Equation (3.28) requires two for-loops with time complexity of $O(N+R-1)^2$. So, the total time complexity is $O((2P+1)*(N+R-1)^2)$ and the total number of elements for storage will be $((2P+1)*(N+R-1)^2)$.

However, the matrix Λ_p is a diagonal matrix where all non-diagonal elements are zeroes. Hence, it simply requires the building of diagonal elements for each of its matrix. The diagonal elements have an exponential term which can be further exploited by simply changing sign values of its mirror values and eliminating computation of their magnitudes. This property reduces the computation time into half by changing the sign of the imaginary part of the second part without re-computing it. For example, the result of Λ_{-1} and Λ_1 is the same with the difference of only in the sign in its imaginary part.

However, the matrix Λ_p is used only for building the clutter subspace matrix **X**. So Λ_p can be directly incorporated into the clutter subspace matrix **X** without building all these (2P+1) matrices as shown in Figure 3-22. Following code is used to build the second term of Equation (3.26).

```
Declare ⟵ S_Ref [N+R-1][K], Input [N+R-1][M]={0};
For i ⟵0 to P {
int Dopp;
For j ⟵ 0 to N+R-1 {
For k ⟵ 0 to N+R-1 {
If (j==k) {
        Input[j][k+i*((N+R-1)+K)] = exp(2πj*Dopp/(k+1);}}}
Dopp++
For j ⟵ 0 to N+R-1 {
For k ⟵ 0 to K {
        Input[j][k+i*((N+R-1)+K)+(N+R-1)]=S_Ref[j][k];}}}
```

$$\left[\overset{\lambda_{-p}}{\begin{pmatrix}\lambda_{11} & \cdots & \lambda_{1n}\\ \vdots & \ddots & \vdots\\ \lambda_{n1} & \cdots & \lambda_{nn}\end{pmatrix}}\begin{pmatrix}Sref_{11} & \cdots & Sref_{1k}\\ \vdots & \ddots & \vdots\\ Sref_{n1} & \cdots & Sref_{nk}\end{pmatrix}\overset{\lambda_{-p+1}}{\begin{pmatrix}\lambda_{11} & \cdots & \lambda_{1n}\\ \vdots & \ddots & \vdots\\ \lambda_{n1} & \cdots & \lambda_{nn}\end{pmatrix}}\begin{pmatrix}Sref_{11} & \cdots & Sref_{1k}\\ \vdots & \ddots & \vdots\\ Sref_{n1} & \cdots & Sref_{nk}\end{pmatrix}\cdots\cdots\overset{\lambda_{p}}{\begin{pmatrix}\lambda_{11} & \cdots & \lambda_{1n}\\ \vdots & \ddots & \vdots\\ \lambda_{n1} & \cdots & \lambda_{nn}\end{pmatrix}}\begin{pmatrix}Sref_{11} & \cdots & Sref_{1k}\\ \vdots & \ddots & \vdots\\ Sref_{n1} & \cdots & Sref_{nk}\end{pmatrix}\right]$$

$$\Downarrow$$

$$\left[\overset{\lambda_{-p}}{\begin{pmatrix}\textit{Directly}\text{ builded}\\ \textit{through}\text{ its}\\ \text{Eqn. (8)}\end{pmatrix}}\begin{pmatrix}Sref_{11} & \cdots & Sref_{1k}\\ \vdots & \ddots & \vdots\\ Sref_{n1} & \cdots & Sref_{nk}\end{pmatrix}\overset{\lambda_{-p+1}}{\begin{pmatrix}\textit{Directly}\text{ builded}\\ \textit{through}\text{ its}\\ \text{Eqn. (8)}\end{pmatrix}}\begin{pmatrix}Sref_{11} & \cdots & Sref_{1k}\\ \vdots & \ddots & \vdots\\ Sref_{n1} & \cdots & Sref_{nk}\end{pmatrix}\cdots\cdots\overset{\lambda_{p}}{\begin{pmatrix}\textit{Directly}\text{ builded}\\ \textit{through}\text{ its}\\ \text{Eqn. (8)}\end{pmatrix}}\begin{pmatrix}Sref_{11} & \cdots & Sref_{1k}\\ \vdots & \ddots & \vdots\\ Sref_{n1} & \cdots & Sref_{nk}\end{pmatrix}\right]$$

Figure 3-22: Second part of clutter subspace matrix (**X**) implementation.

Our code for the second part of Equation (3.26) of clutter subspace **X** can be performed in parallel since there is no dependency. However, fully parallelization for this task causes excessive memory usage and high power dissipation as was seen in the case of the reference matrix implementation. Therefore, loop pipelining technique is also adopted.

Similarly, matrix B has a size of N *(N+R-1). The matrix B consists of few 1's and mostly zeroes. It pre-multiplies matrix X based on Equation (3.26). Building of matrix B and its matrix multiplication can be eliminated by simply using a shifting operation. The operation allows each column in the second term to be shifted up by (R-1) as shown in Figure 3-23.

$$\left[\begin{pmatrix} B_{11} & \cdots & B_{1n} \\ \vdots & \ddots & \vdots \\ B_{m1} & \cdots & \lambda_{mn} \end{pmatrix} \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} & \cdots & a_{1n} \\ a_{21} & a_{22} & a_{23} & a_{24} & \cdots & a_{2n} \\ a_{31} & a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \cdots & \cdots & \vdots \\ a_{n-11} & a_{n-12} & a_{n-13} & a_{n-14} & \cdots & a_{n-1n} \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{pmatrix} \begin{array}{c} \updownarrow \\ \text{R-1} \end{array} = \rightarrow \begin{bmatrix} a_{31} & a_{32} & a_{33} & a_{34} & \cdots & a_{3n} \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{n-11} & a_{n-12} & a_{n-13} & a_{n-14} & \cdots & a_{n-11} \\ a_{n1} & a_{n2} & a_{n3} & a_{n4} & \cdots & a_{nn} \end{bmatrix}\right]$$

Figure 3-23: Clutter subspace matrix (X) implementation using shifting operation.

However, building of matrix B and its matrix multiplication is eliminated by accessing the desired part of the matrix without performing any multiplication or shifting operations. This can be performed by accessing the array from the row at (R-1) until the last row of the matrix and ignoring the first R-1 rows as shown in Figure 3-24.

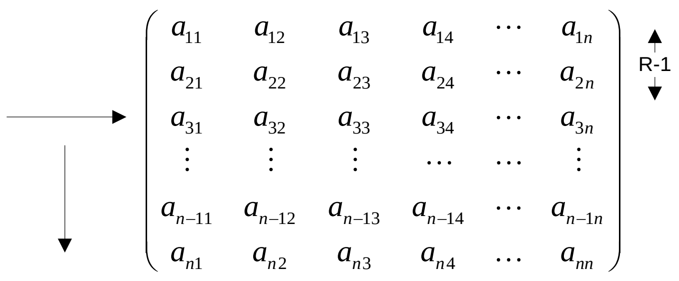


Figure 3-24: Clutter subspace matrix (**X**) implementation without multiplication or shifting operation.

Reorganization of clutter matrix X computation and removal of matrices B, D and Λ_p computation has resulted in reduced computation time, storage, and power dissipation.

ECA also requires trigonometric and other mathematical functions which are not supported by HLS tool. Therefore, customized functions have been developed. It is well known that calling of function incurs overhead. Therefore, an in-lining technique is applied for all functions to minimize the overhead.

The complex matrix multiplication ($X^H X$) in the second phase requires the building of the Hermitian transpose matrix X^H. However, this can be achieved by only changing the indices of the matrix **X** without performing its transpose and conjugate thus reducing the total computation time and the hardware storage requirements as shown in the following code:

```
 Function ($X^H X$)
Declare ⟵ X [] [], Result [] [], Sum;
 For k⟵ 0 to M {
For i ⟵0 to M {
For j⟵0 to N
            {Sum = Sum + X [j] [k] * X [j] [i];}
Result [k] [i] = Sum;
Sum = 0;}}
```

The second step of the second phase involves complex matrix multiplication. A loop pipelining approach is used where the multiplication operation is divided into four stages as shown in Figure 3-25. This technique allows the operations in the loop to be overlapped and executed in a concurrent manner instead of the sequential execution.

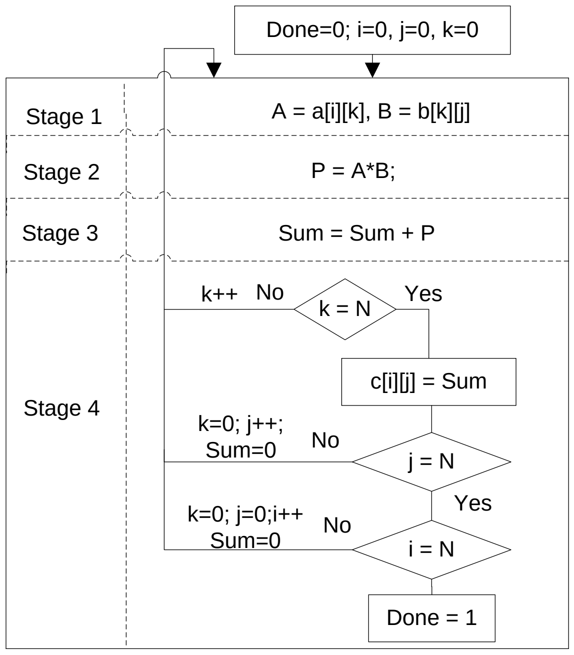


Figure 3-25: Flow chart for pipelined matrix multiplication.

It's also important to use the memory resources efficiently besides reducing the number of arithmetic operations. ECA requires large matrix multiplications where FPGA storage can't hold such large data. So, storing the matrices row by row or column by column increases the number of times of loading the data from external memory to the FPGA. Therefore, only part of the matrix is brought into FPGA which requires replacing previously stored data. However, this process is accomplished via blocking approach where the matrices are divided into a number of sub-matrices that fit into the size of FPGA storage as shown in Figure 3-26.

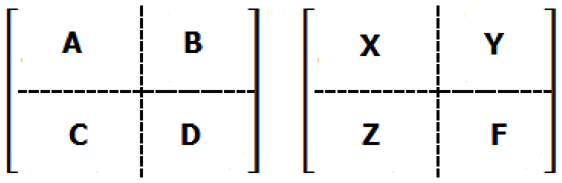


Figure 3-26: Blocking Technique

Also, the code has been modified to access the data efficiently instead of the traditional accessing. Table 3.4 shows the multiplication of blocks A and B of the first matrix with the second matrix using both our modified approach and the traditional approach. Our method loads each block (A & B) only once while the traditional method loads each block two times. This will be very beneficial when the matrices have large dimensions as in our case. So, the data in the FPGA storage can be reused before being replaced which reduces the number of loading times by a factor of N, where N is the blocking factor.

Table 3.4: Our matrix multiplication method versus traditional method

Traditional Method	Our Method
AX	AX
BZ	AY
AY	BZ
BF	BF

Following code is used to build the blocking technique:

Blocked Matrix Multiply

```
Declare ⟵ X [] [], Result [] [], Sum;
For k⟵ 0 to M-1 {
    For i ⟵ 0 to M-1 {
        {read block C(k,i) into FPGA memory}
```

```
For j ⟵ 0 to N-1 {
{read block from  X into FPGA memory}
   C(k,i)=C(k,i)+ X [j] [k] * X [j] [i];} {do a matrix multiply on blocks}
       {write block C(i,j) back to memory}
```

The computation of vector (α) in Equation (3.31) requires the following computation and storages:

- Computation of matrix inversion $Y = (X^H X)^{-1}$
- Matrix multiplication $Z = (YX^H)$
- Matrix-vector multiplication $\alpha = (Zs_{surv})$
- Storage for Y and Z

However, the structure of Equation (3.31) can be exploited to perform the computation of vector (α) in an efficient way using Conjugate Gradient (CG) as in Section 4.1.5. This eliminates matrix to matrix multiplication and its storage requirements. Computation process is re-arranged and following operations are performed:

- Matrix-vector multiplication $z = (X^H s_{surv})$
- The conjugate gradient method is used to perform $(F\alpha = z)$ where $F = (X^H X)$

Full parallelization of CG can't be achieved due to the data dependencies as the next computational step depends on the result of the current step. However, the CG is divided into five computational regions as shown in Figure 3-27 where each region can be performed in parallel.

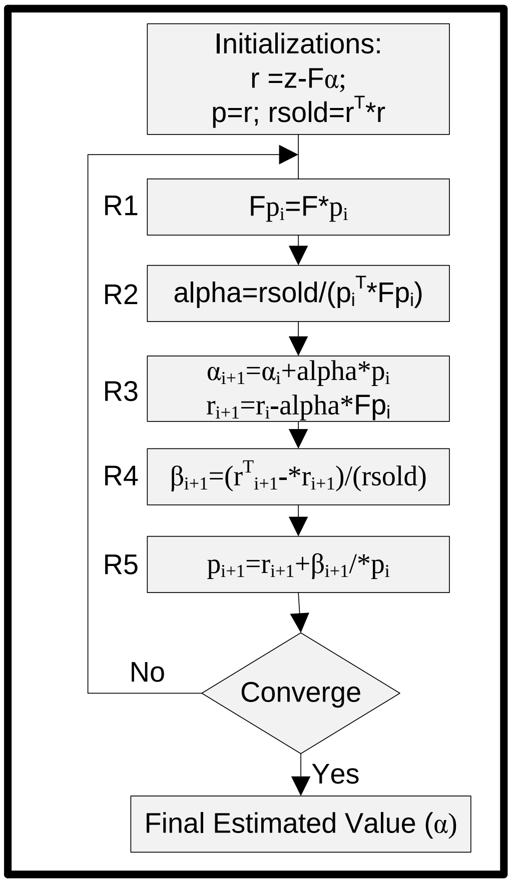


Figure 3-27: Conjugate Gradient (CG) computation steps.

Moreover, Dataflow technique is applied between the regions as in Section 4.1.5. Also, matrix-vector multiplication process is used many times in our work where it can be performed in the same way as in Section 4.1.5 where loop pipelining technique of the outer loop is chosen.

Computation of the surveillance signal after cancellation based on Equation (3.32) requires two operations:

- Calculation of matrix-vector multiplication ($X\alpha$) and a vector to store its result.
- Calculation of vector-vector subtraction and a vector to store its result.

Equation (3.32) was implemented using two methods where each method has a trade-off of speed, area, and power consumption. The first method minimizes the latency as much as possible by performing matrix-vector multiplication and vector-vector subtraction operations in parallel where a dataflow technique is applied between these operations. This is because it is not necessary for vector-vector subtraction operation to wait until matrix-vector multiplication operation completes all its iterations. However, this method requires additional vector to store its result which increases the area and power consumption. On the other hand, the second method performs matrix-vector multiplication in parallel without applying dataflow technique. It reduces the storage and the power dissipation by removing the additional vector to save the result of matrix-vector multiplication as shown in the following code:

Function ($S_{ECA} = S_{sur} - X\alpha$)

```
For i ⟵ 0 to N {
For j ⟵ 0 to M {
            {Sum = Sum + X [j] [i] * α [j];}
    S_ECA [i] = S_sur[i]-Sum; Sum = 0;}}
```

The first option is selected to achieve high speed. The loop pipelining is applied to the outer loop of the matrix-vector multiplication as it's achieved the minimum latency with awareness of resource requirement, power, and area. The number of clock cycles for matrix-vector multiplication and vector-vector subtraction for N=M=20 is 219 and 22 clock cycles respectively as shown in Table 3.5. Table 3.5 shows that the latency of vector-vector subtraction is completely removed using the dataflow technique.

Table 3.5: Latency and resource utilization of second option for m=20 on Artix7 (XA7A100T CSG324) -1q

	No Optimization	Matrix-vector Mult. Only	Vector-vector Subt. Only	Pipelining without Dataflow	Pipelining with Dataflow
Latency (cycles)	3661	219	22	240	219
Clock period (ns)	7.19	7.98	7.19	7.98	7.98
FF	224	2878	15	2891	2899
LUT	213	2321	52	2371	2379
DSP48E	4	80	0	80	80
BRAM_18K	1	1	1	1	2
Power	42	527	6	534	532

3.4.4.2 Parallel implementation on GPU.

Extensive Cancellation Algorithm (ECA) is also implemented and parallelized on GPU architecture to minimize its extensive computation. GPU architecture is massively parallel since it has hundreds of cores running in a concurrent manner, cheap, and highly available. Therefore, in this work we design and implement ECA under CUDA architecture on the GPU.

The first step in the algorithm builds the reference signal vector based on Equations (3.22) and (3.24). Since the samples of the signal from the antennas are received in serial, so the reference signal vector is built in serial on the CPU part.

Our code for reference matrix implementation in Figure 3-21 can be parallelized since there is no dependency. Each thread must work on a group of elements because the creation of the threads adds an overhead more than the task itself as our task is only copy elements from vector to matrix. This will eliminate the index calculation cost over these elements. However, the for-loop in the code can be fully parallelized using column wise where each thread takes one value of counter i. So, each thread is writing its own index to the corresponding location of index and j indices without changing anything in the code. However, the number of elements in each column is (N+R-1) whereas K elements in each row. So, it's desirable to parallelize using rows wise to get more benefits since each row has a few elements. However, the number of elements to be assigned for each row is different. Therefore, our code has been changed to match the threads structure in GPU as shown in the following code:

Reference Matrix Implementation

```
int index ⟵ Thread index; j=0; i=index;
  While (i>= 0) {
        Result [index][j] = Ref_vector[i];
  i=i-1; j=j+1;}
```

Figure 3-28 shows the used parallelization technique of the reference matrix. Each thread copies only a few numbers of elements such that the first thread copies only one element, second thread copies two elements, and thread N+R-1 copies 32 elements.

$$\begin{array}{l} \text{Thread 1} \longrightarrow \\ \text{Thread 2} \longrightarrow \\ \text{Thread 3} \longrightarrow \\ \vdots \\ \text{Thread N+R-1} \longrightarrow \end{array} \begin{pmatrix} a_{1,1} & 0 & 0 & 0 & \cdots & 0 \\ a_{2,1} & a_{2,2} & 0 & 0 & \cdots & 0 \\ a_{3,1} & a_{3,2} & a_{3,3} & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \cdots & \cdots & \vdots \\ a_{N+R-1,1} & a_{N+R-1,2} & \cdots & \cdots & \cdots & a_{N+R-1,32} \end{pmatrix}$$

Figure 3-28: Parallelization method of the reference matrix.

Building the second term of Equation (3.26) requires copying of Λ and reference matrices (2P+1) times. Λ matrix has large dimensions of (N+R-1) (N+R-1). Therefore, performing parallelization using row wise or column wise is not the optimal parallelization method. Each thread needs to copy large number of elements while leaving other threads idle. Therefore, this process is parallelized via blocking technique where the matrix is divided into a number of blocks where each thread copies a few elements (block) of the matrix each with size 16*16 elements as shown in Figure 3-29.

$$\begin{bmatrix} \overset{\text{Block 00}}{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}} & \cdots\cdots & \overset{\text{Block 0N}}{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}} \\ \vdots & \ddots & \vdots \\ \vdots & \ddots & \vdots \\ \overset{\text{Block N0}}{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}} & \cdots\cdots & \overset{\text{Block NN}}{\begin{pmatrix} a_{11} & \cdots & a_{1n} \\ \vdots & \ddots & \vdots \\ a_{m1} & \cdots & a_{mn} \end{pmatrix}} \end{bmatrix}$$

Figure 3-29: Parallelization method via blocking technique.

The pseudo code of copying Λ matrix is shown below:

Copy Λ Matrix Parallelization Method

```
int c = blockDim.x * blockId.x + threadIdx.x;
int r = blockDim.y * blockId.y + threadIdx.y;
x= c*Tile_Size;
y= r*Tile_Size;
For i ⟵ 0 to Tile_Size {
For j ⟵ 0 to Tile_Size {
If (i==j) {  Input[x+i] [y+j]=exp(2π(x+i)*Dopp/( y+j +1);}}}
```

Copying of reference matrix is accomplished row by row since each row has a few elements (K elements). So, each thread copies few elements of reference matrix such that the index calculation cost is amortized over these elements.

The complex matrix multiplication (X^HX) requires $O(NM^2)$ operations. The multiplication can be performed in parallel since there is no data dependency. However, this requires huge matrices multiplication which is parallelized by combining row wise technique with blocking technique. The number of rows of Hermitian matrix (X^H) is more than the number of cores in the GPU. So, the rows are divided evenly on the number of cores by the following relation:

$$Number\ of\ rows\ for\ each\ core = \frac{(\mathrm{N+R-1})*2\mathrm{P}+(\mathrm{K})*2\mathrm{P}}{number\ of\ cores} \tag{3.34}$$

So each core works on a group of rows as shown in Figure 3-30.

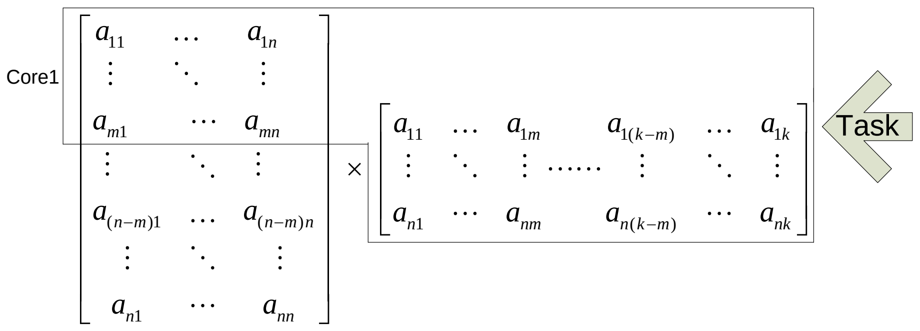


Figure 3-30: Core task of matrix multiplication

Blocking technique is also used since each core has eight blocks and each block has 512 threads. So, each thread will multiply one row from the first matrix with a group of columns in the second matrix as shown in Figure 3-31 based on the following relation:

$$Number\ of\ columns\ for\ each\ thread = \frac{\left[\left((N+R-1)+K\right)*(2P)\right]^2}{\#\ of\ cores*\ \#\ of\ blocks*\#\ of\ threads} \qquad (3.35)$$

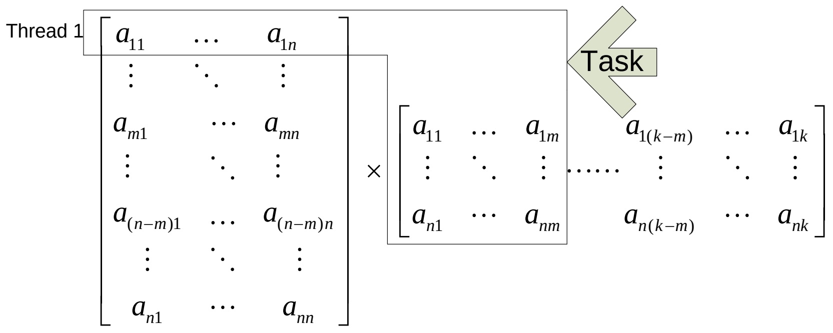


Figure 3-31: Thread task of matrix multiplication

Following paragraphs describe the parallelization method for each region in CG method. Region 1 requires matrix-vector multiplication where the row wise technique has been used since the number of rows of the matrix more than the number of the available threads. So, the rows are divided evenly on the number of cores and each thread multiplies one row by the vector as shown in Figure 3-32 based on the following code:

Matrix-Vector Parallelization Method

```
int index ⟵ Thread index;
Tile_Size=Matrix_Size/# of threads.
x=index*Tile_Size.
y=(index+1)*Tile_Size.
For i ⟵ x to y {
    Vec[i]=0;
    For j ⟵ 0 to M {
        Vec[i] + = F[i][j]*p[j];}}
```

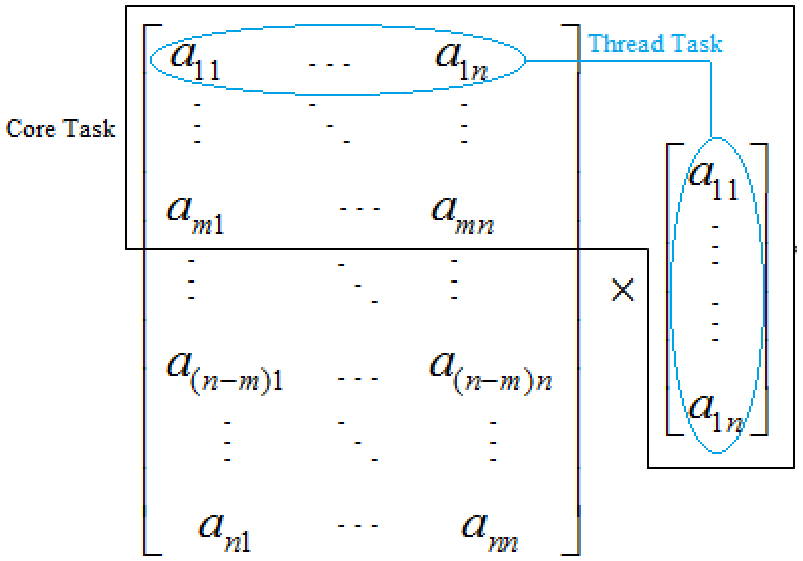


Figure 3-32: Parallelization technique of matrix-vector multiplication

Computation of the surveillance signal after cancellation based on Equation (3.31) requires also matrix-vector multiplication ($X\alpha$) which is performed in the same way.

Regions 3 and 5 require vector-vector addition. The same parallelization method of matrix-vector is applied where the vectors are divided into blocks. Each thread performs vector-vector addition to its own block as shown in Figure 3-33. The same method is applied in Equation (3.31) that requires vector-vector subtraction ($s_{ECA} = s_{surv} - X\alpha$).

Vector-Vector Addition Parallelization Method

```
int index ⟵ Thread index;
Tile_Size=Vector_Size/# of threads.
x=index*Tile_Size.
y =(index+1)*Tile_Size.
For i ⟵ x to y {
Vec[i] = V1[i]+V2[i];}
```

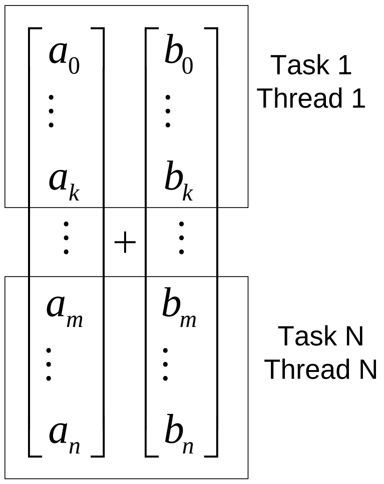


Figure 3-33: Vectors addition via blocking technique.

Regions 2 and 4 require vector-vector multiplication. Since these vectors have large dimension. The vector is divided into the available threads as:

$$\text{Tile\_Size} = \text{Vector\_Size}/\#\text{ of threads} \qquad (3.36)$$

Each thread multiplies a group of elements and save a result into a private variable. However, synchronization scheme is required since an accumulation of these private variables must be performed for all threads results in the final step.

3.5 Parallelization of Block Compressive Sampling Matching Pursuit Algorithm (BCoSaMP)

3.5.1 Introduction

With the advance technology nowadays, most of the systems contain a large number of sensors to increase the measurement accuracy and resolution. However, this large deployment of sensors increases the number of received samples which needs high computation time and memory to store it. It will be practically impossible to store the entire signal into a processing buffer at a time as encountered in streaming applications. For example, sampling at a GHz with 16 bits per sample requires 16 billion samples per second, which needs a lot of computing and storage resources to process these large numbers of samples. However, most of these signals are sparse since they consist of few coefficients and mostly zeroes. So, an efficient method is required to reduce storage and simply keeps only the largest coefficients.

Compressive sensing techniques allow sampling of signal at lower than the Nyquist rate [104-105] and storage of small number of samples. Compressive sensing techniques keep only the largest coefficient while the small coefficients are discarded to reduce the amount of data required to be stored, processed, and transmitted. Compressive sensing techniques proved that the sparse signal can be reconstructed from few incoherent measurements [106].

The major challenge in the compressive sensing strategies is the way to approximate the signal accurately and efficiently from noisy samples. Many algorithms have been developed for that purpose. These algorithms can be divided into three categories [107]:

- Greedy pursuit algorithms: these techniques approximate the signal by applying an iterative method. It's based on the selection of the optimal choice at each step in their process where the approximation of the signal performs one step at a time. Some of the greedy methods are Orthogonal Matching Pursuit (OMP) [108], stagewise OMP (StOMP) [109] and Regularized OMP (ROMP) [110].
- Convex relaxation algorithms: these techniques are developed based on convex method. It's a subfield of the optimization methods to approximate the target signal. It has a wide range of applications in different fields. Some of the convex relaxation methods are interior-point methods [111], projected gradient methods [112], and iterative thresholding [113].
- Combinatorial algorithms: these techniques require acquiring of a large number of samples of the signal. Acquired samples must be structured in a way that supports fast reconstruction via group testing [100]. Some of these techniques are Fourier sampling [114], and the chaining pursuit [115].

Many of the combinatorial techniques require a large number of unusual samples which are difficult to acquire [107]. Convex relaxation techniques are computationally intensive. On the other hand, greedy pursuits techniques are intermediate in their running time and sampling efficiency. The accuracy of these algorithms depends on the approximation of the collected samples and the sparsity of the original signal. The accuracy of the reconstruction is validated by comparing the signal sparse approximation with largest coefficients in the sparsity basis.

The signals in reality have some sort of noise. Many methods [116-121] have been developed to reconstruct the noisy sparse signal accurately. However, some of these techniques don't give a good performance and some of them assume that the noise is either bounded or Gaussian with known variance. Block Compressive Sampling Matching Pursuit algorithm (BCoSaMP) [107, 122, 123-126] is one of these algorithms that has been chosen in our research since it proves high robustness across many applications. BCoSaMP algorithm can represent the entire signal efficiently from only small number of noisy measurements. It uses information about the noise magnitude for stopping criterion rather than assuming that the noise is Gaussian or bounded. Also, it proves that the approximation error decays exponentially at every iteration. Therefore a terminating criterion is needed when an appropriate threshold has been achieved [122]. BCoSaMP algorithm reduces the number of measurements by exploiting the sparsity and compressibility features without sacrificing the robustness. It offers the following advantages:

- It reduces the computational complexity.
- It achieves good accuracy with minimum number of samples even in the presence of noise.
- It supports small error for every target signal.
- It supports efficient resource usage.

3.5.2 Block Compressive Sampling Matching Pursuit algorithm (BCoSaMP)

The collected samples are arranged in a matrix called sampling matrix (φ). BCoSaMP requires that the signal must have the sparsity structure which can be preserved by applying Restricted Isometry Property (RIP) [127]. For a large number of samples N, there is only m samples have nonzero such that m<<N which needs to be stored or transmitted. So, the sampling matrix has a dimensions of (m x N) and the observed signal can be represented as:

$$y = \varphi * x \tag{3.37}$$

Where $x \in R^N$ is the signal to be recoveond, $y \in R^m$ is the observation at the current state (the vector of samples), φ is the sampling matrix with dimensions $m \times N$.

The set of indices of all nonzero entries called the support of x (sup(x)). It can be represented by the following relation [107]:

$$\left|\sup(x)\right| = \left|\left\{ j : x_j \neq 0 \right\}\right| \tag{3.38}$$

There are two critical conditions that must be preserved to recover the signal accurately: the signal must be sparse signal and the sampling matrix must satisfy the RIP. The Restricted Isometry Property (RIP) of φ is given as:

$$(1-\delta_r)\|x\|_2^2 \leq \|\varphi x\|_2^2 \leq (1+\delta_r)\|x\|_2^2 \tag{3.39}$$

where δ_r is the least number of the sampling matrix that satisfies the RIP property.

BCoSaMP algorithm recovers and reconstructs the signal by applying pseudo inverse where the signal reconstruction is based on how accurate the signal is approximated from the available samples [128-129]. Exploiting the dependencies between values and locations of the signal coefficients is very important beside the sparsity and compressibility advantages. This is performed by partitioning the signal into multiple blocks. So, BCoSaMP divides the signal X into K blocks and each block has n elements as shown in Equation (3.40). This substantially decreases the number of measurements without sacrificing robustness. Exploiting both these properties will help in compressing the signal to the lowest possible level instead of dealing with every large coefficient independently.

$$X = \{x_1 \cdots x_n; x_{n+1} \cdots x_{2n}; \cdots\cdots; x_{(k-1)n} \cdots x_{kn}\} \tag{3.40}$$

To divide the signal into blocks, we assume a signal vector X ∈ R^{nK}, with n and K integers. This signal can be re-shaped into a n × K matrix X. Each column of X will be considered as a part of the signal. That is, signals X in a block-sparse model have entire columns as zeros or nonzeros. The measure of sparsity for X is the number of nonzero columns [130]. Common assumptions in the compressive sampling algorithms are:

- The sparsity level s is fixed.
- Sampling matrix obey Restricted Isometry Property (RIP).
- The vector of samples $y = \Phi x + e$.

Following is the summary of the BCoSaMP algorithms:

Input: Sampling matrix (φ), noisy sample vector y, K is the number of recovered samples.

Output: K-sparse approximation for recovering signal **Error! Bookmark not defined.**from original signal x.

- Initialize: $\hat{x}_0 = 0$, r =y
- Compute the residual signal by creating the state of proxy as $e = \varphi^T r$.
- Find the largest components of the proxy and store its index $\Omega = \mathrm{supp}(\mathrm{e}, \mathrm{K})$.
- Merge Ω with x_k to obtain $T = (\Omega \cup \mathrm{supp}(\hat{x}))$.
- Perform pseudo inverse $(b|_T = Pseudo\ inv(\varphi) * y)$.
- Obtain the estimated value $\hat{x} = (b, K)$.
- Update the residue value as $r = y - \varphi\hat{x}$.
- Repeat Steps 2 to 7 until the required criteria of the residue is obtained.
- Return $\hat{x}$.

3.5.3 Analysis and Optimization Techniques

BCoSaMP has proven to be an effective method to reconstruct the sparse signal from small noisy measurements. However, it may not achieve the real time requirements due to extensive computations. So, our goal is to exploit any opportunity of parallelism to minimize the computation time and storage resources to achieve the real time constraints.

3.5.3.1 Parallel implementation on FPGA.

Figure 3-34 shows the computational steps for implementation of BCoSaMP algorithm and its needed memory. It also shows data dependency between various operations and their computational sequence. It is proposed that the input should be stored in a DDR3 SDRAM.

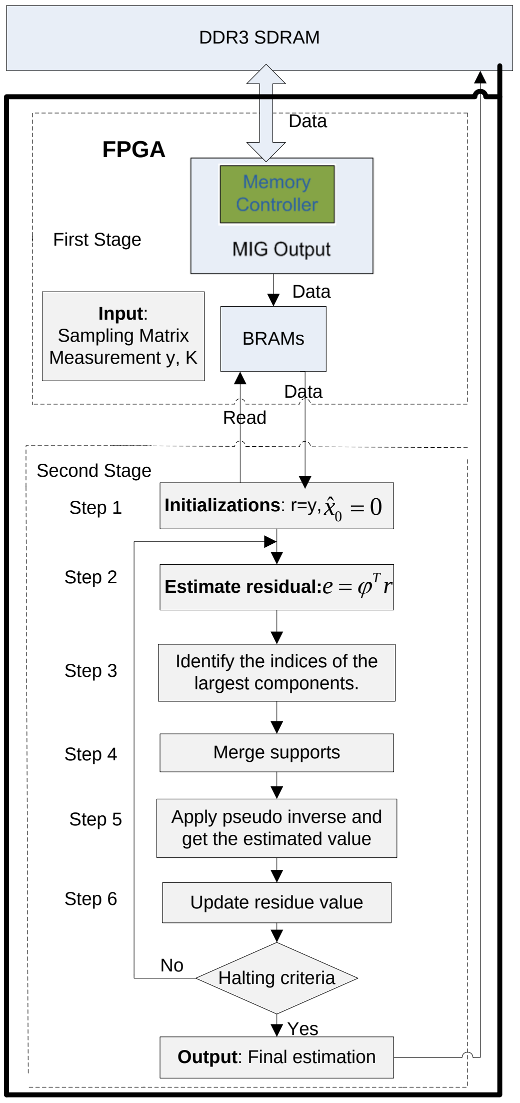

DDR3 SDRAM
Data
FPGA
Memory Controller
First Stage
MIG Output
Data
Input: Sampling Matrix Measurement y, K
BRAMs
Data
Read
Second Stage
Step 1
Initializations: r=y, $\hat{x}_0 = 0$
Step 2
Estimate residual: $e = \varphi^T r$
Step 3
Identify the indices of the largest components.
Step 4
Merge supports
Step 5
Apply pseudo inverse and get the estimated value
Step 6
Update residue value
No
Halting criteria
Yes
Output: Final estimation

Figure 3-34: BCoSaMP computational architecture

A. *Transformation and optimization of step two (residual estimation)*

Step two creates the proxy which requires the cost of matrix Hermitian (φ^H) and matrix-vector multiplication ($e = \varphi^T r$). However, the multiplication of this step can be accomplished without performing Hermitian by changing only the indices reducing the total computation time and the hardware storage since the building of Hermitian requires additional matrix to store the result as shown in the following pseudo code:

```
  **Function** $(X^H r)$
**Declare** $\leftarrow$ X [] [], Result [], Sum;
**For** i $\leftarrow$ 0 to N {
    **For** j $\leftarrow$ 0 to M {
        Sum = Sum + X [j] [i] * r [j];}
    Result [i] = Sum; Sum = 0;}
```

Moreover, matrix-vector multiplication can be performed in parallel by applying the pipelining technique to the outer loop as it’s demonstrated in Section 4.1. Step three requires locating the largest values of a vector and stores the indices of those values. There are a number of sorting algorithms such as quick sort, merge sort, bubble sort etc. available in the literature [131]. They can be used to sort entries of the signal in decreasing order of magnitude and then selects the largest values of them. However, these sorting algorithms have many drawbacks and require:

- Allocation of a new array of size (N).
- Copy of the original array to the new one with a cost of O(N).
- Sorting the new array with a cost of (N logN).k
- Iterating over the original array and searching for the largest elements with a cost of (N logN).

Also, the sorting algorithm must be stable [107]. However, the objective of this task is to find only the indices of the largest elements without sorting the array. So, it’s desirable to implement a new efficient code for this task as follows:

Largest K indices of a Vector

```
Declare ⟵ Input [N], indices [K], value [K];
For i ⟵ 0 to N {
For j ⟵ 0 to K {
if (indices[j] == 0 && value[j]==0)
        {indices[j]=i; value[j]=Input[i]; break;}
else if(value[j] < Input[i]){
          For k ⟵ K-1 to j
      {indices[k]=indices[k-1]; value[k]=value[k-1];}
        indices[j]=i; value[j]=Input[i]; break;}}}
```

Our new code allocates two small arrays, each with size K instead of one large array with size N. It also eliminates the copying of the original array. However, it requires O(K^2N) in the worst case. It uses break statements whenever any condition fails then it will terminate the execution of the inner loop. This mechanism contributes in faster computations of this part. This approach will be more efficient as K is a small value and it is the same as used in the BCoSaMP algorithm. K represents only the largest values in the input sparse signal.

B. *Transformation and optimization of step four (merging support)*

Step four requires merging of two sets of data which can be performed by creating following new efficient code:

```
Declare ⟵ c[N+M], flag; k=M;
%M number of elements in vector a
%N number of elements in vector b
Loop1: For i ⟵ 0 to M
  {c[i]=a[i]; }
Loop2: For i ⟵ 0 to N
            {flag=0;
              For j ⟵ 0 to M
 {if(b[i]==c[j]) { flag=1; break; }}
  if(flag==0) {c[k]=b[i]; k++; } }
```

The code consists of two major loops; Loop1 and Loop2. Loop1 can be fully parallelized by applying fully loop unrolling technique which will be executed in one clock cycle instead of M clock cycles. However, Loop2 can't be executed in parallel since there is a dependency between store (write) operation (**c**[k]=**b**[i]) and load (read) operation (**b**[i]==**c**[j]). The load operation needs to wait until the store operation is completed. However, pipelining technique is an effective method for elimination of this type of dependency. Pipelining inherently inserts delays and there will be no conflict in getting the correct value. So, pipelining technique is applied and the latency is improved as shown in Table 3.6.

Table 3.6: Latency and resource utilization of merging operation for N=50 and m=25 on Artix7 (XA7A100T CSG324 -1q)

	No Optimization	Optimization with both unrolling and pipelining techniques
Latency (cycles)	4002	1315
Clock period (ns)	4.87	4.87
FF	83	87
LUT	139	314
Power	21	39

C. *Transformation and optimization of step five (pseudo inverse)*

The computation of step five requires following computation and storages:

- Computation of matrix inversion $X = (\Phi)^{-1}$.
- Matrix-vector multiplication $b = X * y$.
- Storage for X and b.

However, the computation of vector (**b**) can be performed in an efficient way using the conjugate gradient method as it can deal with different dimensions as sampling matrix (φ) has different dimensions where many other inversion algorithms require a square matrix. It also eliminates storing of an inverted matrix (**X**). CG is divided into five computational regions and the parallelization methods are applied as demonstrated in Section 4.1.

D. Transformation and optimization of step six (residual update)

Step six updates the residue value for the next round which requires two operations; calculation of matrix-vector multiplication ($\varphi\hat{x}$) and storage of a resulting vector, and calculation of vector-vector subtraction and storage of the resulting vector. This step was implemented using two methods as demonstrated in Section 4.2 where the pipelining technique with the dataflow approach is applied.

3.5.3.2 Parallel implementation on GPU.

Step two in Figure 3-34 can be fully parallelized where each thread must work on a group of elements because the creation of the threads adds an overhead more than the task itself. This will eliminate the index calculation cost over these elements. However, the number of elements (N) in each row is higher than the number of elements (M) in each column (M<<N). So, it’s desirable to parallelize it using column wise. This approach will create a higher number of threads (N) that will operate on fewer elements (M) as can be seen in the following code:

Parallel Matrix-Vector Multiplication

```
int index ⟵Thread index;
For j⟵ 0 to M
{ Res[index] + =X[j][index]* r[j];}
```

Step four requires merging of two sets of data which consists of two major loops; Loop1 and Loop2. Loop1 can be fully parallelized using M threads that are executed in one clock cycle instead of M clock cycles. However, a synchronization scheme is required after the statement of Loop1 to insure that all the elements of vector **a** is copied to the vector c before executing Loop2. Loop2 is parallelized where each thread only compares one element value in vector **b** with the element values in vector **c** instead of comparing all the values in vector **b**. This will improve the performance by a factor of P, where P is the number of elements in vector **b**. However, Loop2 requires a synchronization scheme since the merged output vector is shared between all the threads to get correct results as follows:

```
Declare⟵  c[k], flag; k=M;
int index⟵Thread index;
Loop1: c[index]=a[index];
__syncthreads();
flag=0;
For j⟵  0 to M
{ if(b[index]==c[j]) { flag=1; break; }}
if(flag==0) {
__syncthreads();
c[k]=b[index]; k++;
Unlock_suncronization();}
```

The parallelization method for each region in CG method is accomplished using the same way in Section 4.2. Step six updates the residue value for the next round which requires two operations: matrix-vector multiplication ($\varphi\hat{x}$) and vector-vector subtraction. Matrix vector multiplication can be parallelized using the same technique in step two and vector-vector subtraction is accomplished by N threads where each thread subtracts one subtraction operation.

3.5.4 Model-Based Iterative Hard Thresholding (MB-IHT) Method

Iterative Hard Thresholding (IHT) method has also proven to be an effective method to reconstruct the sparse signal from noisy measurements and small number of samples beside its high robustness. So, efficient implementation of MB-IHT on FPGA has been proposed for high performance applications.

Iterative Hard Thresholding (IHT) method is one of the methods that can represent the entire signal efficiently from only a small number of received samples. It reduces the number of measurements by exploiting the sparsity and compressibility features without sacrificing the robustness. It offers the following advantages [132-134]:

- Guarantees very small error.
- Proves robustness to the observation noise.
- Achieves good accuracy with a minimum number of observations.
- Supports many sampling operators.
- Supports efficiency in the hardware resources usage.
- Achieves good signal to noise ratio using fixed number of iterations.

In order to represent the concept in a mathematical relation, the observed signal can be represented as:

$$y = \varphi x + e \tag{3.41}$$

Where $x \in R^N$ is the signal to be recoveond, $y \in R^m$ is the observation at the current state (the vector of samples), φ is the sampling matrix with dimensions $m \times N$, and e is observation noise.

The IHT algorithm uses an iterative method to recover x signal from the given observed signal (y) and sampling matrix (φ) based on the following relation:

$$x^{n+1} = L_K(x^n + \varphi^T(y - \varphi x^n)) \tag{3.42}$$

where L_K represents the largest K elements of a vector.

Moreover, Richard in [130] made some improvements of IHT method to provide more accuracy and call it Model-Based Iterative Hard Thresholding (MB-IHT) which can be summarized by the following steps:

Input: Sampling matrix (φ), measurement y, K

Output: K-sparse approximation $\hat{x}$ to signal x.

1. Initialization: $\hat{x}_0 = 0$, r =y
2. Form the signal estimation as $b = \hat{x} + \varphi^T r$.
3. Find the largest K components and store the index of that. $\hat{x} = \sup(b, K)$.
4. Update the residue value for the next round as $r = y - \varphi\hat{x}$.
5. Repeat step 2 to 4 until the required criteria of the residue is obtained.
6. Return $\hat{x}$.

3.5.4.1 Parallel implementation on FPGA

Figure 3-35 shows the computational steps of MB-IHT implementation and its needed memory. It also shows data dependency between various operations and their computational sequence.

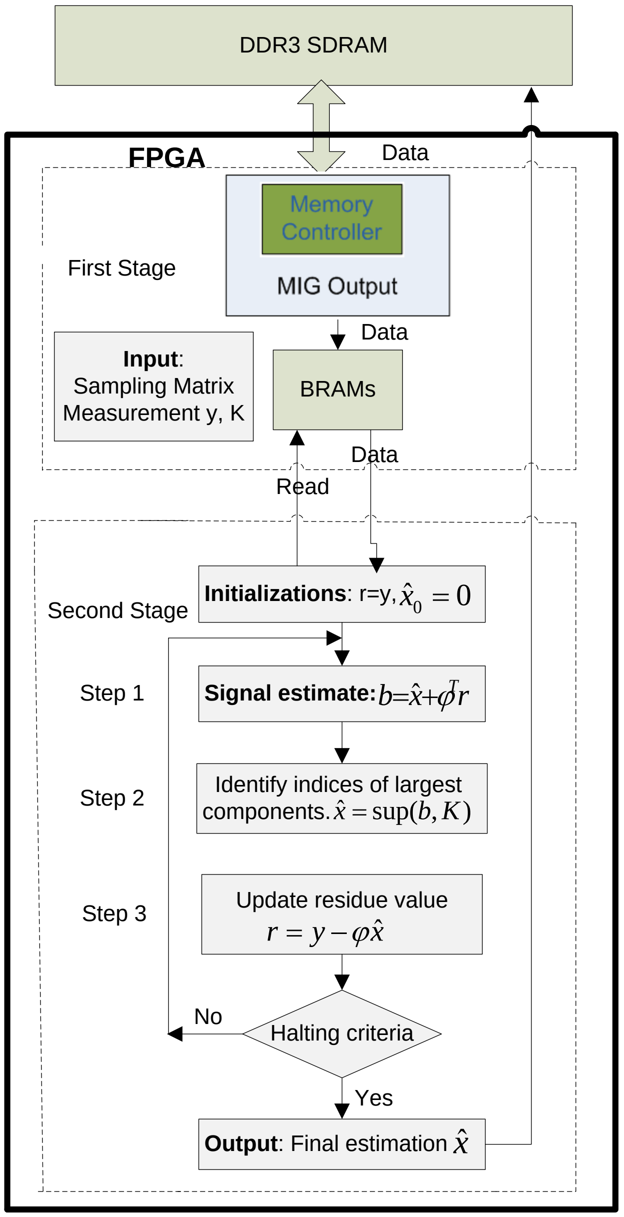

DDR3 SDRAM
FPGA
Data
Memory Controller
MIG Output
First Stage
Data
Input: Sampling Matrix Measurement y, K
BRAMs
Data
Read
Second Stage
Initializations: r=y, $\hat{x}_0 = 0$
Step 1
Signal estimate: $b = \hat{x} + \phi^T r$
Step 2
Identify indices of largest components. $\hat{x} = \sup(b, K)$
Step 3
Update residue value $r = y - \varphi\hat{x}$
No
Halting criteria
Yes
Output: Final estimation $\hat{x}$

Figure 3-35: MB-IHT computational architecture

Each step of the MB - IHT method is analyzed and parallelized to exploit the advantages of both the parallel processing platform and the inherent parallelism of the MB - IHT method. Step one forms the signal estimation requiring the following three operations:

- Calculation of matrix Hermitian (φ^H) which needs to do conjugate and transpose and additional matrix to store the result.
- Calculation of matrix-vector multiplication ($\varphi^T r$) and requires additional vector to store its result.
- Calculation of vector-vector addition and requires additional vector to store its result.

The efficient implementation of this step can be accomplished by combining the optimization techniques in Steps two and six in Section 3.5.3. It exploits the advantages of removing Hermitian operation by changing only in the indices as in Step two and pipelining technique with dataflow technique for matrix-vector multiplication and vector-vector addition operations as in Step six. So, both methods with and without dataflow techniques are applied and simulated for large dimensions to show the obtained advantages of dataflow optimization as shown in Table 3.7. Table 3.7 shows that the latency of vector-vector subtraction is completely removed using the dataflow technique.

Table 3.7: Latency and resource utilization of the first step on Artix7 (XA7A100TCSG324) -1q for both options for M=10 and N=256

	No Optimization	Parallel matrix-vector Mult. Only	Parallel vector-vector Add. Only	Pipelining without Dataflow	Pipelining with Dataflow
Latency (cycles)	23809	1293	258	1550	1295
Clock period (ns)	7.19	8	7.19	8	8
FF	243	1434	22	1454	1466
LUT	234	1248	59	1313	1280
DSP48E	4	40	0	40	40
Power	45	271	7	280	277

Step two and three require locating the largest values of a vector and update the residue value for next round respectively. The same optimization techniques in steps three and six of section 3.5.3 are applied.

3.5.4.2 Parallel implementation on GPU

Steps one, two, and three of the MB-IHT algorithm have been parallelized using the same techniques in steps two, three, and six of the BCoSaMP algorithm, respectively.

3.6 Parallelization of Discrete Wavelet Transformation Method

3.6.1 Introduction

Discrete Wavelet Transformation (DWT) is an effective signal processing method used in many applications [135-137] such as image processing, video compression, signal analysis, and computer vision. Signal processing in frequency-domain provides the spectral content of the signal. On the other hand, signal processing in time-domain doesn’t support wide outlook of the signal as most of the information is hidden in the frequency content. Wavelet transform provides time-frequency representation which gives a multi - resolution outlook of the signal. It is also a powerful technique to remove the noise from the signals without distorting the quality of the processed signal. Moreover, the process of DWT reconstruction is considered lossless which is very attractive for signal de-noising.

Several VLSI based hardware implementations [138-139] have been proposed to implement DWT where many large resources and complex routing are used. VLSI based implementations lack of flexibility and can’t be easily reconfigured for other operations even within the same domain. It also imposes a lack of adaptability as the device is in use within a system for purposes such as correcting faults. Moreover, the development is costly and time consuming, and thus they are not an attractive option for implementing the wavelet transforms [140].

A survey of existing implementations and architectures are demonstrated by the limited contributions in [141-142]. In [143], a pipelined architecture for real-time DWT was proposed and implemented on FPGA. It shows the advantage of the operating frequency, but with highly increasing the number of resources (three times the direct one) which consequently increase the area and the power consumption. In [144-145], a design and implementation of 3-D Haar Wavelet Transform (HWT) using dynamic partial reconfiguration was proposed which presents advantage of speed. However, it requires high resources utilization (more than 20,000 slices) and high power consumption (more than 1600 mW) because it implemented the array as registers causing a lot of memory space is wasted. Moreover, the design is complex and large due to use of large numbers of multiplexers which consequently increasing the power consumption and complexity of the design.

In [146], FPGA implementation was proposed of 3-D wavelet for video segmentation. The 3-D DWT also widely applied for medical applications as it provides perfect reconstruction property. However, it involves several computational steps to calculate DWT coefficients. The design has a slice utilization of 63% and the maximum frequency allows a 100 MHz system clock.

The decomposition and reconstruction computation of the DWT is a computationally intensive process, especially for 3-D and may fall short in meeting real time applications. Therefore, efficient design of DWT is required to achieve the desirable goals. The following sections show an efficient design of the DWT method for all dimensions (1-D, 2-D, and 3-D) for both forward and backward types.

3.6.2 Discrete Wavelet Transformation (DWT) Method

DWT performs the transform and reconstruction of signals using filter banks and wavelet filters. DWT analyses the input signal at different time periods. The signal is decomposed into an approximation and detail information and required variable number of steps depending upon the length of the transform. In this work, Haar wavelet transform (HWT) [147] has been chosen as it offers fast, memory efficient, reversible without the edge effects, and is appropriate for hardware implementation. Each step of DWT will provide a set of approximation and detailed values that is half of the original signal size. This procedure proceeds until reaches one coefficient value as shown in Figure 3-36. Approximation corresponds to a Low Pass Filter (LPF) that keeps only the low frequencies of the data. The detail process corresponds to a High Pass Filter (HPF) that keeps only the higher frequencies of the data. Low and high pass filters are defined by the following relations:

$$\mathrm{LPF} = \left(\frac{x_i + x_{i+1}}{2}, \frac{x_{i+2} + x_{i+3}}{2}, \ldots \ldots \ldots, \frac{x_{i+(n-1)} + x_{i+n}}{2} \right) \tag{3.43}$$

$$\mathrm{HPF} = \left(\frac{x_i - x_{i+1}}{2}, \frac{x_{i+2} - x_{i+3}}{2}, \ldots \ldots \ldots, \frac{x_{i+(n-1)} - x_{i+n}}{2} \right) \tag{3.44}$$

Where, i is the sample number and n is the total number of samples.

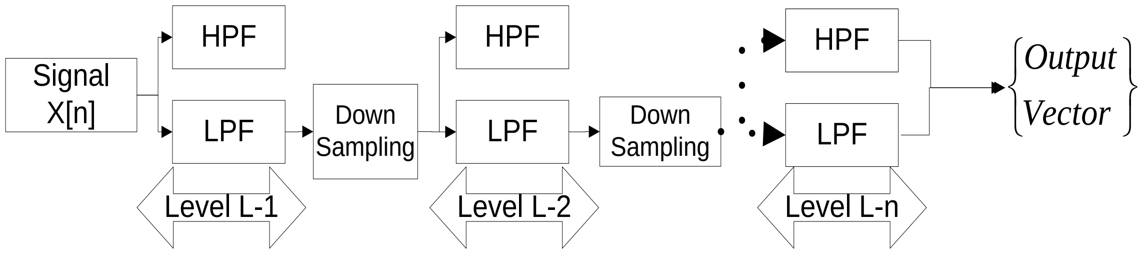


Figure 3-36: Haar Wavelet Transform (HWT).

The Haar function can be described as a step function $\psi(t)$:

$$\psi(t) = \begin{cases} 1, & 0 \le t \le 0.5 \\ -1, & 0.5 \le t \le 1 \\ 0, & otherwise \end{cases} \tag{3.45}$$

$\psi(t)$ is called mother wavelet and its scaling function $\phi(t)$ can be described as:

$$\phi(t) = \begin{cases} 0, & 0 \le t < 1 \\ 1, & Otherwise \end{cases} \tag{3.46}$$

In order to perform wavelet transform, Haar wavelet uses translations and dilations of the function using the following formula:

$$\psi_{ab}(t) = \frac{1}{\sqrt{|a|}} \ \psi\left(\frac{t-b}{a}\right) \quad \text{with } a,b \in \mathbf{R}\ ,\ a \neq 0\ . \tag{3.47}$$

In Haar transform, 2^n data length uses n levels. Averaging and differencing coefficients are computed for the next level from the previous level. The process is called Fast Haar Transform (FHT). An example of FHT with sixteen samples and four levels is shown in Figure 3-37. Its sample data is as follows:

F= [4 5 3 6 12 7 8 0 14 3 3 4 5 2 8 0]

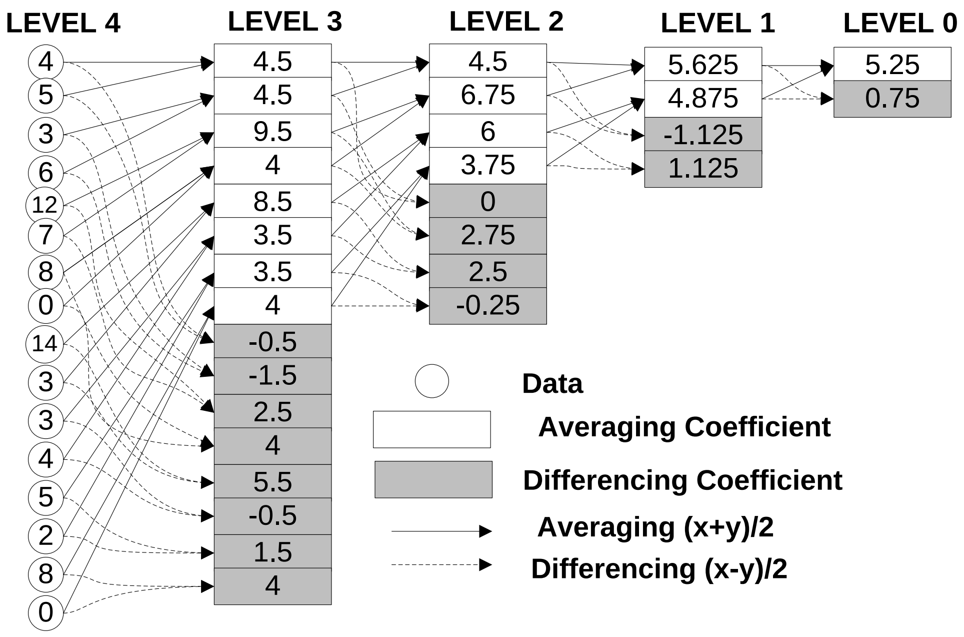


Figure 3-37: Fast Haar Transform (FHT) operations.

3.6.3 Analysis and Optimization Techniques

1-D forward DWT is shown in Figure 3-38 to illustrate how DWT will be optimized.

Function DWT 1-Dimension Forward

```
HDWT(Input[N])
Loop1: {For count ⟵ N to 1, count/=2
   Loop2: {For i ⟵ 0 to count/2
        {OP1: avg[i]= (Input[2*i]+Input[2*i+1])/2;
         OP2: diff[i] = Input[2*i]-Input[2*i+1])/2;}
   Loop3: For i ⟵ 0 to count/2
        {Input[i]= avg[i];
        Input[i + count/2]= diff[i];}}}
```

Figure 3-38: 1-D DWT Implementation

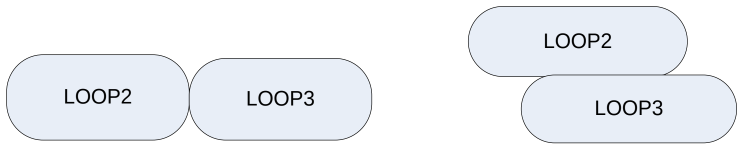
DWT has one main loop (Loop 1) and two inner loops (Loop 2 and Loop 3) as shown in Figure 3-38. Full parallelization of DWT can’t be achieved due to data dependencies between the inner loops (Loop 2 and Loop 3). However, Loop 2 can be executed in parallel. This parallelization can be prevented due to the resource contention because both the statements share the same resource array “Input array”. Thus, multi read and write ports are applied to the input array to overcome this dependency. Also, Loop 3 can be executed in parallel after Loop 2 is completed its iteration.

Loop unrolling and loop pipelining for Loop 2 have been experimented via simulation and their performance parameters are shown in Table 3.8 to determine which approach will provide superior results. Loop unrolling achieves better speed, but it causes excessive memory usage in terms of Look Up Tables (LUTs), Flip-Flops (FFs) usage and high power dissipation. On the other hand, the performance of the pipelining was found to be slightly less than unrolling in terms of latency but it achieves approximately 43 times less power dissipation. Hardware resources in terms of LUTs and FFs are also reduced 41 and 44 times respectively. So, pipelining technique is chosen in our work as our goal is to achieve low latency with awareness of hardware resources and power dissipation.

Table 3.8: Latency and Resources Utilization of Vector Size 1000

	No Optimization	Pipelining Method	Unrolling Method
Latency (cycles)	5428	2747	2701
Clock period (ns)	8.03	8.03	7.83
FF	158	180	7992
LUT	347	361	14963
Power (mW)	49	53	2295

Moreover, it is not necessary for Loop 3 to wait until Loop 2 completes all its iterations. So, Loop 3 can start execution after the first iteration of Loop 2 is completed. This can be exploited by applying the dataflow technique between these loops where the data can flow asynchronously from the first loop to the next one. So, Loops 2 and 3 execute in a pipelined fashion until all iterations are completed as shown in Figure 3-39.



A) Without Dataflow (B) With Dataflow

Figure 3-39: Loop dataflow pipelining technique.

In the case of 2-D and 3-D, we can achieve better performance as the computations in other dimensions are independent of each other. Therefore, previous 1-D parallelization of the filter bank is applied to other dimension in cases of 2-D (x and y) and 3-D (x, y and z).

This parallelization approach provides shorter computation time. However, this will be prevented due to the resource contention because all the computations share the same resource array “Input array”. Thus, in order to overcome this dependency we can implement the array using LUT based memory. This approach requires excessive usage of LUTs and FFs due to use a lot of multiplexers. It then results in higher power consumption as shown in Table 3.9. However, our implementation goal is to achieve high speed with awareness of power, area and cost. Therefore, we neglect this high resource requirement and storage overhead to minimize the area, cost, power consumption and design complexities.

Table 3.9: Resource Utilization of 2-D DWT of Vector Size 128

	Block Memory	LUT Based Memory
Latency (cycles)	43952	29683
Clock period (ns)	4.76	4.76
Occupied slices	2654	13827
Power (mw)	249	1492

3.7 Parallelization of Particle Filter for Tracking Applications

Particle filter has been proven to be a very effective method for identifying targets in non-linear and non-Gaussian environment. However, particle filter is computationally intensive and may not achieve the real time requirements. So, it’s desirable to implement particle filter on FPGA by exploiting parallel and pipelining architecture to achieve its real time requirements.

3.7.1 Introduction

Target tracking [148-149] can be defined as a sequential estimation of a variable or target of interest based on some observations over a period of time. Target tracking is performed by obtaining the position of the target. It is accomplished by performing position predictions and estimating the target positions in consecutive time scans. Different factors play important role in the efficiency of tracking process such as target parameters (position, velocity), target motion, and algorithm selection. However, the objective of the tracking process is to provide sequential prediction of the target using some observations. So, the tracking process can be divided into two stages; state stage which represents the values of target interest (prediction); and observation stage which represents some specified probabilistic relationship between observation and state (feature extraction).

Tracking applications are very important for both military and civilian applications. Different versions of Kalman filters and particle filters are available in the literature [150]. Kalman filters are used when the system is linear and Gaussian whereas particle filters are popular when the system is non-linear and non-Gaussian. However, most of tracking methods are computationally intensive, especially in Multi Target Tracking (MTT) radar systems. This leads to a heavy computation burden which prevents tracking to be performed in real-time. So, efficient hardware implementation will be required with the use of the parallel processing platform. Hardware implementation should be efficient in terms of latency, area, power consumption, cost, and flexibility.

3.7.2 Particle Filter Operation

Particle filter [150-153] offers the following advantages:

- It is very effective in identifying the targets in an efficient and accurate manner.
- It can be useful in radar tracking applications with high cluttered environment.
- It is appropriate for tracking targets where the system is nonlinear and non-Gaussian.

Moreover, in the presence of multiple targets, tracking becomes difficult as the discrimination of target is inaccurate. Particle filters calculate the posterior density for different values of the targets which is converted to likelihood functions and helps to detect the number of targets. This approach can simultaneously handle processing, data association and target tracking. Particle filter consists mainly of four steps as follows:

1. Prediction and measurement step.
2. Importance step.
3. Resampling step.
4. Output estimation.

Figure 3-40 shows the computational steps to implement the particle filter. It shows data dependencies between various operations and their computational sequence.

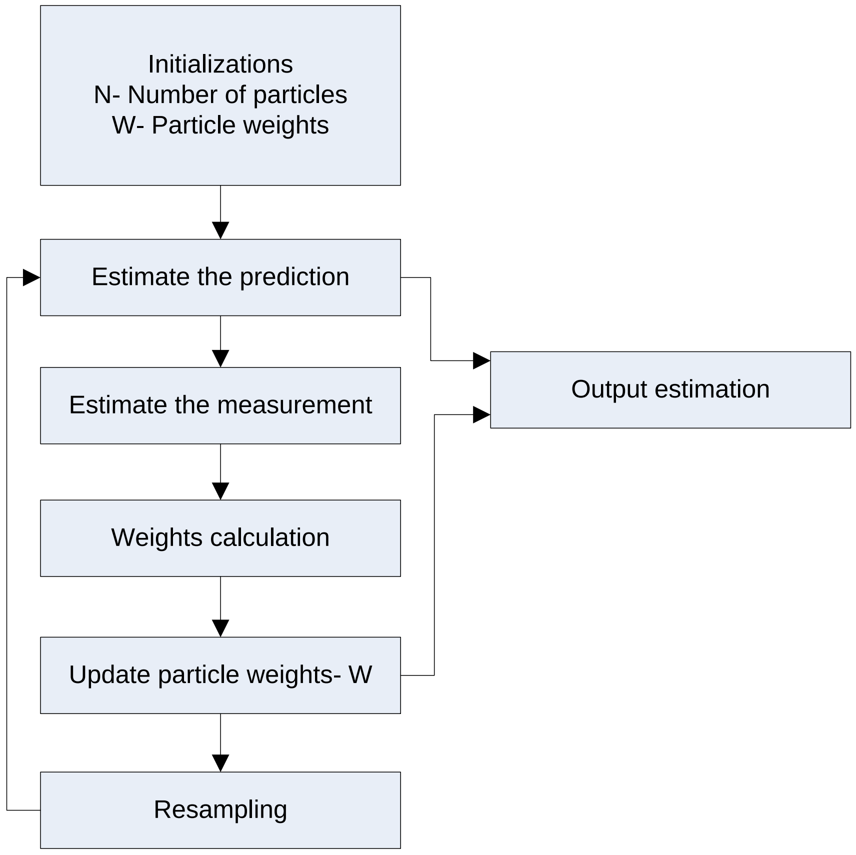


Figure 3-40: Particle filter computational steps

The first step will initialize a nonlinear system prediction for assumed number of particles. Then the prediction and measurement of the new state of the target will be performed through the particles by considering a nonlinear system as follows:

$$x_k = f(x_{k-1}) + w_{k-1} \tag{3.48}$$

$$z_k = h(x_k) + v_k \tag{3.49}$$

Where x_{k-1} is the input target position, w_{k-1} and v_k are the process and measurement noise, f and h are nonlinear functions of process and observation vector, and z_k is the current observed measurement.

In order to show the robustness and effectiveness of particle filter for tracking purposes, a complex system with difficult state estimation is used in both the processes and measurements. It is expressed as [150]:

$$x_k = \frac{1}{2}x_{k-1} + \frac{25x_{k-1}}{1+x_{k-1}^2} + 8\cos[1.2(k-1)] + w_k \tag{3.50}$$

$$y_k = \frac{1}{20}x_k^2 + v_k \tag{3.51}$$

This highly nonlinear system is widely used for state estimation and comparison of efficiency and performance of new algorithms [154-155].

The second step will involve the estimation and normalization of the particle weights [150] based on Equations (3.52) and (3.53). This identifies the particles that have the highest probability to represent the desired target. The weights of few particles will have large values as time progresses while the remaining weights of other particles will decrease in their values.

$$w_i = \frac{1}{(2\pi)^{m/2}\left|R\right|^{1/2}}\exp\left(\frac{-[z^* - h(x_{k,i}^-)]^T R^{-1}[z^* - h(x_{k,i}^-)]}{2}\right) \tag{3.52}$$

$$w_n^m = w_n^m / \sum_{m=1}^{M} w_n^m, \text{ for m=1,...,M.} \tag{3.53}$$

Resampling process in third step will remove small negligible weights particles and keep the largest one. This will improve the estimation of the future state by considering particles of higher posterior probability. This can be accomplished in different ways. One straightforward way can be performed in the following two steps:

- Generate a random number r [0,1].
- Accumulated the likelihoods w_n^m into a sum until the total sum is greater than r ($\sum_{m=1}^{j} w_m \geq r$). Then the new particle will be set to the old particle.

The final step will perform output calculations by multiplying the normalized weight by the predicted measurement of the particle as follows:

$$p(x_k \left| z_{1:k} \right.) \approx \sum_{i=1}^{Ns} W_{ik} * x_k^i \tag{3.54}$$

3.7.3 Analysis and Optimization Techniques

3.7.3.1 Parallel implementation on FPGA

Particle filter is broken into set of regions as shown in Figure 3-41 in order to exploit the parallel architecture of FPGA platform and the inherent parallelism of particle filter. However, full parallelization of particle filter can’t be achieved as particle filter is an iterative algorithm where the new particle prediction can’t be performed until the resampled step is completed. Also, there is a data dependency as the next computational step depends on the result of the current step. So, we need a way to arrange the operations of the algorithm to be performed in parallel without affecting its functionality. The implementation of particle filter can be improved by applying two optimization techniques:

- Merging technique: consecutive loops will be merged to reduce overall latency, increase sharing and optimization.
- Dataflow technique: allows sequential loops to be performed in a pipeline fashion to improve throughput and latency.

The merging technique allows the operations to be performed in one operation to reduce the additional overhead. For example, prediction step, measurement step, and weight calculation can be performed in one loop. This reduces the overhead from the unnecessary loops as additional N iteration loops for each step is removed. Also, weight normalization can be merged with resampling step to remove the N iteration loops of the normalization with little modification of resampling. The modified particle filter algorithm is as follows:

```
Merged Particle Filter
For i ← 0 to N {
  Prediction step
  Measurement step
  Weight calculation}
For j ← 0 to N {
  Normalization step
  Resampling step based on $\sum_{m=1}^{j} w_m \geq r$
```

So, the operations of the particle filter in this approach are merged instead of specifying a separate loop of N iterations for each particle filter operation.

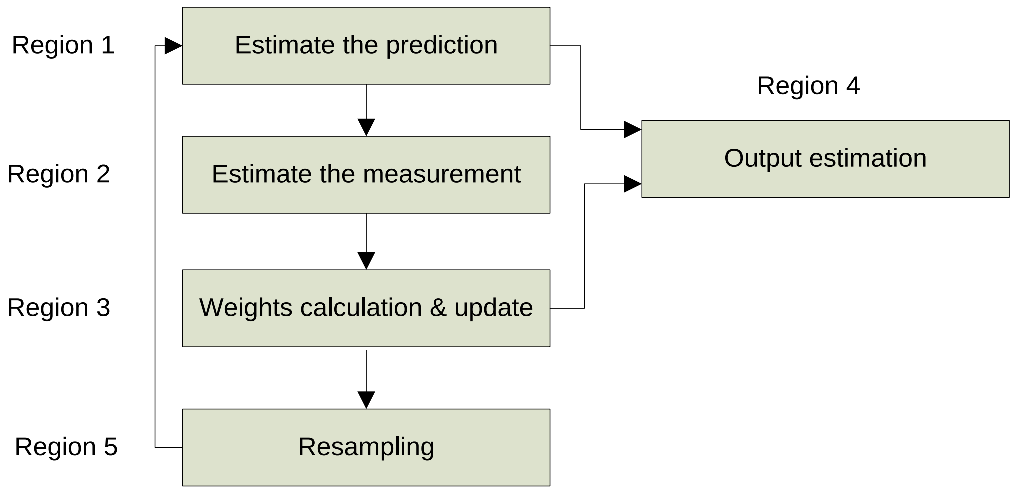


Figure 3-41: Computational regions of particle filter

Moreover, it is not necessary for region 2 to wait until region 1 completes all its iterations. So, region 2 can start execution after the first iteration of region 1 is completed. This can be exploited by applying the dataflow technique between these regions where the data can flow asynchronously from the first region to the next one as shown in Figure 3-42.

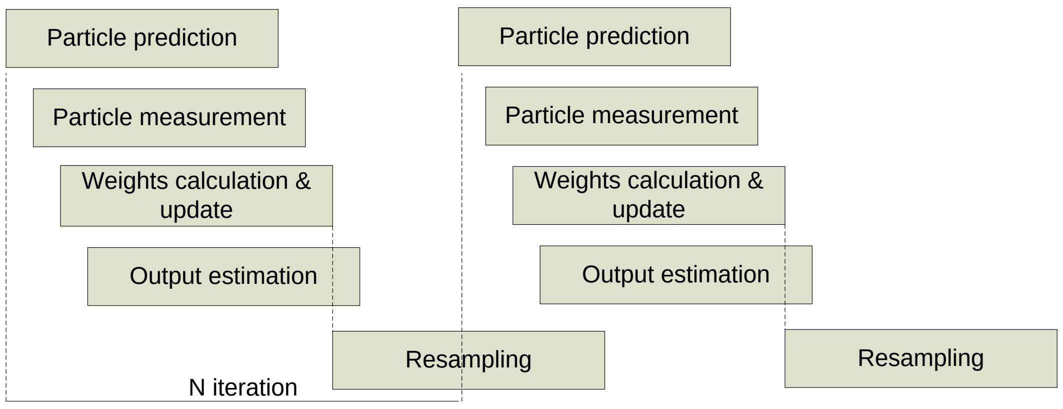


Figure 3-42: Timing diagram for overlapping particle filter operations

So, regions 1 and 2 execute in a pipelined fashion until all iterations are completed. Region 1 forwards the value from current iteration to region 2 and begins with the next iteration at the same time region 2 can start execution. Similar approach is applied to other regions and is shown in the following pseudo code.

```
Parallelization Method
For each blip:

For k ← 1 to N                          } This Loop can be executed in parallel.
    $x_k = f(x_{k-1}) + w_{k-1}$
End

⇩ Applying Loop Dataflow Pipelining

For k ← 1 to Ns                         } This Loop can be executed in parallel.
    $z_k = h*(x_k) + v_k$
End

⇩ Applying Loop Dataflow Pipelining

For k ← 1 to Ns                         } This Loop can be executed in parallel.
    Wight calculation
End

⇩ Applying Loop Dataflow Pipelining

For k ← 1 to Ns                         } This Loop can be executed in parallel.
    $p(x_k | z_{1:k}) \approx \sum_{i=1}^{Ns} W_{ik} * x_k^i$
End

⇩ Applying Loop Dataflow Pipelining

For k ← 1 to Ns
    Resampling based on $\sum_{m=1}^{j} w_m \geq r$ .
End
```

Moreover, all these operations of particle filter are for only x-dimension. We need to implement the same operations for y-dimension. Fortunately, they are independent of each other and can be executed in parallel. Moreover, the steps of x-dimension and y-dimension can also be merged together to reduce overall latency, increase sharing and optimization.

3.7.3.2 Parallel implementation on GPU

Step one of particle filter is the position initialization of each particle. This step can be fully parallelized by N threads where each thread is assigned for each particle as shown in the following code:

Parallel Particle Positions Initialization

```
int i ◄—  Thread index;
   {Xpart[i] = x + sqrt (Q) * random();  //Q is process noise covariance
    Ypart[i] = y + sqrt (Q) * random();}
```

Step two is the prediction and measurement state of each particle. This step can be fully parallelized by N threads where each thread is assigned for each particle as shown in the following code:

Parallelization of Particle Prediction State

```
int i ◄— Thread index;
   {Xpartminus[i] = Xpart[i]+sqrt (Q) * random();
    Ypartminus[i] =  Ypart[i] + sqrt (Q) * random();}
```

Parallelization of Particle Measurement State

```
int i ◄— Thread index;
   {XXpart[i] = H * Xpartminus[i]; //H is the measurement transition matrix
   YYpart [i]= H * Ypartminus[i];}
```

Moreover, particle filter requires repeating the basic calculation of random function generator for all the particles in each iteration. This basic computation typically involves a significant amount of calculation which represents a small fraction of the total computational effort. So, random number generation is also parallelized where each thread generates one random value instead of a large number of values.

Step three is the weight calculations of each particle. This step can be fully parallelized by N threads as shown in the following code:

Parallelization of Particle Prediction State

```
int i ⟵ Thread index;
{Vhat = xmeasured[k] - XXpart[i];
 yyhat = ymeasured[k] - YYpart[i];
 q[i] = (1 / sqrt (r) / sqrt (π) * exp(-vhat^2 / 2 / R);  //R is measurement noise covariance
 qy[i] = (1 / sqrt (r) / sqrt (π) * exp(-yyhat^2 / 2 / R);}
```

Step four requires the normalization of the particle weights which needs summing all the particle weights. In order to parallelize this operation efficiently, we have divided the particle weights by multiple threads where each thread sums a group of weights elements into its local variable. Then the global summation will be performed by adding these local variables. However, this technique requires a lock and barrier synchronizations to ensure correct results since the global summation variable is shared by all the threads. Following code is used to find the summation value of particle weights.

Parallelization of Particle Weights Summation

```
int index ⟵ Thread index;
Tile_Size=Weights_Vector/# of threads.
x=index*Tile_Size, y =(index+1)*Tile_Size.
For i ⟵ x to y {
Local_sum_X= Local_sum_X + W_x[i];
Local_sum_Y= Local_sum_Y + W_y[i];}
__syncthreads();
Global_sum_X+=Local_sum_X;
Global_sum_Y+=Local_sum_Y;
Unlock_suncronization();
```

Normalization step which divided each particle weight by the total weights summation can be fully parallelized by N threads as shown in the following code:

Parallelization of Particle Weights Normalization

```
int i ◄— Thread index;
 {q[i] = q[i] / Global_sum_X;
 qy[i] = qy[i] / Global_sum_Y;}
```

3.8 Conclusion

A parallel algorithm is developed for IR video processing which includes background subtraction, noise filtering and connected component labeling algorithms on the GPU. The video processing algorithm was also partitioned, parallelized, mapped and scheduled on multi-core. We have analyzed and estimated the energy consumption for all the components of the NoC platform for processing elements, memory, caches, routers and communication architecture to find the bottlenecks in the platform for IR video processing. Also, a new modeling and simulation approach regarding the channel width and buffer sizing is proposed to get a better performance. D^3, ECA, BCoSaMP, particle filter, IHT, and DWT algorithms have been also transformed for optimal execution, implemented and parallelized on both FPGA and GPU architectures. An extensive analysis and demonstration of various parallel strategies are performed. The developed parallel implementation uses different methods and approaches to design a parallel strategy of these algorithms efficiently.

Chapter 4

Radar Signal Processing Tool for Parallel Architectures

4.1 Radar Signal Processing Tool (RSPT)

A new software tool called Radar Signal Processing Tool (RSPT) has been developed. RSPT is a framework tool unifying the aspects of algorithms, architectures, and software. It bridges the gap between the algorithm and architecture scientific communities. So, hardware software co-design has been performed that pushes performance and energy efficiency while reducing cost, area, and overhead. RSPT allows the designer to auto-generate a fully optimized VHDL representation for selected radar signal processing algorithms. This work focusses on development of FPGA based hardware for real-time execution of the selected signal processing algorithms.

The RSPT allows the designer to specify user input parameters for a specified algorithm through a Graphical User Interface (GUI). This offers great flexibility in designing a radar signal processing applications for a SoC without having to write a single line of VHDL code. Moreover, RSPT provides the designer a feedback on various performance parameters. So, the system designer will have an ability to make any adjustments to the radar signal processing until the desired performance of the overall System on Chip (SoC) is satisfied. The tool will utilize optimization techniques such as pipelining, code in-lining, loop unrolling, loop merging, and dataflow techniques by allowing the concurrent execution of operations to improve throughput and latency. The tool will provide FPGA implementation to achieve high speed with awareness of power, area and cost.

Our developed Radar Signal Processing Tool (RSPT) for FPGA consists of two main parts, VHDL Library, and a Java Graphical User Interface (GUI). The High-Level Synthesis Tool (HLST) is used to generate the VHDL Library for a certain algorithm. The VHDL library contains various functions and entities to construct the VHDL files based on user specified parameters. The GUI communicates with Xilinx and VHDL library to synthesize and generate the optimized VHDL code for the specified component as shown in Figure 4-1. It communicates using a standard worker thread process and redirected input/output streams. The worker thread checks for the available FPGA parts installed with the Xilinx toolset. They are also responsible for executing the VHDL synthesis command chain.

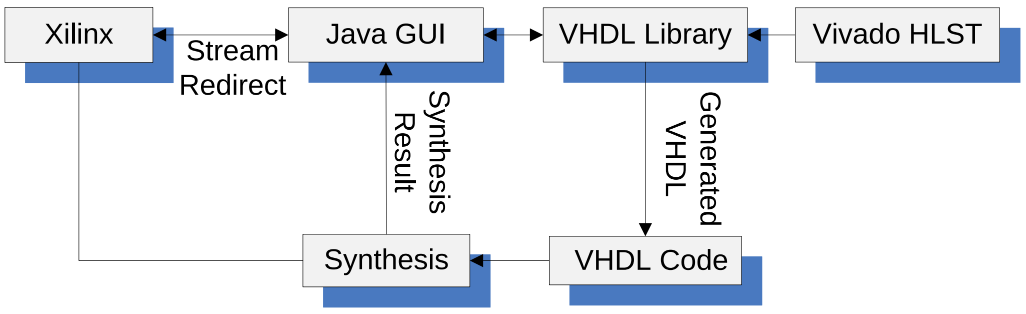


Figure 4-1: Overview of software package components

RSPT works as follows: the user specifies all input parameters of a certain algorithm and the desired FPGA part. Then, RSPT communicates with Xilinx toolset to check for the available FPGA parts. The RSPT then generates VHDL files for the specified parameters. RSPT tests the generated VHDL file through all the design process steps such as synthesis, translation, mapping, placing and routing, timing, and generating the bitstream file. Finally, RSPT provides the designer a feedback on occupied slices, maximum frequency, and dynamic range. It allows the designer to make any necessary changes to the component to achieve the desired goal.

The RSPT uses the procedure as shown in Figure 4-2. It auto-generates the top level VHDL for selected radar signal processing algorithm. The generation of VHDL implementation of a certain algorithm consists mainly of five steps as shown in Figure 4-2. The first step creates CADWORK folder to save the VHDL representation file. Then, it determines the algorithm location in the library and copies the VHDL library file that contains the main source code of the algorithm. It determines all the specified input parameters of the algorithm. Finally, it generates the VHDL file and ends the process. The generated VHDL file includes a full optimization representation that is ready to be synthesized by the Xilinx ISE.

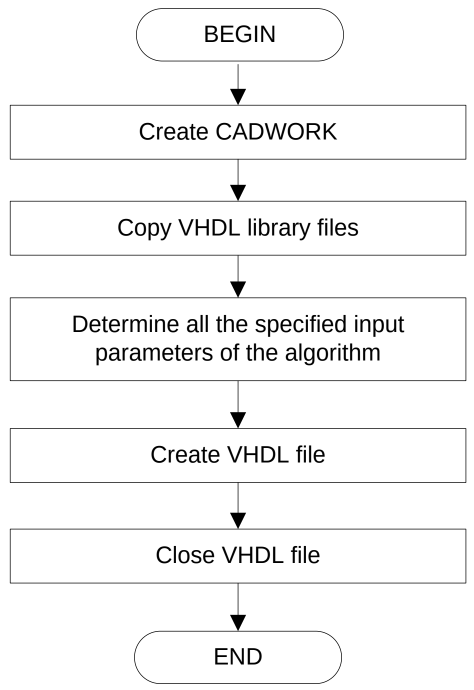
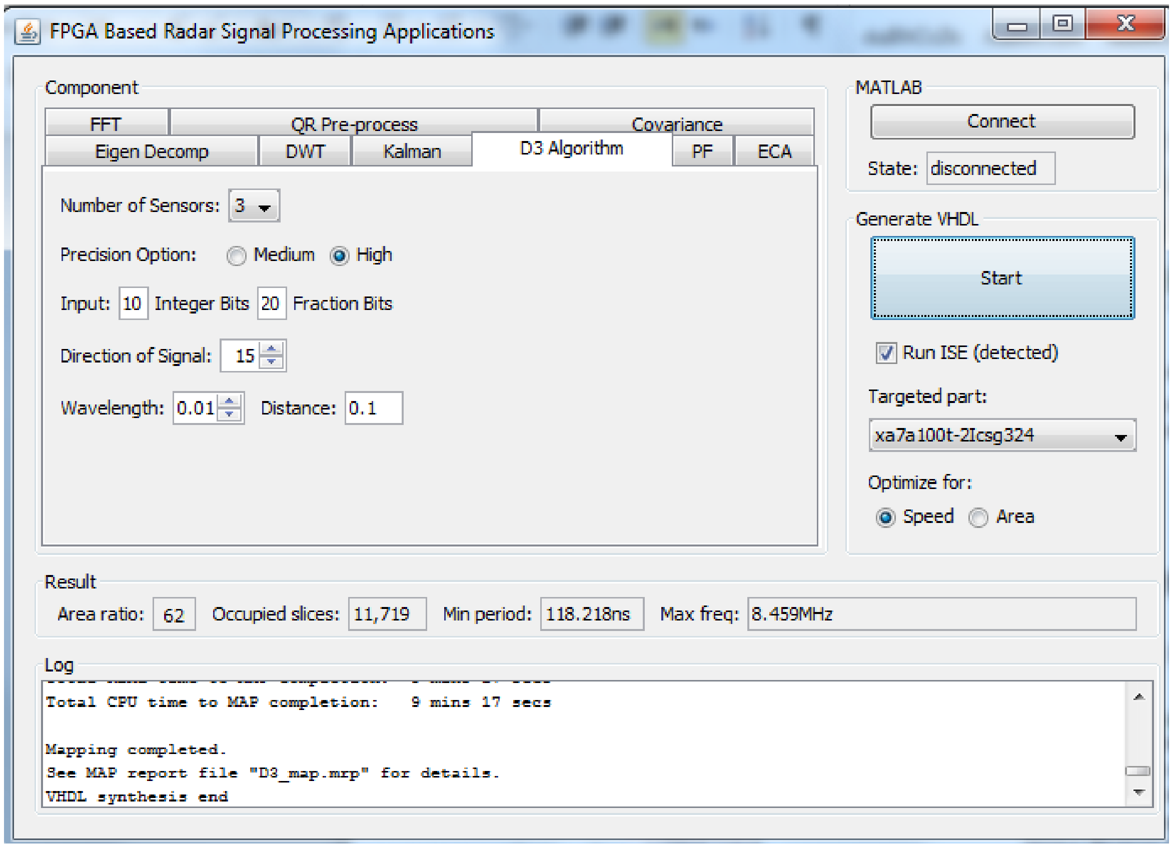


Figure 4-2: GUI flowgraph for VHDL auto-generation.

The GUI of D^3 algorithm for the RSPT is shown in Figure 4-3. One can see the redirected Xilinx output stream in the Log window, the Xilinx synthesis results, and the D^3 parameters selected for VHDL auto-generation. The functionality of the RSPT tool is driven by the user input and relies on flow graphs plus logic to auto-generate VHDL D^3 for a SoC. This offers great flexibility in designing a D^3 component for a SoC without having to write a single line of VHDL code. The D^3 parameters accepted by the tool are:

- Number of sensors.
- Precision option: Uses fixed point implementation with high and medium precisions. The high precision consists of ten integer bits and twenty fraction bits. The medium precision consists of eight integer bits and twelve fraction bits.
- Direction of the desired signal (0º to 90º).

- The wavelength of the signal (0.01-0.5 m) which corresponds to a frequency between 600MHz to 30GHz. The distance between antennas is at least half the wavelength.



Number of Sensors: 3

Precision Option: Medium [✓ High]

Input: 10 Integer Bits 20 Fraction Bits

Direction of Signal: 15

Wavelength: 0.01 Distance: 0.1

State: disconnected

[✓] Run ISE (detected)

Targeted part: xa7a100t-2Icsg324

Optimize for: [✓ Speed] Area

Area ratio: 62 Occupied slices: 11,719 Min period: 118.218ns Max freq: 8.459MHz

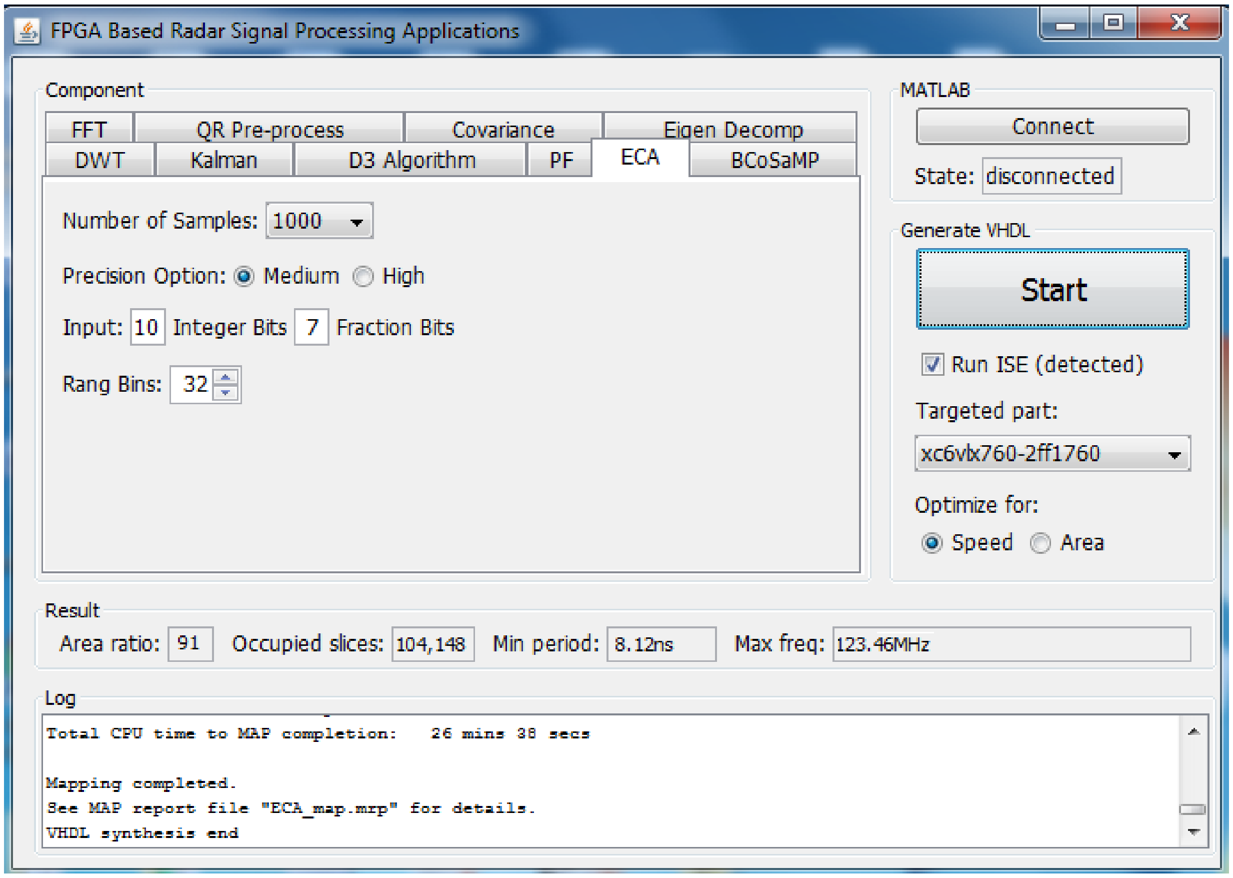
```
Total CPU time to MAP completion:   9 mins 17 secs

Mapping completed.
See MAP report file "D3_map.mrp" for details.
VHDL synthesis end
```

Figure 4-3: Graphical user interface for Radar Signal Processing Tool (RSPT) of D^3.

The GUI for ECA of RSPT is shown in Figure 4-4. One can see that the redirected Xilinx output stream in the Log window, the Xilinx synthesis results, and the ECA parameters selected for VHDL auto-generation. The ECA parameters accepted by the tool are:

- Number of samples.

- Precision option: Uses fixed point implementation with high and medium precisions. The high precision consists of ten integer bits and twenty fraction bits. The medium precision consists of ten integer bits and ten fraction bits.
- Range bins (from 1 to 100).



Number of Samples: 1000

Precision Option: [✓ Medium] High

Input: 10 Integer Bits 7 Fraction Bits

Rang Bins: 32

State: disconnected

[✓ Run ISE (detected)]

Targeted part: xc6vlx760-2ff1760

Optimize for: [✓ Speed] Area

Area ratio: 91 Occupied slices: 104,148 Min period: 8.12ns Max freq: 123.46MHz

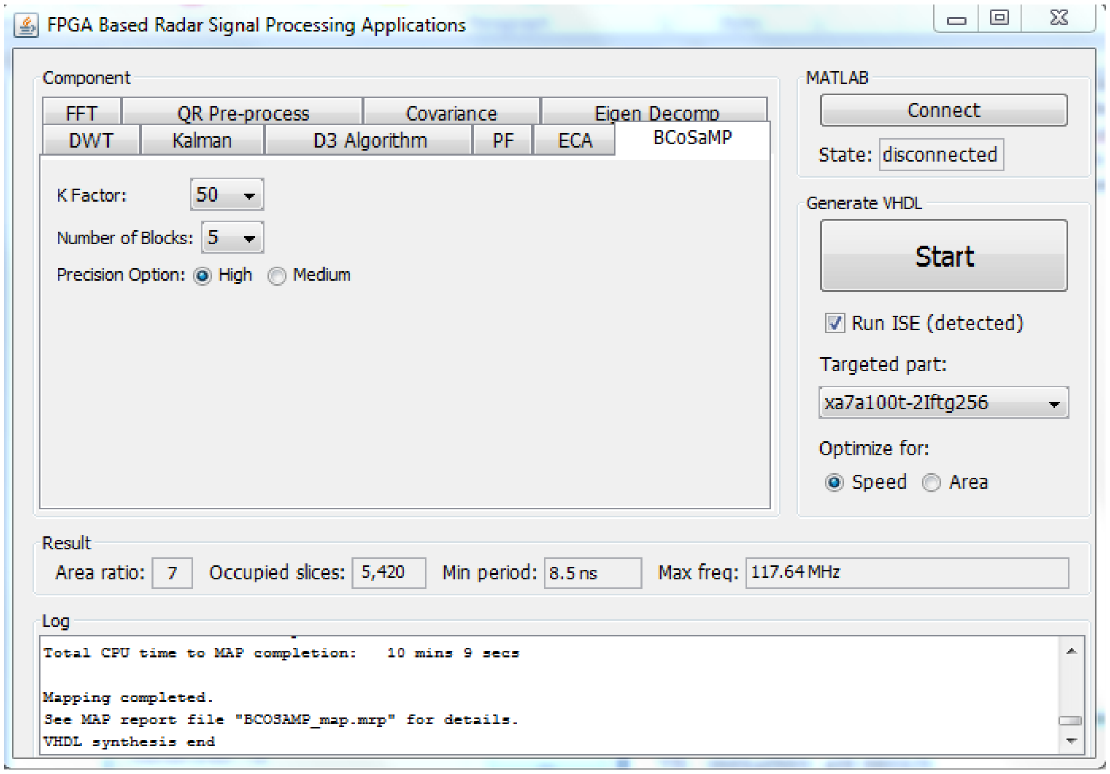
```
Total CPU time to MAP completion:   26 mins 38 secs

Mapping completed.
See MAP report file "ECA_map.mrp" for details.
VHDL synthesis end
```

Figure 4-4: Graphical user interface for Radar Signal Processing Tool (RSPT) of ECA.

The GUI for BCoSaMP of RSPT is shown in Figure 4-5. The BCoSaMP parameters accepted by the tool are:

- K Factor: this parameter identifies the largest components of the signal vector to get the final approximation.

- Number of Blocks: this identifies the number of divisions of the signal vector.
- Precision option: Uses fixed point implementation with high and medium precisions. The high precision consists of ten integer bits and twenty fraction bits. The medium precision consists of eight integer bits and twelve fraction bits.



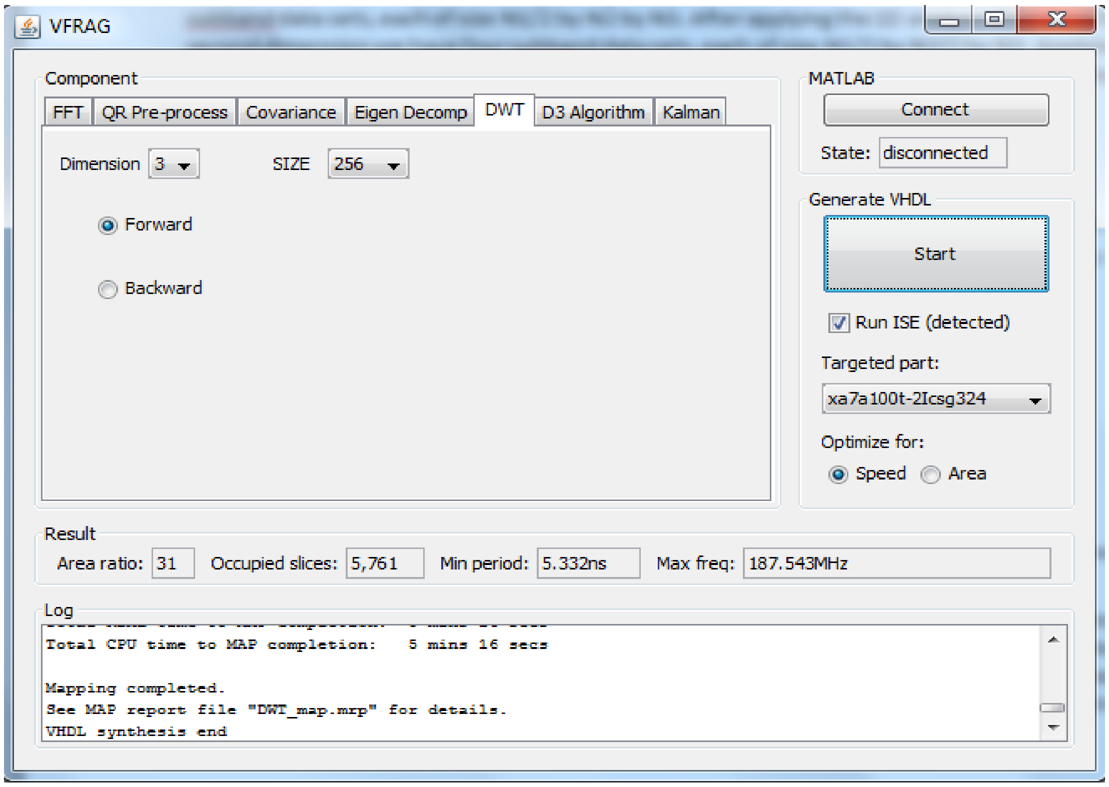
K Factor: 50
Number of Blocks: 5
Precision Option: [✓ High] Medium
State: disconnected
[✓ Run ISE (detected)]
Targeted part: xa7a100t-2Iftg256
Optimize for: [✓ Speed] Area
Area ratio: 7 Occupied slices: 5,420 Min period: 8.5 ns Max freq: 117.64 MHz

```
Total CPU time to MAP completion:   10 mins 9 secs

Mapping completed.
See MAP report file "BCOSAMP_map.mrp" for details.
VHDL synthesis end
```

Figure 4-5: Graphical user interface for Radar Signal Processing Tool (RSPT) of BCoSaMP.

The GUI for DWT of RSPT is shown in Figure 4-6. The DWT parameters accepted by the tool are:

- Dimension (1-D, 2-D, and 3-D).
- Input size (4, 16, 32, 64, 128, 256, 512, and 1024).
- Type (Forward or Backward).



Dimension 3 SIZE 256

[✓ Forward] Backward

State: disconnected

[✓] Run ISE (detected)

Targeted part: xa7a100t-2Icsg324

Optimize for: [✓ Speed] Area

Area ratio: 31 Occupied slices: 5,761 Min period: 5.332ns Max freq: 187.543MHz

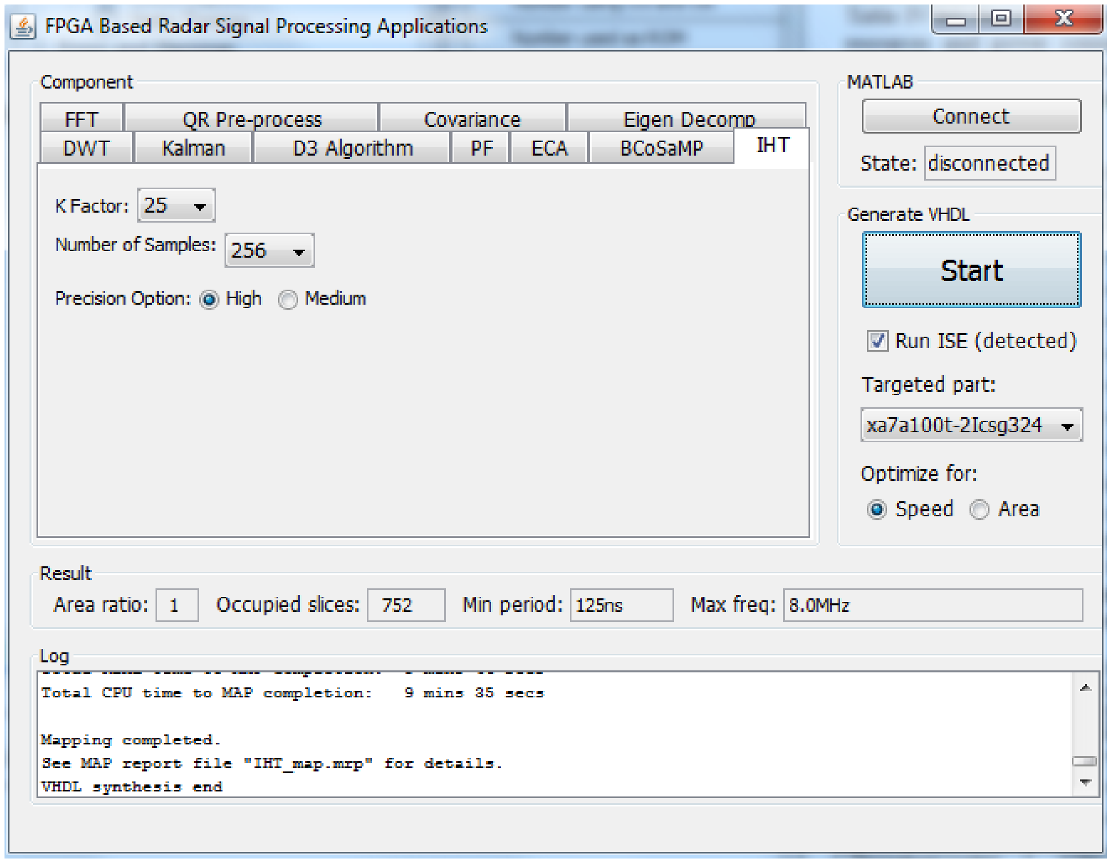
```
Total CPU time to MAP completion:   5 mins 16 secs

Mapping completed.
See MAP report file "DWT_map.mrp" for details.
VHDL synthesis end
```

Figure 4-6: Graphical user interface for Radar Signal Processing Tool (RSPT) of DWT.

The GUI for MB-IHT method of RSPT is shown in Figure 4-7. The MB-IHT parameters accepted by the tool are:

- K Factor: this parameter identifies the largest components of the signal vector to get the final approximation.
- Number of Signal Samples: this identifies the number of samples of the signal vector.
- Precision Option: Uses fixed point implementation with high and medium precisions. The high precision consists of ten integer bits and twenty fraction

bits. The medium precision consists of eight integer bits and twelve fraction bits.



K Factor: 25

Number of Samples: 256

Precision Option: [✓ High] Medium

State: disconnected

[✓ Run ISE (detected)]

Targeted part: xa7a100t-2Icsg324

Optimize for: [✓ Speed] Area

Area ratio: 1 Occupied slices: 752 Min period: 125ns Max freq: 8.0MHz

```
Total CPU time to MAP completion:   9 mins 35 secs

Mapping completed.
See MAP report file "IHT_map.mrp" for details.
VHDL synthesis end
```

Figure 4-7: Graphical user interface for Radar Signal Processing Tool (RSPT) of MB-IHT.

4.2 Conclusion

A new software tool called Radar Signal Processing Tool (RSPT) has been developed. It unifies the aspects of algorithms, architectures, and software which bridges the gap between the algorithm and architecture scientific communities. This helps in performing hardware software co-design that pushes performance and energy efficiency while reducing cost, area, and overhead. It will allow the designer to auto-generate fully optimized VHDL representation for any radar signal processing algorithm. This work focusses on development of FPGA based hardware for real-time execution of D^3, ECA, particle filter, DWT, IHT, and BCoSaMP algorithms. The RSPT allows the designer to specify user input parameters of any of these algorithms through a Graphical User Interface (GUI). This offers great flexibility in designing these algorithms for a SoC without having to write a single line of VHDL code. Moreover, RSPT provides the designer a feedback on various performance parameters such as occupied slices, maximum frequency, estimated power consumption and dynamic range. This offers the designer an ability to make any adjustments to the algorithm component until the desired performance of the overall System on Chip (SoC) is satisfied. RSPT also uses many optimization techniques to improve throughput and latency.

Chapter 5

Simulation Results

5.1 Simulation of Parallel IR Video Processing Algorithm on GPU

A bird and bat monitoring system has been developed that uses IR camera to monitor birds and bats activity. This will be useful in developing mitigation techniques to minimize the impact of wind turbines on birds and bats. Data from IR was recorded during migratory periods near Ottawa National Wildlife Refuge. Collected IR data need to be processed in order to track the targets of interest. However, IR video processing is computationally intensive and may not achieve the real time requirements. So, the IR video processing algorithm is parallelized for implementation on a GPU. It was implemented in C and CUDA. The algorithm was tested on the Intel Nehalem Quad Core processor with GPU. The average computation time of a single frame for each step in video processing in NVIDIA GeForce GTX 260 GPU and computation for serial implementation (Single Core) is shown in Table 5.1. We considered the communication overhead factor in our implementation which includes the data exchange between CPU and GPU. Our implementation minimizes communication overhead by not returning the result back from GPU to CPU in every phase.

Table 5.1: Average Computation Time in Milliseconds

	Single Core	GPU	Speedup
Background subtraction	16.58	5.81	2.853
Dilation	13.74	0.9	15.26
Erosion	13.71	0.9	15.233
Labeling	36.69	14.55	2.521
Total	80.72	22.16	3.642

Figure 5-1 shows the average execution time to process a frame for a single core is 80.72 ms giving a frame rate of 12.388 fps. So, it doesn't achieve the real time requirement of 25-30 fps. On the other hand, the average execution time to process a frame on GPU is 22.16 ms giving a frame rate of 45.126 fps that satisfies the real time requirement of 30 frames per second (fps).

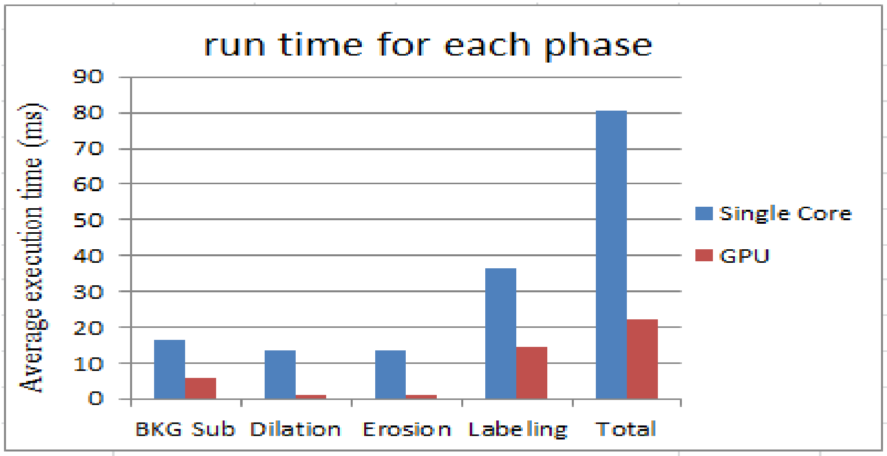


Figure 5-1: Execution time for each algorithm of video processing

Figure 5-2 shows the speedup for various parts of this algorithm. It shows that labelling part has smaller speedup as compared to other modules. This is due to data dependencies and overhead caused by synchronization of the threads. Background subtraction part includes data transfer to and from CPU to GPU. The parallel algorithm minimizes data transfer overhead by not returning the result back from GPU to CPU in every phase.

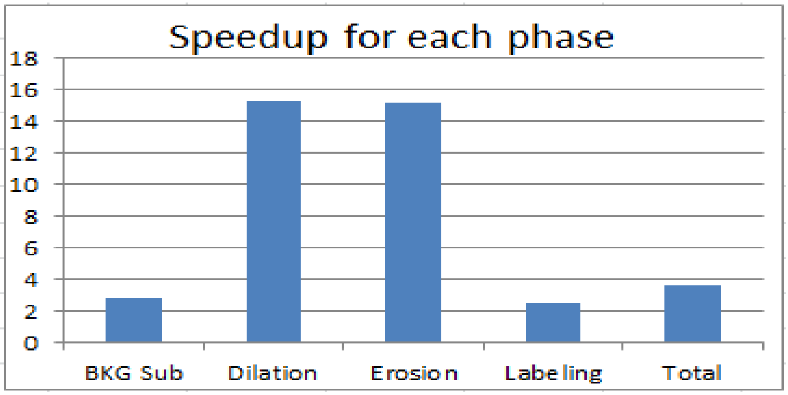


Figure 5-2: Speedup for each algorithm of video processing

5.2 Simulation of IR Video Processing Algorithm on MPSoC

An IR video processing data is used from our bird and bat monitoring system for simulation purposes. Simulation is performed by running a video processing application on a heterogeneous NoC platform based on mesh topology. A master processing element is responsible for reading the video frames and distributing partitioned tasks to other slave processing nodes. The memory hierarchy for IR video processing on MPSoC consists of a main memory in the Master PE, and a cache in each Slave PE (storage of the subtask). It was observed from the simulation that the memory and caches consume a large amount of the energy. This is compared to the energy dissipation in the communication architectures which consume much lower energy as shown in Figure 5-3.

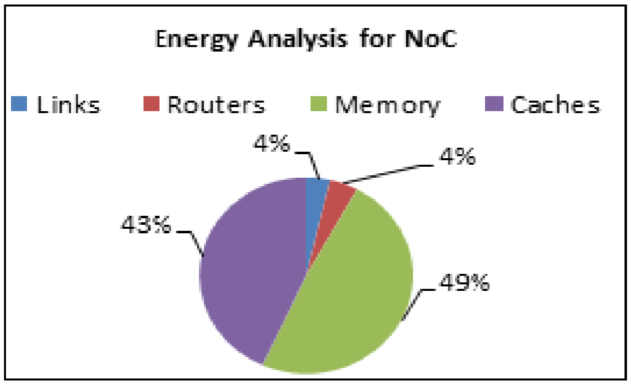


Figure 5-3: Energy analysis for the embedded platform.

Table 5.2 clearly shows that the energy consumption of the Master PE memory and Slave PEs caches are dominant. Therefore, some kind of optimization is needed for reduction of this energy bottleneck. So, the designer may reduce the energy dissipation in these bottlenecks by adopting techniques such as Data Transfer and Storage Exploration (DTSE), code transformation [156], circuit level [157-158] or other techniques.

Table 5.2: Energy dissipation and analysis of MPSoC platform

Architecture Component	Energy Dissipation (mj)
Master PE Memory	429.96
Slave PEs Caches	381.055
Routers (total)	37.95837
Links (total)	31.66011

This energy analysis shows contrary to the common belief that the global network interconnection is the bottleneck for the energy dissipation in handheld devices for multimedia applications.

The physical network model needs different bandwidths, different buffer sizes for links, and different routers sizes as there is a lot of congestion on the master core links and its router. So, the bandwidth of the links connected to the master core is increased by a factor of two by adding more virtual channel to improve the performance. The IR video processing algorithm was implemented in C on the Intel Nehalem Quad Core processor for comparison purposes. The average computation time of a single frame for single core, multi-cores with and without optimization are shown in Table 5.3. The results show the average execution time to process a frame for a single core is 80.72 ms giving a frame rate of 12.388 fps. On the other hand, the average execution time to process a frame on Multi-Core (without Optimization) is 42.48 ms giving a frame rate of 23.54 fps while with optimization giving a frame rate of 27.255 fps.

Table 5.3: Average Computation Time in Milliseconds

Method	**Latency**	**Speedup**
Single core	80.72	1
Multi-core (without Optimization)	42.48	1.9
Multi-core (with Optimization)	36.69	2.2

5.3 Simulation of Parallel Direct Data Domain (D^3) Algorithm

The parallelized Direct Data Domain (D^3) algorithm was tested using a single signal with jammer and additive white Gaussian noise as shown in Figure 5-4. Then the weights have been calculated based on a combination of inexact inverse iteration algorithm and conjugate gradient method to nulls the noise and interference while maintaining the desired signal as shown in Figure 5-5. It can be seen that the filtered signal has high amplitude and jammer were completely attenuated giving a very small amplitude for the jammer. The program uses following test signal and jammers.

$$S(t) = A \times \sin(wt + \phi)$$

where,

$A_s = 1.0$; $f_s = 1$ MHz; $\phi_s = 0.25\pi$ rad

$J_1 = 1.25$; $f_{j1} = 1$ MHz; $\phi_{j1} = 0.4167\pi$ rad

$J_2 = 2.00$; $f_{j2} = 1$ MHz; $\phi_{j2} = 0.3333\pi$ rad

$J_3 = 1.25$; $f_{j3} = 1$ MHz; $\phi_{j3} = 0.00\pi$ rad

Where A and J represent amplitude of signal and jammers respectively, f represents frequency, and ϕ represents the angle from the broadside of the arrays. The separation between the array elements (d) is equal to λ/2. The results obtained for three antennas are as follows:

Signal $[S(t)]$	**Received Signal** $[X(t)]$	**Interference before filter** $[X(t) - S(t)]$	**Interference after filter** $[(X(t) - \alpha S(t)) \times W]$
-0.5982 + 0.7859i	-1.7585 + 0.9105i	-1.1603 + 0.1246i	(0.3331 + 0.7216i)*10^{-15}
-0.2630 - 0.9520i	-2.0783 - 0.1391i	-1.8153 + 0.8130i	(-0.4441 - 0.3331i)*10^{-15}

The weight vector for this single signal example is W_1 = 0.4795 + 0.2771i; W_2 = -0.1716 - 0.8440i; α = 1.0823 - 0.8565i.

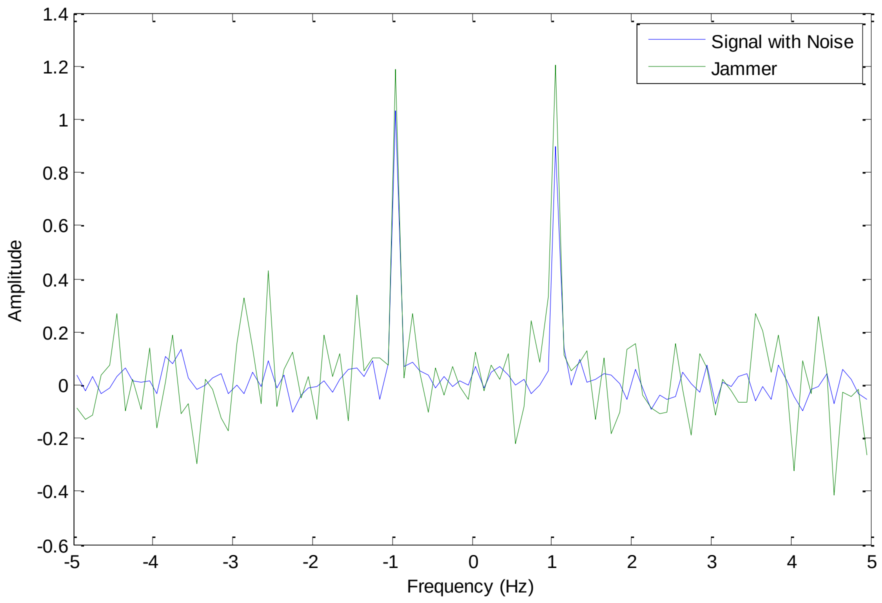


Figure 5-4: The received noisy signal and jammer before filtration.

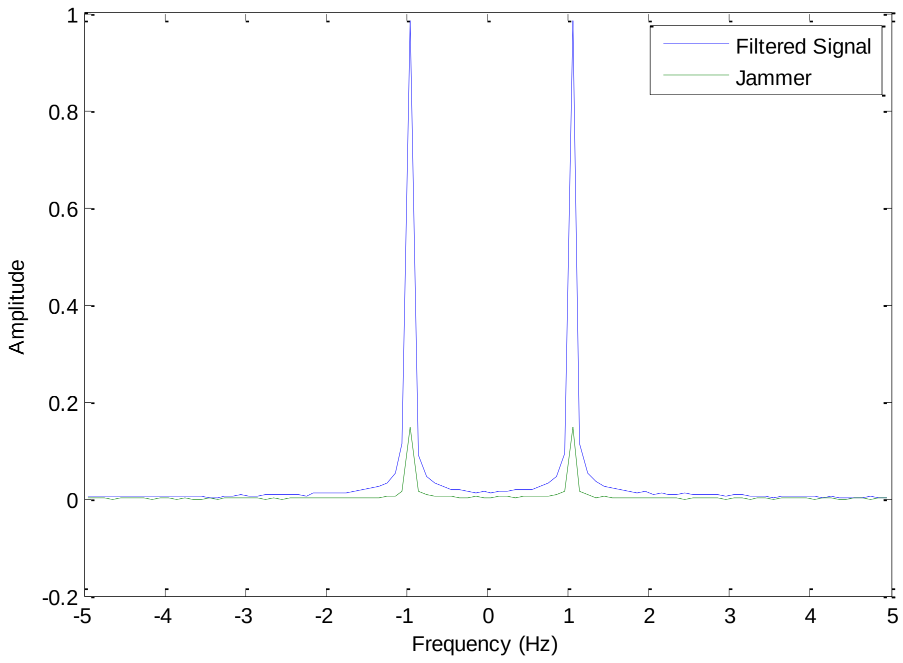


Figure 5-5: The received noisy signal and jammer after filtration.

The software tool (RSPT) automatically generates the VHDL D^3 component and synthesizes with the Xilinx ISE [159]. XA7A100T CSG324 -2I FPGA device is used in this work. Table 5.4 lists the overall performance results in terms of area, power consumption, and the maximum frequency for both high and medium precisions. The performance is measured with respect to following evaluation metrics:

- The throughput given in terms of the frequency.
- Hardware utilization given in terms of occupied slices, Flip Flop (FF), Lookup table (LUT), BRAM_18K, and DSP48E.
- The power dissipation.

It can be seen from the Table 5.4 that the hardware resources are higher for high precision than for medium which is expected. However, when the number of sensors equals two, the hardware usage such as FFs, LUTs, Slices is more than that of three sensors. It is due to the fact that some parameters and arrays depend on the number of sensors that is implemented on distribution memory of FPGA from logic elements. In the case of three sensors, more memory from FPGA (BRAM) is used instead of logic elements.

Table 5.4: Resources utilizations and overall implementation performance on Artix7 -2I (XA7A100T CSG324 -2I).

Parameters	Medium Precision			High Precision		
	Number of Sensors			Number of Sensors		
	2	3	4	2	3	4
Max. Freq. (MHz)	14.0	14.0	14.0	8.45	8.45	8.45
Occupied Slices	7001	6192	7181	12468	11719	12268
Slice LUTs	18463	18402	19096	40992	41497	42572
Slices of FF	11182	9914	12325	16526	14623	18251
LUT FF Pairs	24486	22419	25504	47403	44953	47269
DSP48E1s	18	18	18	64	64	64
BRAM_18K	6	14	14	6	14	14
Power (mW)	2395	2331	2529	4087	3999	4284

The design of the control unit of a deep pipelined data-path that controls the scheduling for medium and high precision is 49 and 71 stages respectively. The number of clock cycles, throughput, clock period and the execution time from the start of execution until the final output is written for medium precision and high precision for different number of sensors are shown in Table 5.5 and Table 5.6.

Table 5.5: Simulation time for different number of sensors for medium precision.

Parameter	Number of Sensors		
	2	3	4
Clock Frequency (MHz)	14	14	14
Clock Period (ns)	71.42	71.42	71.42
Throughput (cycles)	15206	25330	36052
Execution Time (ms)	1.086	1.809	2.574

Table 5.6: Simulation time for different number of sensors for high precision.

Parameter	Number of Sensors		
	2	3	4
Clock Frequency (MHz)	8.456	8.456	8.456
Clock Period (ns)	118.26	118.26	118.26
Throughput (cycles)	20680	35208	50734
Execution Time (ms)	2.445	4.163	6.00

D^3 is coded in C for its quick performance evaluation. Programs have been executed on a conventional PC powered by a 2.6 GHz i7-3720QM CPU with 8 GB RAM. The results of the execution times for i7 processor, our optimized FPGA, and GPU implementations are summarized in Table 5.7. Execution time is plotted in Figure 5-6 to show the effect of varying the number of sensors on the performance. The results show that the FPGA implementation performs better than other implementations. The superior performance of the FPGA-based implementations is attributed to the highly parallel and pipelined architecture.

Table 5.7: Execution time on different platform implementations

Implementation	Number of Sensors		
	2	3	4
i7-3720QM CPU (ms)	20.458	45.76	71.84
FPGA (ms)	2.445	4.163	6.00
GPU (ms)	3.22	5.89	7.58

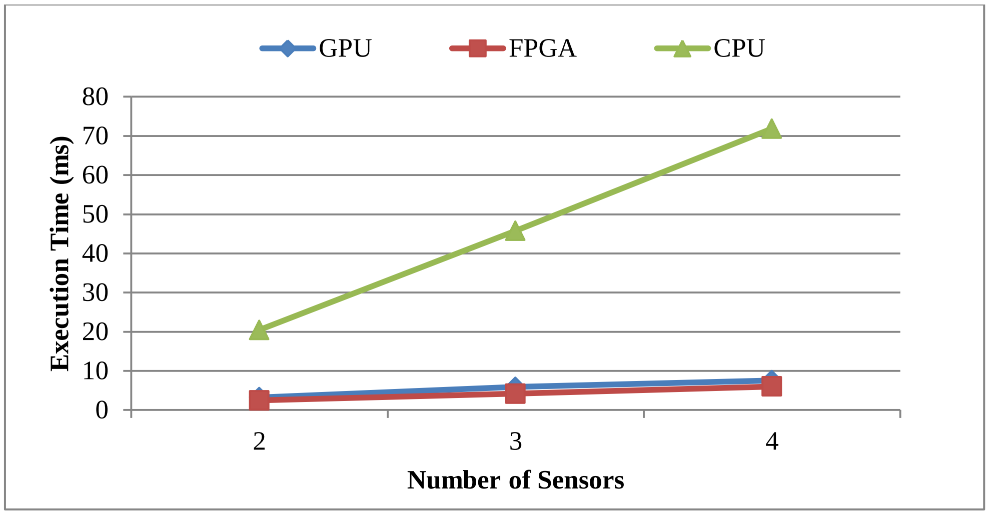


Figure 5-6: Effect of changing the number of sensors on execution time for different implementations.

5.4 Simulation of Parallel Extensive Cancellation Algorithm (ECA)

A real data is used to examine and verify ECA algorithm. The data were obtained from Radar Sensing Group of Electro-Science Laboratory, Department of Electrical Engineering, Ohio State University, USA [160]. The experimental FM PBR system focused on the use of TV station signals. However, the ECA based PBR approach can be applied to any other transmission sources such as cell-phone transmissions, radio waveforms, navigation satellites, and others.

The data were collected on long integration time in order to get an acceptable signal to noise ratio. The antennas were wideband hybrid log-periodic and bowtie antennas from ETS Lindgren (one for the reference and the other for surveillance). They were mounted on tripods on the roof of the Electro-Science Laboratory [160], at a height above ground of approximately 10 m.

The incoming digital television waveform was amplified and bandpass filtered. The spectrum for the real-sampled signal at 100 MHz represents frequencies from 500-550 MHz. The primary channel of interest is DTV ch 24, WSFJ-TV, from 530 to 536 MHz (DTV broadcasts have a 6 MHz bandwidth.)

The collected passive radar data were digitized for one second as there was no way of verifying whether the detections were spurious noise or actual target. The processing was segmented by independently processing the first half second of the data and the last half second of the data. Between these two data collections, if a target is present at the same range and Doppler, we can expect that the detection was successful.

Figure 5-7 shows the processing of the first half and the second half of the data in (a) and (b) respectively, whereas (c) represents the common detections between the two portions of the data (only the true target is shown).

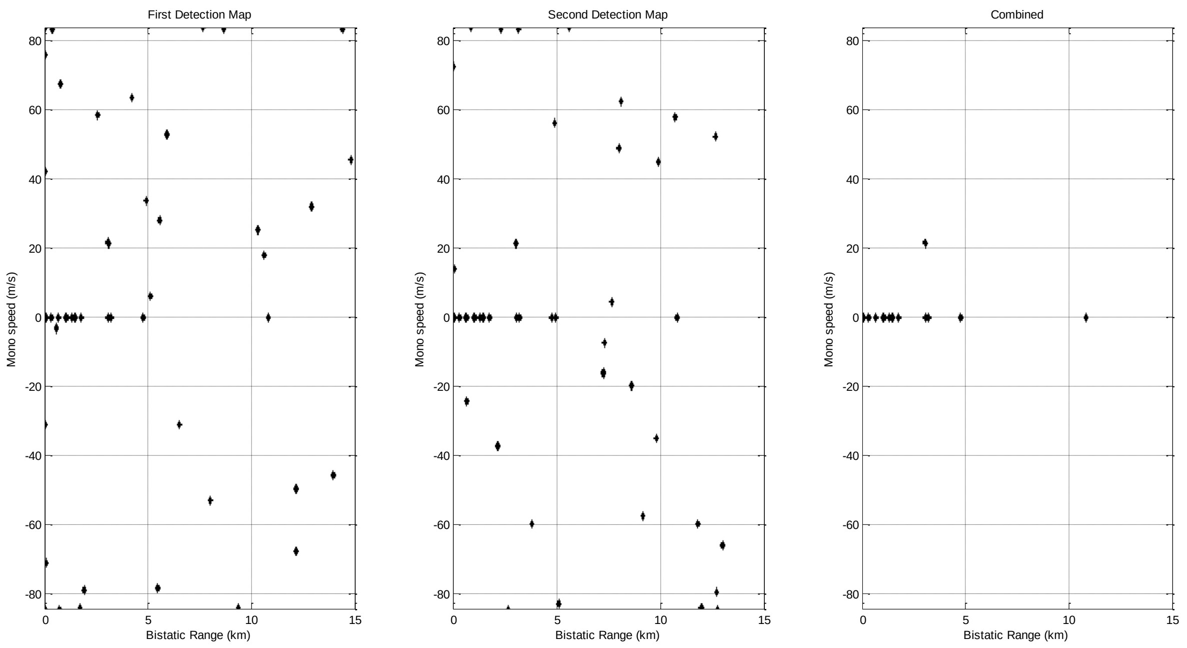


Figure 5-7: 2D-CCF after cancellation with ECA.

(a) Sketch of reference scenario. (b) Sketch of surveillance scenario.

(c) 2D-CCF after the cancellation with ECA.

Figure 5-8 shows the 2D-CCF between the reference and surveillance waveforms when the direct signal and all echoes from stationary scatters are cancelled. A target is detected at 22 m/s bistatic range rate and 4 km bistatic range where it is clearly visible since there are no sidelobes in either range or velocity dimension. It proves the validity of the modified ECA algorithm and shows the detection of a moving target.

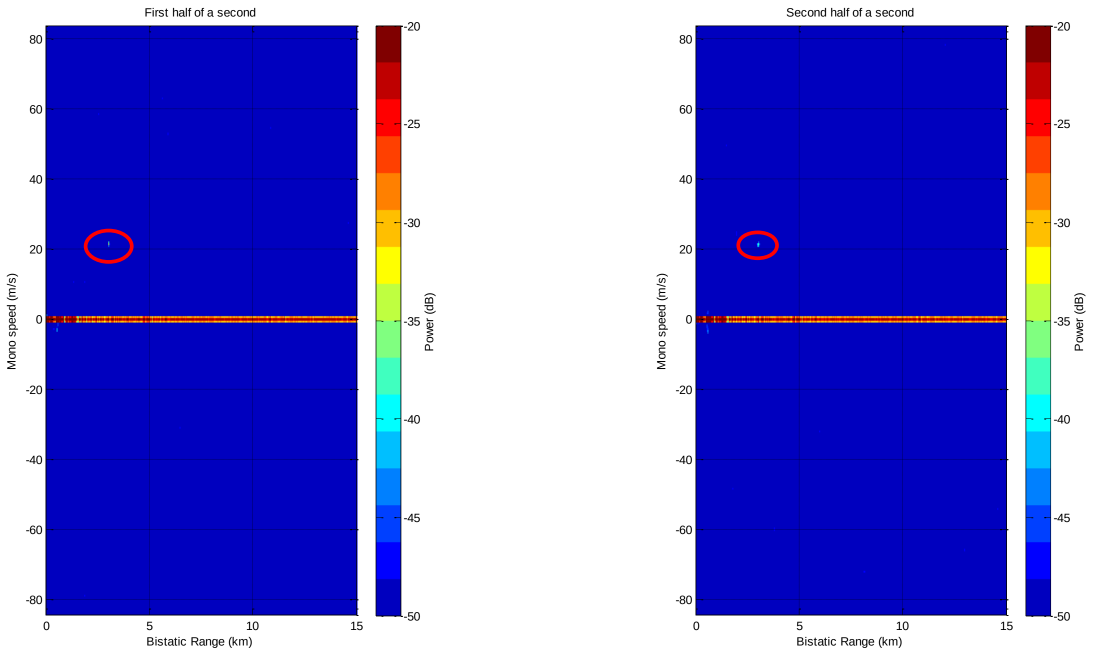


Figure 5-8: Target (circled) at (4,22) detected in ECA processing.

The software tool (RSPT) automatically generates the VHDL ECA component and synthesizes with the Xilinx ISE [159]. Xc6vlx760-2ff1760 FPGA device is used in this work. Table 5.8 lists the overall results in terms of hardware resources, and power consumption. The hardware utilization given in terms of occupied slices, Flip Flop (FF), Lookup table (LUT), BRAM_18K, IOBs, LUT FF pairs, DSP48E, and the power dissipation.

Table 5.8: Resource utilization ON Xc6vlx760-2ff1760 FOR N=1000 & K=32

Parameters	Resource Utilization
Number of Slice Register	422,311
Number of Slice LUTs	193,781
Number of Occupied Slices	104,148
Number of LUT FF pairs used	445,760
Number of IOBs	1186
Number of DSP48E1s	113
Power Consumption (mW)	4122

Tables 5.9 and 5.10 summarize the ECA major processes involved in ECA and illustrate the time complexity and memory usage before and after the optimization for each step.

Table 5.9: Time complexity of the major steps in ECA before and after optimization

Process	Before Optimization	After Optimization
Matrix D Building	O(N^2)	Removed
D^K	O(KN^3)	Removed
$D^K s_{ref}$	O(KN^2)	Removed
S_{ref} Matrix Building	O(KN)	O(K)
Matrix Λ Building	O($2PN^2$)	Removed
Matrix B Building	O(N^2)	Removed
Matrix Hermitian Building	$O(MN)$	Removed
Second part of Matrix X	$O(2P(N^2+KN))$	$O(P(N+K)$
Matrix X Building	O(NMN)	Removed
Complex matrix multiplication F=($X^H X$)	O(NM^2)	O(M^2)
Complex matrix inverse Z=$(X^H X)^{-1}$	Varies by used method	$O(CM)$
Complex matrix multiplication D=Z*X^H		
Matrix-vector multiplication Ds_{surv}	O(NM^2) $O(MN)$	$O(M)$
Matrix-vector multiplication $X\alpha$	$O(MN)$	$O(M)$
Vector-vector subtraction $s_{surv} - X\alpha$	$O(N)$	O(1)

Table 5.10: Storage complexity for the major steps in ECA before and after optimization

Process	Before Optimization	After Optimization
S_{surv} vector building	N	N
s_{ref} vector building	N+R-1	N+R-1
Matrix D Building	$(N+R-1)^2$	0
S_{ref} Matrix Building	(N+R-1)K	(N+R-1)K
Matrix Λ Building	$(2P+1)\ (N+R-1)^2$	0
Matrix B Building	N(N+R-1)	0
Matrix X Building	NM	NM
Matrix Hermitian Building	MN	0
Complex matrix multiplication (X^HX)	MM	MM
Complex matrix inverse $(X^HX)^{-1}$	MM	
Complex matrix multiplication $(X^HX)^{-1}*X^H$	MN	M
Matrix-vector multiplication $(X^HX)^{-1}X^Hs_{surv}$	M	M
Matrix-vector multiplication $X\alpha$	N	N
Vector-vector subtraction $s_{surv}-X\alpha$	N	N

The design of the control unit of a deep pipelined data-path that controls the scheduling for FPGA has 73 stages. The number of clock cycles, throughput, clock period, and the execution time from the start of execution until the final output is written for different number of range bins are shown in Table 5.11.

Table 5.11: Simulation time for different number of range bins

Parameter	Range bins (K)		
	32	48	64
Clock Frequency (MHz)	123	123	123
Clock period(ns)	8.12	8.12	8.12
Throughput (cycles)	5,788,1773	12,931,0344	24,0147783
Execution Time (s)	0.47	1.05	1.95

ECA is coded in C for its quick performance evaluation. Programs have been executed on a conventional PC powered by a 2.6 GHz i7-3720QM CPU with 8 GB RAM. The results of the execution times for i7 processor, our optimized FPGA, and GPU implementations are summarized in Table 5.12. Execution time is plotted in Figure 5-9 to show the effect of varying the number of range bins on the performance. The results show that the FPGA implementation performs better than other implementations. The superior performance of the FPGA-based implementations is attributed to the highly parallel and pipelined architecture, and the flexibility in allocating the needed resources.

Table 5.12: Execution time of different platforms of ECA

Implementation	Range bins (K)		
	32	48	64
CPU before Optimization (s)	32.34	45.2	60.21
CPU After Optimization (s)	8.52	13.05	20.8
GPU (s)	0.98	1.457	2.377
FPGA (s)	0.47	1.05	1.95

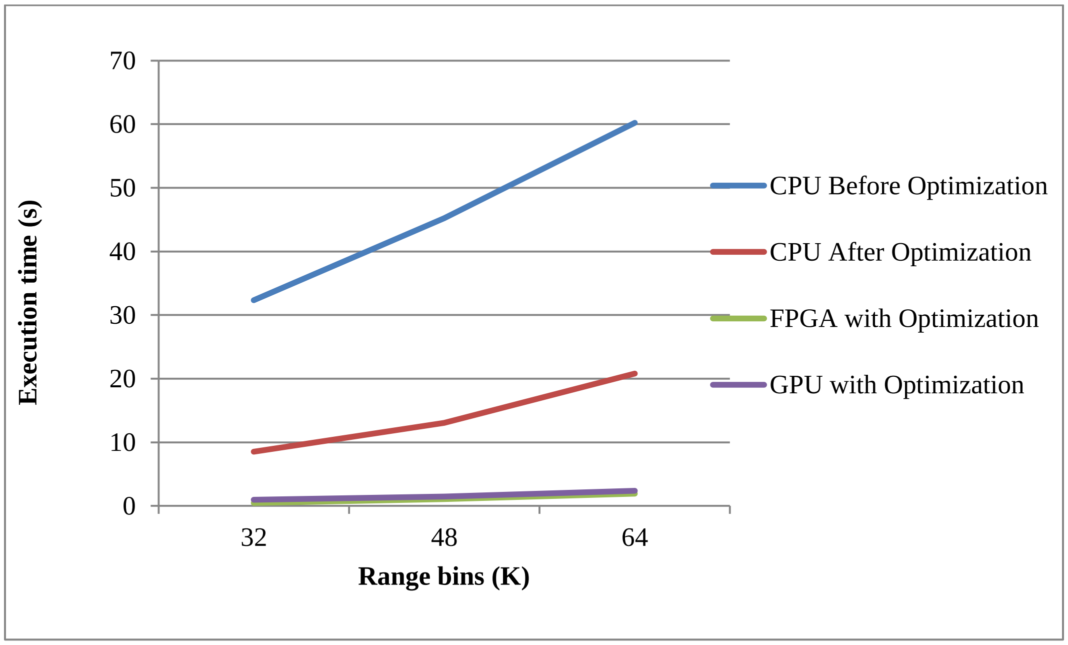


Figure 5-9: Effect of changing range bins (K) on execution time for different implementations.

5.5 Simulation of Parallel Block Compressive Sampling Matching Pursuit (BCoSaMP) algorithm

A numerical experiment is performed in order to examine and verify BCoSaMP algorithm. The experiment is accomplished using a signal with length N=4096 from noise-free random Gaussian numbers. The block size is 32 and numbers of active blocks are 16. The setup of our experiment is important for microphone applications where sparse acoustic signal frequencies have different amplitudes and delays. The recorded signals from the microphone are digitized and re-constructed in a way to match the structure of blocking technique for BCoSaMP algorithm. However, the BCoSaMP algorithm can be applied to any other sources such as image processing applications.

Figures 5-10 and 5-11 show the original block sparse signal and the result of recovering the signal from BCoSaMP algorithm, respectively. We observe that BCoSaMP algorithm can recover the original sparse signal accurately.

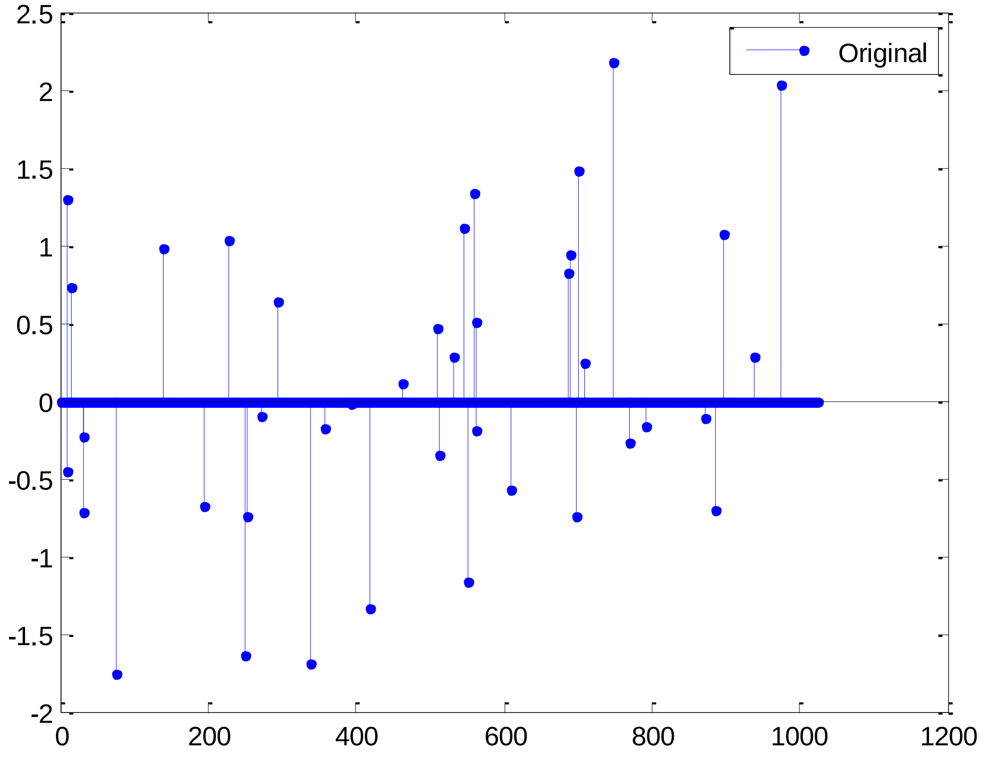


Figure 5-10: Original block-sparse signal

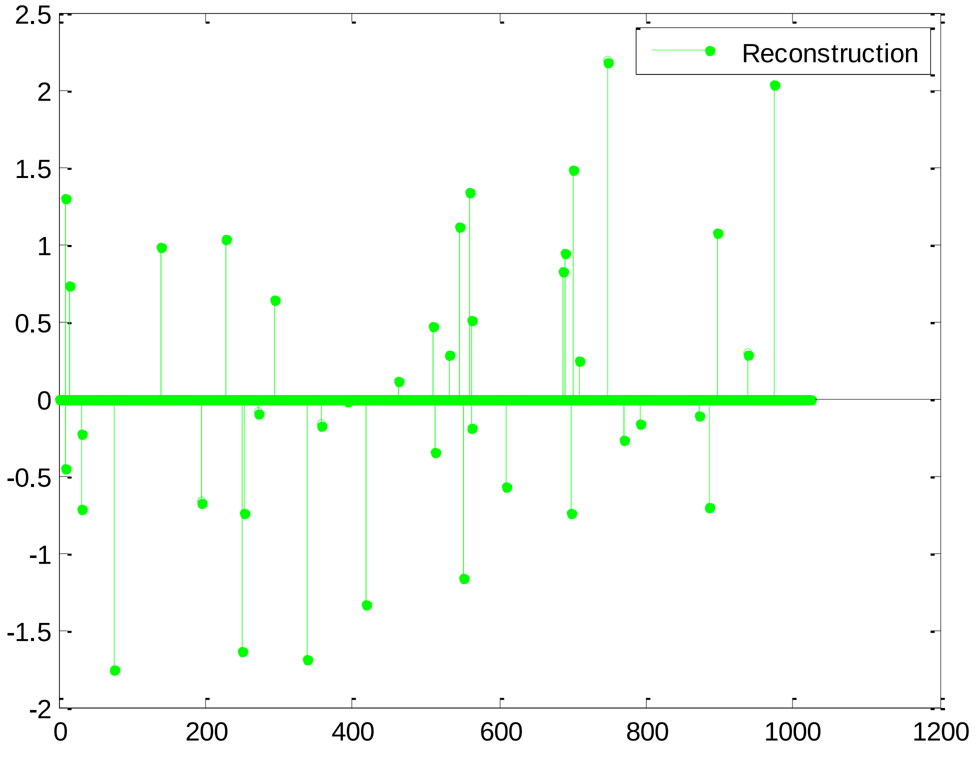


Figure 5-11: BCoSaMP-based recovery signal.

RSPT tool automatically generates the VHDL BCoSaMP component and synthesizes with the Xilinx ISE [159]. XC7A100TCSG324-3 FPGA device is used in this work. Table 5.13 lists the overall results in terms of hardware resources and power consumption. The hardware utilization is given in terms of slices of BRAM_18K, DSP48E, Flip Flop (FF), Lookup table (LUT), IOBs, and the power dissipation.

Table 5.13: Resource utilization and overall implementation performance of BCoSaMP ON Artix7 xc7a100tcsg324-3

Parameters	Resource Utilization
BRAM_18K	14
DSP48E	42
FFs	8753
LUTs	10884
Number of IOBs	140
Power Consumption (mW)	1960

The design of the control unit of a deep pipelined data-path that controls the scheduling is 57 stages. The number of clock cycles, throughput, clock period, and the execution time from the start of execution until the final output is written for different number of vector size are shown in Table 5.14.

Table 5.14: Simulation time comparison of different number of vector size with K=50 and Number of blocks =5

Parameter	Density (Number of samples)		
	256	512	1024
Clock Frequency (MHz)	117.64	117.64	117.64
Clock Period(ns)	8.5	8.5	8.5
Throughput (cycles)	41025	71793	116304
Execution Time (ms)	0.3487	0.61	0.988

The execution times of BCoSaMP algorithm for the sequential, FPGA, and GPU implementations are summarized in Table 5.15. The results show that the FPGA and GPU implementations perform much better than the alternative sequential implementation. The superior performance of the FPGA-based implementations is attributed to the highly parallel and pipelined architecture. The result also shows the effect of changing the vector size on the performance. It can also be seen that our optimized implementations achieves more speed-up with increasing vector size which is attributed to the high parallelism and pipelining exploited in the array architecture as opposed to the sequential behavior implementation.

Table 5.15: Execution time of different implementations for BCoSaMP

Implementation	Density (Number of blips)		
	256	512	1024
Before Optimization (ms)	3.1	6.39	14.1
Optimization with FPGA (ms)	0.3487	0.61	0.988
Optimization with GPU (ms)	0.442	0.76	1.32

5.6 Simulation of Parallel Discrete Wavelet Transform (DWT)

The generated VHDL DWT file from our software tool is evaluated and synthesized with the Xilinx tool ISE [159]. XA7A100T CSG324 -2I FPGA part is used in our work. The hardware resources in terms of occupied slices, performance in terms of maximum frequency, and power dissipation of our generated VHDL file for all dimensions for size N=128 are shown in Table 5.16. Our implementation provides significant improvement with respect to the maximum frequency whereas a slight increase in number of slices and the power consumption.

Table 5.16: Hardware Utilization and Overall Performance for N=64.

1-D				
Parameters	Forward Type		Backward Type	
	Non-Optimized	Optimized	Non-Optimized	Optimized
Slices	1646	2247	1022	1110
Power (mW)	96	101	83	83
Max. Fr.(MHz)	116.12	214.6	118.55	226.55
2-D				
Slices	2525	2654	1250	1353
Power (mW)	218	249	189	203
Max. Fr.(MHz)	113	210	113.66	218
3-D				
Slices	2890	3159	1896	2064
Power (mW)	361	404	294	328
Max. Fr.(MHz)	106.17	203	104	204.45

The DWT performance is compared with other implementations and platforms. The achieved throughput of Pentium III processor, DSP, and the optimized FPGA implementation are shown in Table 5.17. The result shows that FPGA implementation achieves high performance compared with other implementations and platforms. This is due to the highly parallel and pipelined architecture provided by the FPGA implementation of DWT and the efficiency of our developed software processing tool.

Table 5.17: Throughput of Different Implementations for 2-D for N =128.

Platform	Forward (MHz)	Backward (MHz)
Pentium III	0.00781	0.00673
TMS320C6711 DSP	6.530	3.620
Conventional one	113	113.66
Our Method	210	218

To underline the influence of different transform size on area, power consumption and maximum frequency, we have implemented the optimized design that generated from our tool on Xilinx FPGA devices, Xa7a100t-2icsg324. Figures 5-12, 5-13, and 5-14 illustrate the relationship for each performance indicator for backward type for 1-D, 2-D, and 3-D. The results obtained are clearly shown that the 3-D consumes more area and power than 1-D and 2-D due to its high computation and complexities.

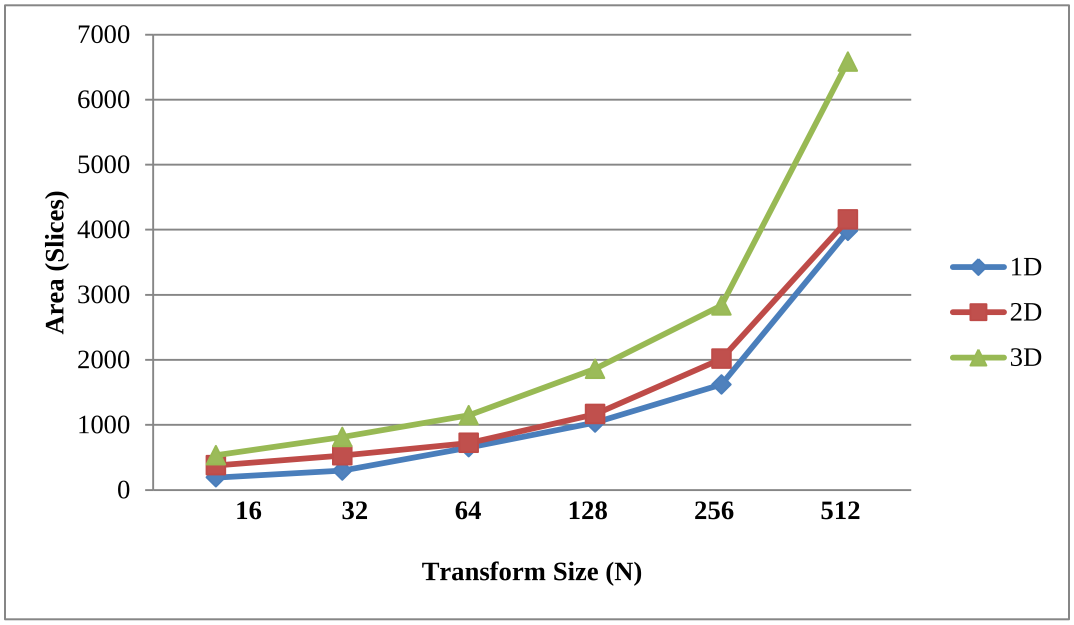


Figure 5-12: Influence of transform size on area (slices).

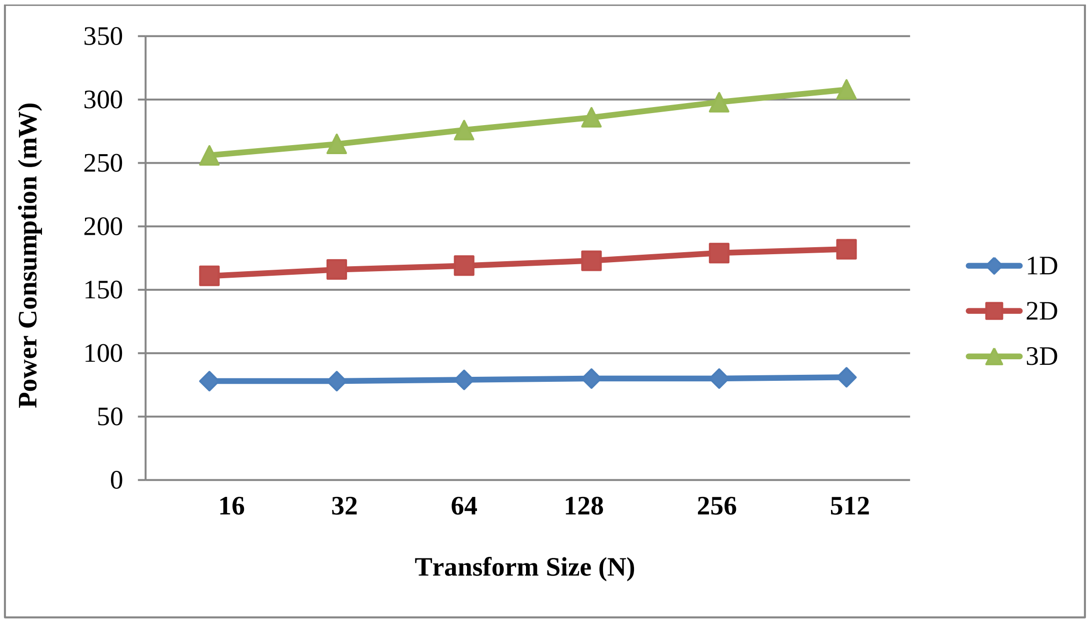


Figure 5-13: Influence of transform size on power consumption (mW).

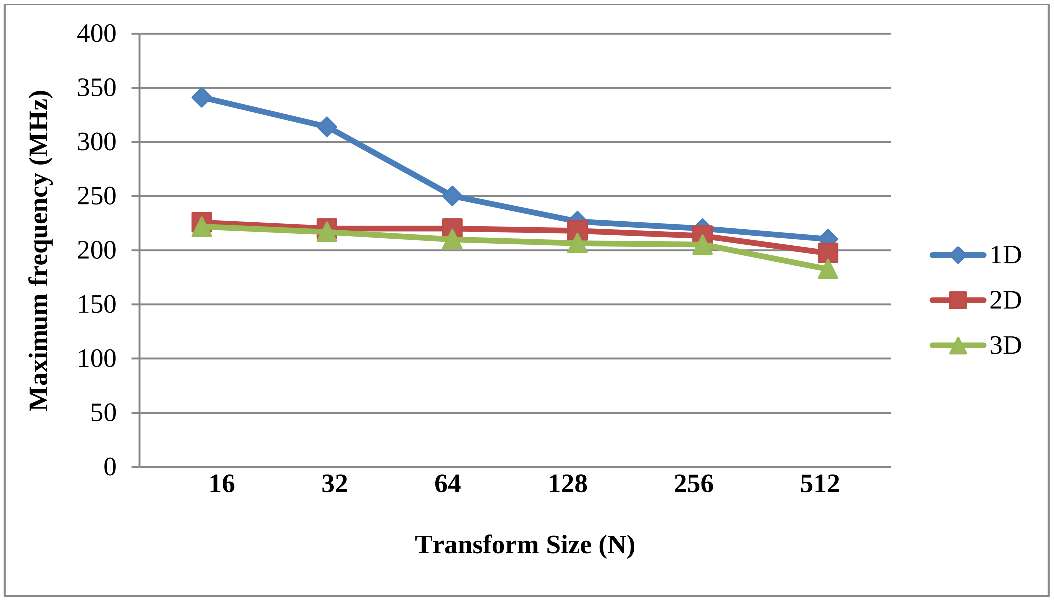


Figure 5-14: Influence of transform size on maximum frequency (MHz).

5.7 Simulation of Parallel Particle Filter

In order to examine and verify particle filter method, it must be tested against highly non-linear and non-Gaussian data. So, complex system with difficult state estimation is considered in our work in both the process and measurements based on Equations (3.50) and (3.51). Figures 5-15 and 5-16 show the particle filter estimation performance and the error rate over the true state respectively. It shows that the estimating state is close to the true states which validate the efficiency of particle filter operation.

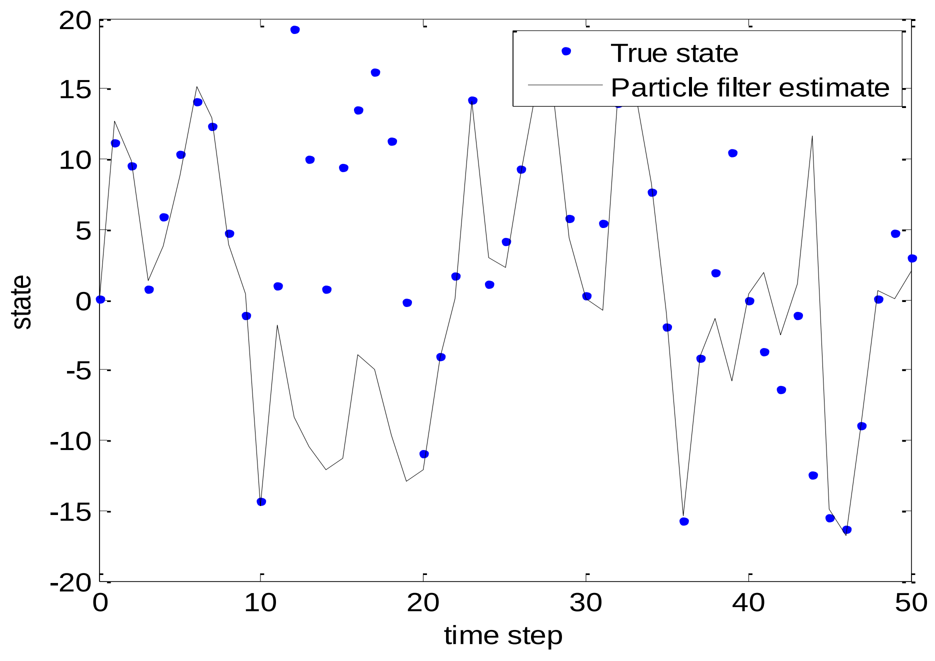


Figure 5-15: Particle filter estimation performance

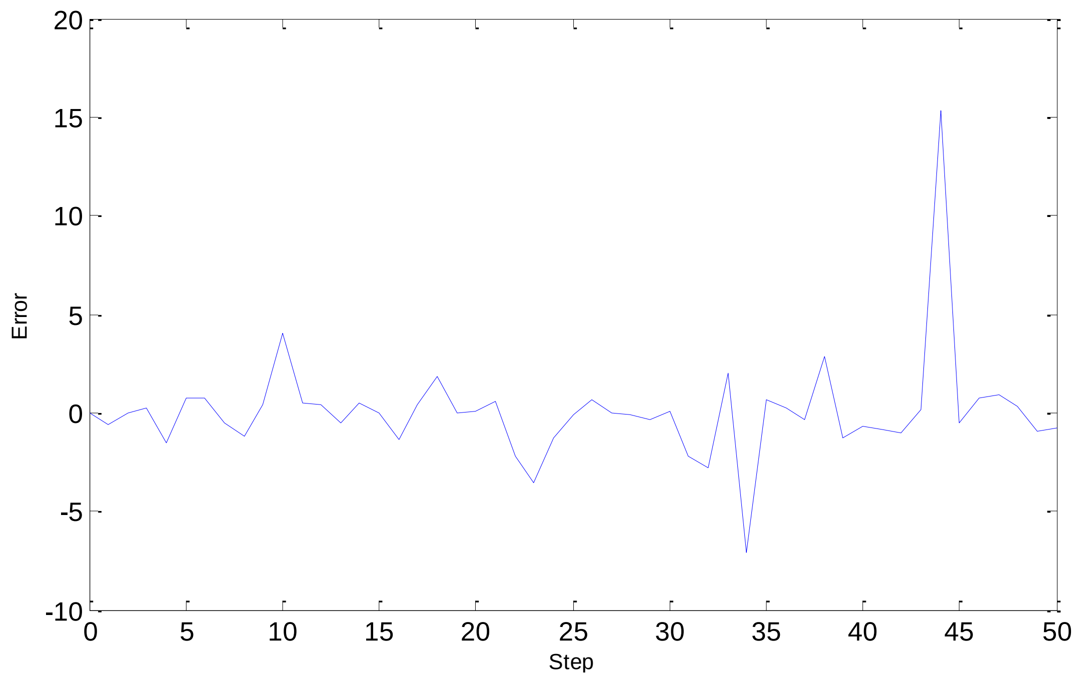


Figure 5-16: Error rates of 100 particles over 50 time step

The particle filter component is synthesized with the Xilinx ISE [159]. XA7A100TCSG324-1q FPGA device is used in this work. Table 5.18 lists the overall results in terms of hardware resources and power consumption.

Table 5.18: Resource utilization and overall implementation performance

Parameters	Resource Utilization
BRAM_18K	8
FFs	8724
LUTs	15644
Number of IOBs	230
Power Consumption (mW)	2434

The design of the control unit of a deep pipelined data-path that controls the scheduling has 101 stages. The number of clock cycles, throughput, clock period, and the execution time from the start of execution until the final output is written for different number of vector size are shown in Table 5.19.

Table 5.19: Simulation time comparison of different number of particles

Parameter	Number of particles		
	250	500	1000
CLK Freq. (MHz)	145.34	145.34	145.34
Clock Period(ns)	6.88	6.88	6.88
Throughput (cycles)	410325	710773	1100800
Execution Time (ms)	2.823	4.89	7.57

The execution times of particle filter implementation for the sequential, FPGA, and GPU implementations are summarized in Table 5.20. The results show that the optimized

FPGA and GPU implementations perform much better than un-optimized one. The superior performance of the optimized implementation is attributed to the exploitation of parallel architecture of the FPGA and the parallelization of the particle filter. The result also shows that the optimized implementation achieves more speed-up with increasing number of particle which is attributed to the high parallelism and pipelining exploited in the array architecture.

Table 5.20: Execution time of different implementation

Implementation	Number of particles		
	250	500	1000
Before Optimization (ms)	21.93	43.94	87.5
Optimization with FPGA (ms)	2.823	4.89	7.57
Optimization with GPU (ms)	3.07	6.48	11.86

5.8 Simulation of Parallel Model Based- Iterative Hard Thresholding (MB-IHT)

A numerical experiment is performed in order to examine and verify MB-IHT method. The experiment is accomplished of a signal with length N=1024 from noise-free random Gaussian. The number of measurements is 240 and the number of active blocks is 40. Figures 5-17 and 5-18 show the original block sparse signal and the result of recovering the signal using the MB - IHT method. Figure 5-19 shows the error rate of reconstruction the sparse compressed signal. It shows that the MB-IHT method can recover the original sparse signal accurately which validate the efficiency of the algorithm.

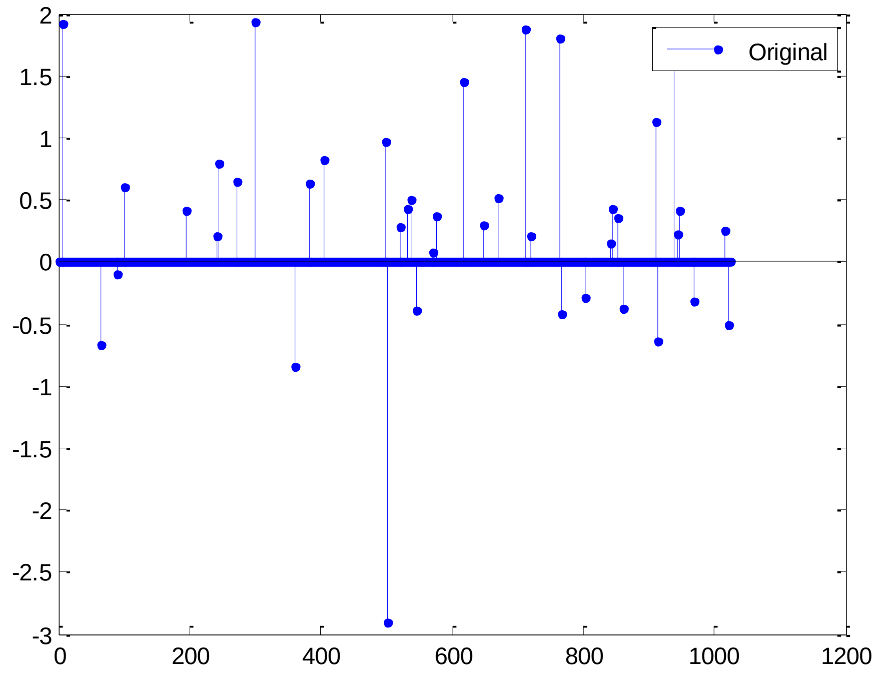


Figure 5-17: Original block-sparse signal

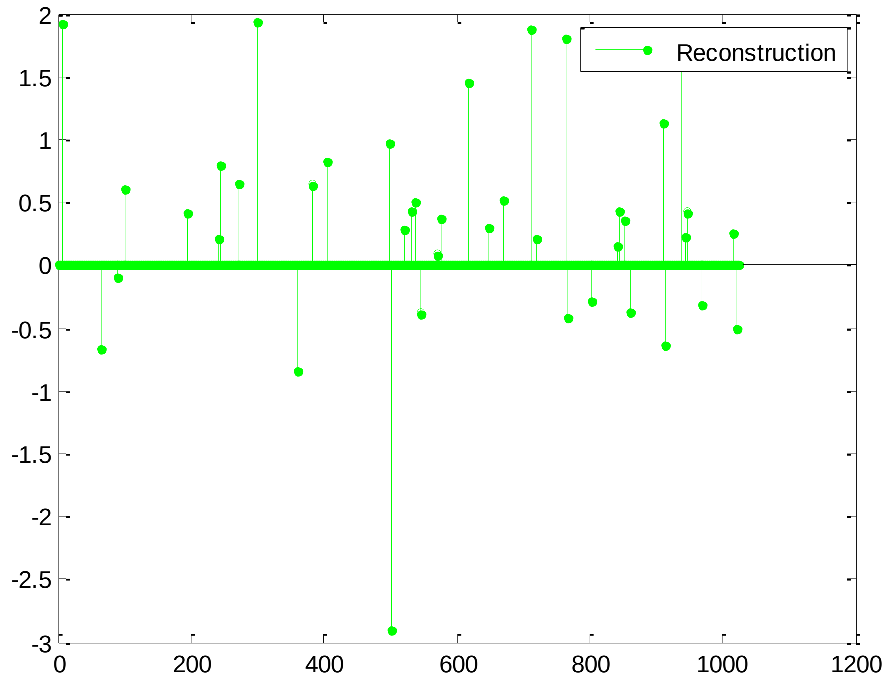


Figure 5-18: MB-IHT based recovery.

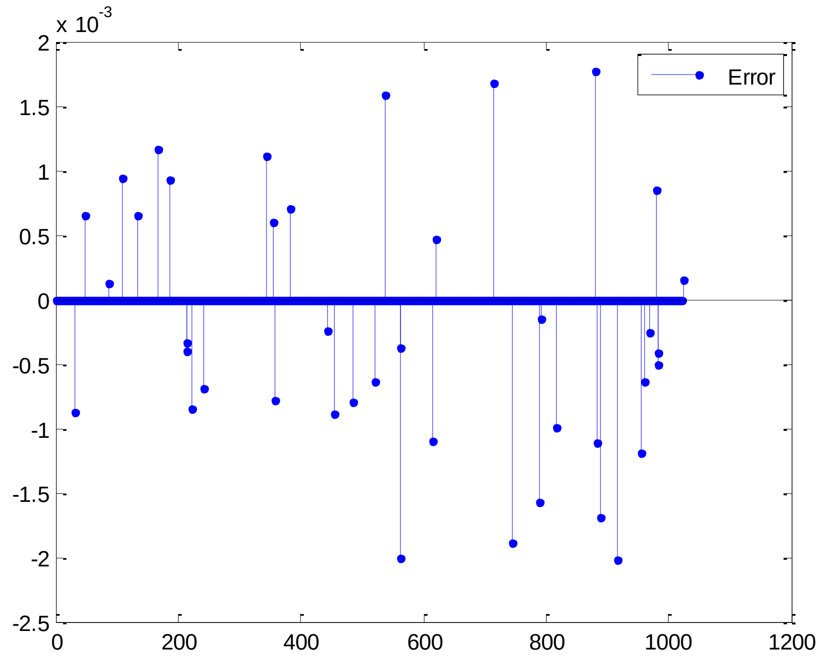


Figure 5-19: Error rates of recovering compressed sparse signal using MB-IHT.

The MB - IHT method has been implemented and synthesized on xc7a100tcsg324-3 FPGA device. Table 5.21 lists the overall results in terms of hardware resources and power consumption. The hardware utilization is given in terms of slices of BRAM_18K, DSP48E, Flip Flop (FF), Lookup table (LUT), IOBs, and the power dissipation.

Table 5.21: Resource utilization and overall implementation performance on Artix7 xc7a100tcsg324-3

Parameters	Resource Utilization
BRAM_18K	4
DSP48E	8
FFs	593
LUTs	805
Number of IOBs	140
Power Consumption (mW)	138

The design of the control unit of a deep pipelined data-path that controls the scheduling is 57 stages. The number of clock cycles, throughput, clock period, and the execution time from the start of execution until the final output is written for different number of vector size are shown in Table 5.22.

Table 5.22: Simulation time comparison of different number of vector size with K=25

Parameter	Density (Number of samples)		
	256	512	1024
CLK Freq. (MHz)	125	125	125
Clock Period(ns)	8.0	8.0	8.0
Throughput (cycles)	27861	55501	110700
Execution Time (μs)	222	441	885

The execution times of the MB - IHT method for the sequential, FPGA, and GPU implementations are summarized in Table 5.23. The results show that the optimization with FPGA achieves much better than the alternative implementations. The superior performance of the FPGA-based implementations is attributed to the highly parallel and pipelined architecture. The result also shows the effect of changing the vector size on the performance; which shows that our optimized implementations is achieved more speed-up with increasing vector size which is attributed to the high parallelism and pipelining exploited in the array architecture as opposed to the sequential behaviour implementation.

Table 5.23: Execution time of different implementations on Artix7 -3 (XC7A100T CSG324 -3)

Implementation	Density (Number of blips)		
	256	512	1024
Before Optimization (μs)	437	856	1693
Optimization with FPGA (μs)	222	441	885
Optimization with GPU (μs)	301	535	1050

5.9 Data Storage Location Analysis and Optimization

Placing data at different storage resources on FPGA devices such as look-up tables, internal block RAM, and external memory will highly affect the overall performance such as latency, power dissipation, hardware resources, and the total area. So, a careful consideration must be taken in the way of storing and implementing the array and matrices. For example, if the data elements in the vector or matrices need to be accessed only one time, then the efficient implementation is to store these data in a block RAM to take the advantages of the efficiency of memory architecture. On the other hand, when the data elements need to be accessed simultaneously to support some optimization techniques to improve the performance, then these elements must be stored through internal configurable logic blocks. This will help in improving the performance but it loses the efficient architecture of RAM and increases power dissipation, hardware resources, and the area. So, it will consume large logic resources that FPGA uses for other logical and mathematical operations.

Also, the size of data elements that need to be stored has a critical factor in placing them at different places. In order to store the data through configurable logic blocks, the logic blocks such as Look-Up Table (LUT) and Flip Flop (FF) need to be configured and wired together. So, with large number of data elements, the implementation will be not efficient as it causes high wiring delays and uses a large number of multiplexers. So, in this case, the Block RAM will be utilized to reduce the latency, hardware resources, and power dissipation. On the other hand, with a small number of elements, the configurable logic block will be an appropriate option as the block RAMs has fixed modules in terms of size. So, if you map small number of elements on the block RAM, then it will waste the rest of the space in RAM.

Moreover, both distributed memory through configurable logic blocks and block RAMs are different in the way they are operated. Obviously the memory has two ports; write and read. Both of them are synchronous in writing operation whereas distributed memory is asynchronous and the block RAMs are synchronous in the reading operation. The advantage of the asynchronous feature of the configurable logic blocks in reading operations is the possibility of reading the data from memory as soon as the address is given without waiting for the clock edge. However, the synchronous operations can only happen at the clock edge.

In summary, the distributed reconfigurable logic blocks are used in the following cases:

- Requiring multiple accesses to the data elements in the same clock cycle by applying optimization techniques to improve the performance. This will help in executing some operations concurrently.
- The amount of data required to be stored is smaller.
- Not enough free embedded block RAM on the FPGA for storage.

However, the blocks RAMs of FPGA are used in the following cases:

- Requiring only one access of the data in the same clock cycle.
- The amount of data required to be stored is large.
- Not enough free reconfigurable logic blocks available on the FPGA.

In order to explore the performance of placing data at various memory location and architecture, we take 2-D discrete wavelet transform method as a case study with dimension N=128. So, its matrix (128x128) is implemented through both reconfigurable logic blocks and block RAM. Reconfigurable logic blocks based implementation avoids any contention on the matrix resource where many elements can be accessed simultaneously. It was also implemented as dual ports block RAM for comparing purposes. Table 5.24 shows the overall performance of running 2-D Forward type DWT for size N=128 in terms of Flip Flop (FF), Look Up Table (LUT), and the power consumption for both implementations. Xa7a100tfgg484-2i FPGA device is also used for both of them. Table 5.24 shows that reconfigurable logic blocks based implementation requires a lot of hardware resources that used and the design will be large due to use a lot of multiplexers; consequently the power consumption, area, and design complexities are increased dramatically. This is because the matrix has a large data element that is not appropriate for reconfigurable logic blocks.

Table 5.24: Hardware resources and power consumption of placing data at various memory locations for 2-D DWT

Hardware resource	Dual port block RAM	Reconfigurable logic blocks as LUT	Percentage of increased
Throughput (MHz)	113	210	86%
LUT	1642	10778	556%
FF	835	9548	1043%
Power dissipation (mW)	246	2031	725%

Also, we take another large example to show the impact of each option of storing the data. BCoSaMP algorithm is fully explored of placing data at both look up tables and block RAM. In this algorithm, we consider four vectors each with size 50 elements, one vector with size 256, two vectors each with size 30, and matrix with size 10x10. Using reconfigurable logic blocks option, Xilinx tool gives warning for the vectors with size 50, vector with size 256, and the matrix one. Sizes of these vectors were too large for implementation with LUTs. If warning is ignored then it requires long runtime and suboptimal Quality of Results (QoR). It is due to use of large numbers of multiplexers. Table 5.25 shows how much the hardware resources and power dissipation is increased with the option of reconfigurable logic blocks as opposed to block RAM option for only the vector with size 256 and the two vectors with size 30.

Table 5.25: Hardware resources and power consumption of placing data at various memory locations for BCoSaMP algorithm

Hardware resource	Dual port block RAM	Reconfigurable logic blocks as LUT	Percentage of increased or decreased
Latency (ms)	3.1	0.3487	Decreased 88%
Block RAM	14	10	Decreased 28%
LUT	12763	23202	Increased 82%
FF	11516	31798	Increased 176%
Power dissipation (mW)	2438	5505	Increased 126%

Our goal is to achieve low latency without increasing other critical performance parameters dramatically such as power dissipation, hardware resources, area, and complexity. So, a careful implementation and decision for each vector and matrix in all our algorithms (D^3, ECA, particle filter, DWT, IHT, and BCoSaMP) are made based on above mentioned cases. Sometimes when a parallel operation is performed on a large matrix such as in the case of DWT, multi ports RAM is used instead of reconfigurable logic blocks. This helps in neglecting high resource requirement and storage overhead to minimize the area, cost, power consumption, and design complexities.

Also, the implementation of some algorithms on FPGA devices such as ECA algorithm requires an external memory to store the data where the capacity of internal FPGA memory is not enough. This is very critical as it dictates the overall performance of the system which deals with I/O ports that have high latency. The interface must provide high bandwidth for both read and write operations to keep up with the flow of the data. So, to minimize the low performance and the overhead of the interface, it is only used with the following cases:

- Not enough space to store the data on FPGA internal memory.

- At the first time where the input data need to be fetched from external memory to be stored on FPGA internal memory.
- At the last time where the result data need to be stored back to the external memory.

5.10 Vivado HLS Tool versus VHDL Programming

The most limitations with FPGA platform is the requirements of large design time and high development of the algorithm. However, this limitation is removed by using Vivado HLS tool. In order to show the effectiveness of HLS tool, an extensive analysis of matrix vector multiplication task is performed since it is used mostly in all the algorithms. The matrix vector multiplication task was written in standard VHDL and synthesized. It was then compared with results from HLS tool. Both implementations are synthesized and simulated on the same device and FPGA part (Vertix7 XC7VX330T FFG1157-2). Table 5.26 shows the performance in terms of the achieved throughput, LUT, FF, IOBs, and the power dissipation. The result shows that VHDL implementation based on HLS tool achieves better throughput and IOBs than manual implementation. On the other hand, LUTs and FFs are slightly increased where the power dissipation almost the same. However, our objective is to achieve high throughput without increasing other performance parameters dramatically. So, HLS tool is used since it achieves high performance in terms of throughput, simplifies the design and simulation tasks, reduces the design time, increases the productivity, improves the reliability, and enables exploration of the design space. It also enables the designer to build the most efficient implementation in terms of performance with given design constraints. Therefore, the

HLS tool should be considered as a new paradigm in FPGA design especially for high performance applications.

Table 5.26: HLS tool based implementation vs manual implementation of matrix vector multiplication task

Performance parameters	HLS tool	VHDL
Throughput (MHz)	418	176
IOBs	100	242
FF	225	113
LUT	227	202
Power dissipation (mW)	148	143

5.11 Conclusions

A parallel algorithm has been developed for IR video processing on GPU with all its steps; background subtraction, noise filtering and connected component labelling. It provides a frame processing rate of 45.126 frames per second meeting the real time requirement of 30 fps. Complete analysis and estimation of energy consumption for all the MPSoC platform components such as processing elements, memory, caches, routers and communication architecture were performed to find the bottlenecks in the platform for video processing application. The video processing algorithm was also partitioned, parallelized, mapped and scheduled on multi-core. We showed that the energy dissipation appears to be the most critical factor for memory and caches not for the communication architecture as per common belief. Also, a better performance is obtained by proposing a new modelling and simulation approach regarding the channel width and buffer sizing. D^3, ECA, particle filter, DWT, IHT, and BCoSaMP algorithms are firstly simulated and experimented for verification purposes. Then, all the algorithms implemented and parallelized on FPGA and GPU based architecture. Experimental results show that our FPGA and GPU architectures of these algorithms can significantly outperform an equivalent sequential implementation. The results also show that our FPGA implementation provides better performance than the GPU implementation.

Chapter 6

Conclusion and Future Work

A parallel processing approach using GPU platform is used to process IR video data that will meet real time requirements. The IR video processing algorithm including all its tasks; background subtraction, noise filtering and connected component labeling; were partitioned, parallelized, mapped and scheduled. We achieved the real time requirements and the necessary performance for analyzing the IR image of size 704x480.

We have analyzed and assessed the energy dissipation for heterogeneous NoC-based MPSoC platform running a video application. We have estimated the energy consumption for all the components of heterogeneous NoC platform including processing elements, memory, caches, routers and communication architecture. The video processing algorithm was partitioned, parallelized, mapped and scheduled on Multi-Core (slave cores). We showed that the energy dissipation appears to be the most critical factor for memory and caches not for the communication architecture as per common belief. Also, a new modeling and simulation approach regarding the channel width and buffer sizing is proposed to get a better performance. We showed that there are some hot spots in the system regarding the channel width and buffer size. They have been optimized to get a better performance.

D^3, ECA, particle filter, DWT, IHT, and BCoSaMP algorithms have been transformed into parallel architectures. They have been implemented on both FPGA and GPU platforms. A generalized approach for parallelizing a target algorithm has been developed. It is accomplished by creating a methodology for various processes such as evaluation of data dependencies and exploring parallel processing opportunities. The performance evaluation of placing data at various memories such as cache, look up tables, SRAMs, and external memories has been explored. It shows high impact on the overall performance such as latency, power dissipation, hardware resources, and the total area. So, a careful consideration was taken in the way of array and matrices implementation of a certain algorithm which depends on many factors such as:

- Number of required access of data elements in the same clock cycle.
- The size of the vector and the matrix need to be stored.
- The available storage area on embedded block RAM on the FPGA.

High Level Synthesis Tool (HLST) has been exploited with our techniques by enabling rapid development to generate efficient parallel codes from high-level problem descriptions. The HLST automatically generate the pipelined data-path and the control unit which significantly simplifies the design and simulation process. It significantly reduces the design time of FPGA-based hardware, enables design exploration, and therefore, should be considered as an alternative in FPGA design for complex applications.

A new software tool called Radar Signal Processing Tool (RSPT) has been developed. It unifies the aspects of algorithms, architectures, and software which bridges the gap between the algorithm and architecture scientific communities. This helps in performing hardware software co-design that pushes performance and energy efficiency while reducing cost, area, and overhead. It is capable of auto-generating a fully optimized VHDL representation of each processing approach with different parameters through a Graphic User Interface (GUI). It provides the designer a feedback on different performance parameters such as the number of occupied slices, maximum frequency, and dynamic range performance. Using this feedback, the designer can focus on the overall SoC performance and make adjustments to any of these components as necessary until the desired performance of the overall SoC is achieved. This provides great flexibility in designing signal processing applications such as D^3, ECA, particle filter, DWT, IHT, and BCoSaMP algorithms for a SoC without having to write a single line of VHDL code. RSPT also utilizes optimization techniques such as pipelining, code in-lining, loop unrolling, loops merging, and dataflow techniques by allowing the concurrent execution of operations to improve throughput and latency.

Following are several issues which need to be considered in order to improve and expand our work:

- Although our selected algorithms for radar and compressive sensing are the interesting topic of many researchers in recent years, there are many other algorithms that are still needs to be developed of different fields.
- The results of the selected algorithms are obtained through simulations. However, it is more complementary if these algorithms are implemented and simulated on real hardware. This will be interesting for researchers in different fields.
- Our selected algorithms are implemented based on different programming languages such as C, C++, Vivado Syntax, Matlab. However, the development of these algorithms in a deeper level based on circuit level based on VLSI research will be very helpful. This takes a long time for designing, developing, testing, and verification, but it will improve the performance in terms of latency, power dissipation, and storage area.
- Although many algorithms are implemented efficiently on different parallel processing platforms to reach the real time requirements, it is more desirable to develop and implement these algorithms based on system level not on component level. This will include a complete system consisting of transmitter phase with its antenna, RF oscillator, modulator etc, receiver phase with its local oscillator, amplifier, limiter, detector, Analog to Digital Converter (ADC) etc, input stage design, storage phase, signal processing design with its filter, integration, compression etc, and output stage design. This will help in linking all the application requirements resources and

therefore more optimization will be achieved since several optimization techniques will be applied for all its parts not just for the computational part.

References

1. A. V. Oppenheim and R. W. Schafer, Discrete-Time Signal Processing. Englewood Cliffs, NJ: Prentice-Hall, 1998.
2. M. A. Richards, Fundamentals of Radar Signal Processing. New York: McGraw-Hill, 2005.
3. J. P. Costas, “A study of a class of detection waveforms having nearly ideal range-Doppler ambiguity properties,” Proc. IEEE, vol. 72, pp. 996–1009, Aug. 1984.
4. L. Xu, J. Li, and P. Stoica, “Target detection and parameter estimation for MIMO radar systems,” IEEE Trans. Aerosp. Electron. Syst., vol. 44, no. 3, pp. 927–939, Jul. 2008.
5. Q. He, R. Blum, H. Godrich, and A. Haimovich, “Target velocity estimation and antenna placement for MIMO radar with widely separated antennas,” IEEE J. Sel. Topics Signal Process., vol. 4, no. 1, pp. 79–100, Feb. 2010.
6. D. W. Bliss and K. W. Forsythe, “Multiple-input multiple-output (MIMO) radar and imaging: Degrees of freedom and resolution,” in Proc. 37th IEEE Asilomar Conf. Signals, Systems, Computers, Nov. 2003, vol. 1, pp. 54–59.

7. Amira, A., Chandrasekarana, S., Montgomery, D.W. & Uzunb, I.S. (2008) A segmentation concept for positron emission tomography imaging using multiresolution analysis. Neurocomputing , vol.71 (2008), pp. 1954– 1965.

8. Bernard, O. & Friboulet, D. (2009) Fast medical image segmentation through an approximation of narrow-band B-spline level-set and multiresolution. IEEE International Symposium on Biomedical Imaging: From Nano to Macro, pp. 45 – 48.

9. European Telecommunications Standards Institute. Digital broadcasting system for television, sound, and data services. ETS 200 421, 1994.

10. European Telecommunications Standards Institute. Digital video broadcasting (DVB); interaction channel for satellite distribution systems; ETSI EN 301 790 V1.2.2 (2000-12), 2000.

11. European Telecommunications Standards Institute. Digital video broadcasting (DVB) second generation framing structure, channel coding and modulation systems for broadcasting, interactive services, news gathering and other broadband satellite applications. DRAFT EN 302 307 DVBS2-74r15, 2003.

12. Johnson, D. H.; Dudgeon, D. E. (1993). Array Signal Processing. Prentice Hall.

13. Krim, H.; Viberg, M. (July 1996). "Two Decades of Array Signal Processing Research". IEEE Signal Processing Magazine: 67–94. Retrieved 8 December 2010.

14. S. Haykin and K.J.R. Liu (Editors), "Handbook on Array Processing and Sensor Networks", Adaptive and Learning Systems for Signal Processing, Communications, and Control Series, 2010.

15. Ibm, cell broadband processor, http://www.ibm.com/developerworks/power.

16. Intel 80 core chip, http://techresearch.intel.com/articles/Tera-Scale/1449.htm.

17. L. Benini, G. D. Micheli, Networks on chips: a new soc paradigm, IEEE Computer 35 (1) (2002) 70{78}.

18. C. Spampinato et.al. “Automatic Fish Classification for Underwater Species Behavior Understanding”, Proc. of the ACM Int. Workshop Anal. and Retrieval of Tracked Events and Motion in Imagery Streams (ARTEMIS), October 29th, 2010, Florence, Italy.

19. K. Devrari, K.Vinay Kumar, “Fast Face Detection Using Graphics processor”, International Journal of Computer Science and Information Technologies, Vol. 2 (3), 2011,1082-1086.

20. A. Faro, C. Spampinato, G. Scarciofalo, R. Leonardi, An Automatic System for Skeletal Bone Age Measurement by Robust Processing of Carpal and Epiphysial/Metaphysial Bones, IEEE Transactions on Instrumentation and Measurement , Vol. 59, pp. 2539-2553, 2010.

21. Qingyi Gu, Takeshi Takaki, and Idaku Ishii, "a fast multi-object extraction algorithm based on cell-based connected components label-ing", IEICE Transactions on Information and Systems, vol. E95-D, no. 2, pp. 636-645, 2012.

22. Paulo S. B. Nascimento, Helber E. P. de Souza, Francisco A. S. Neves, Member, IEEE, and Leonardo R. Limongi, “FPGA Implementation of the Generalized Delayed Signal Cancelation—Phase Locked Loop Method for Detecting Harmonic Sequence Components in Three-Phase Signals", IEEE Transactions on ON INDUSTRIAL ELECTRONICS, VOL. 60, NO. 2, FEBRUARY 2013.

23. Mrs. S. Allin Christe, Mr.M.Vignesh, and Dr.A.Kandaswamy “AN EFFICIENT FPGA IMPLEMENTATION OF MRI IMAGE FILTERING AND TUMOUR CHARACTERIZATION USING XILINX SYSTEM GENERATOR” International Journal of VLSI design & Communication Systems (VLSICS) Vol.2, No.4, December 2011.

24. Trang Hoang, Van Loi Nguyen, “An Efficient FPGA Implementation of the Advanced Encryption Standard Algorithm” International Conference on Computing and Communication Technologies, Research, Innovation, and Vision for the Future (RIVF), Feb. 27 2012-March 1 2012.

25. Carlos González, Daniel Mozos, Javier Resano, and Antonio Plaza, “FPGA Implementation of the N-FINDR Algorithm for Remotely Sensed Hyperspectral Image Analysis” IEEE TRANSACTIONS ON GEOSCIENCE AND REMOTE SENSING, VOL. 50, NO. 2, FEBRUARY 2012.

26. Nonel Thirer, “FPGA Implementation of a Genetic Algorithm for solving Sudoku Puzzles”, IEEE 27th Convention of Electrical and Electronics Engineers, 2012.

27. Mahdizadeh, H. ; Islamshahr Azad Univ., Tehran, Iran ; Masoumi, M., “Novel Architecture for Efficient FPGA Implementation of Elliptic Curve Cryptographic Processor Over GF(2163)” IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 21 (12), Dec. 2013.

28. W. Dally and B. Towles, “Route Packets, Not Wires: Interconnect, Woes Through Communication-Based Design”, Proc. of the 38th Design Automation Conference, June 2001.

29. T. Ye, L. Benini, and G. De Micheli, "Analysis of power consumption on switch fabrics in network routers". In Proc. Design Automatin Conf. (DAC), June 2002.

30. N. Eisley and L. Peh. "High-level power analysis for on-chip networks". In Proc.Intl. Conf. on Compilers, Architectures and Synthesis for Embedded Systems (CASES), Sept. 2004.

31. T. T. Ye, L. Benini, and G. De Micheli. "Packetization and routing analysis of on-chip multiprocessor networks". Jounal of Systems Architecture, pp.81-104, Feb 2004.

32. Nvidia Corporation Geforce GTX 260 http://www.nvidia.com/object/product_geforce_gtx_260_us.html.

33. Jason Sanders, Edward Kandrot "CUDA by Example An Introduction to General-Purpose GPU Programming", Addison Wesley 2010.

34. David B. Kirk and Wen-mei W. Hwu "Programming Massively Parallel Processors: A Hands-on Approach" Morgun Kaufman Publisher is an imprint of Elsevier 2010

35. NVIDIA CUDA: "Compute Unified Device Architecture", NVIDIA Corp.

36. http://www.eecg.toronto.edu/~jayar/pubs/kuon/foundtrend08.pdf

37. A. I. Kayssi and K. A. Sakallah, "Delay Macromodels for Point-to-Point MCM Interconnections," in Proc. IEEE Multi-chip Module Conference, 1992, pp. 79-82.

38. Kalapi Roy-Neogi and Carl Sechen, "Multiple FPGA Partitioning with Performance Optimization" Proceeding FPGA '95 Proceedings of the 1995 ACM third international symposium on Field-programmable gate arrays. Pages 146-152.

39. Wim Meeus, Kristof Van Beeck, Toon Goedemé , Jan Meel, and Dirk Stroobandt; “An overview of today’s high-level synthesis tools”; Springer Science+Business Media; 12 August 2012.

40. E. Monmasson and M. N. Cirstea, "FPGA design methodology for industrial control systems: a review," IEEE Transactions on Industrial Electronics, vol. 54, no. 4, pp. 1824-1842, August 2007.

41. G. Martin and G. Smith, "High-level synthesis: Past, present, and future," IEEE Design & Test of Computers, vol. 26, no. 4, pp. 18-25, 2009.

42. J. Cong, L. Bin, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhiru, "High-level synthesis for FPGAs: From prototyping to deployment," IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 30, no. 4, pp. 473-491, 2011.

43. D. Varma, D. Mackay, and P. Thiruchelvam, "Easing the verification bottleneck using high level synthesis," in 28th VLSI Test Symposium, 2010, pp. 253-254.

44. P. G. Paulin and J. P. Knight. Force-directed scheduling for the behavioral synthesis of ASICs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 8(6), pp.661-679, 1989.

45. Navarro, D. Lucia Gil, O.; Barragan, L.; Urriza, I.; Jimenez, O. High-Level Synthesis for Accelerating the FPGA Implementation of Computationally-Demanding Control Algorithms for Power Converters. IEEE Transactions on Industrial Informatics, 9(99), 2013.

46. J. Cong, "A new generation of C-base synthesis tool and domain specific computing," in IEEE International SOC Conference, 2008, pp. 386-386.

47. C. Economakos and G. Economakos, "FPGA implementation of PLC programs using automated high-level synthesis tools," in Industrial Electronics, 2008. ISIE 2008. IEEE International Symposium on, 2008, pp. 1908-1913.

48. http://www.xilinx.com/support/documentation/sw_manuals/xilinx2012_4/ug902-vivado-high-level-synthesis.pdf

49. M. M. Jamali, Brett Snyder, John Williams, Ryan Kindred, Gavin St. John, M. W. Majid, J. Ross, J. Frizado, P. V. Gorsevski, V. Bingman, "Remote Avian Monitoring System for Wind Turbines, Paper presented at the 2011 IEEE International Conference on Electro/Information Technology, May 2011.

50. S. Bastas, G. Mirzaei, M. W. Majid, J. Ross, M. M. Jamali, J. Frizado, P. V. Gorsevski, V. Bingman, "A Novel Classification System for Flight Calls," The 2012 IEEE International Symposium on Circuits and Systems, Seoul, Korea, May 2012.

51. Golrokh Mirzaei, Mohammad Wadood Majid, Selin Bastas, Jeremy Ross, Mohsin M. Jamali, Peter V. Gorsevski, Joseph Frizado,Verner P. Bingman, "Acoustic Monitoring Techniques for Avian Detection and Classification," The Asilomar Conference on Signals, Systems, and Computers," November 2012.

52. James W. Davis and Vinay Sharma."Fusion-based background-subtraction using contour saliency". In IEEE International Workshop on Object Tracking and Classification Beyond the Visible Spectrum, IEEE OTCBVS WS Series Bench, 2005.

53. Mariusz Jankowski, Jens-Peer Kuska. “Connected components labeling - algorithms in Mathematica, Java, C++ and C#”. 2004 International Mathematica Symposium.

54. R. D. Yapa, K. Harada, “ Connected Component Labeling Algorithms for Gray-Scale Images and Evaluation of Performance using Digital Mammograms,” IJCSNS International Journal of Computer Science and Network Security, 8(6), pp. 33–41, 2008.

55. K. Suzuki, I. Horiba, and N. Sugie, “Linear-time connected-component labeling based on sequential local operations,” Comput. Vis. Image Underst. 89(1), pp. 1–23, 2003.

56. B. Zeigler, “DEVS formalism: A framework for hierarchical model development”, IEEE Transactions on Software Engineering 14 (2) pp. 228-241, Feb 1988.

57. Choi, B. Park, J. Park, A formal model conversion approach to developing a DEVS-based factory simulator, Simulation 79 (8) pp. 440-461, 2003.

58. J. Lee, Y. Lim, S. Chi, “Hierarchical modeling and simulation environment for intelligent transportation systems”, 80 (2) pp. 61-76, 2004.

59. P. Zeigler, S. Mittal, X. Hu, “Towards a formal standard for interoperability in M&S/ system of systems integration”, GMU-AFCEA Symposium on Critical Issues in C4I, 2008.

60. T. Kim, C. Seo, B. P. Zeigler, “Web-based distributed network analyzer using a system entity structure over a service-oriented architecture”, 86 (3) pp. 155-180, 2010.

61. B. Zeigler, "Today: Recent advances in discrete event-based information technology" In Proc. of the 11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS), 2003, Orlando, FL, USA.

62. J. Hu and R. Marculescu,"Energy and Performance Aware Mapping for Regular NoC Architecture", IEEE Tran. on CAD, April 2005.

63. Noxim Simulator: http://noxim.sourceforge.net/.

64. J. L. Ayala, M. L´opez-Vallejo, and C. L. Barrio, "A case study on power dissipation in the memory hierarchy of embedded systems" in Design of Circuits and Integrated Systems Conference, November 2003.

65. K. Yang, Y. Zhang, and Y. Mizuguchi. A Signal Subspace-Based Subband Approach to Space-Time Adaptive Processing for Mobile Communications. IEEE Trans. Signal Processing, 49(2), pp.401-413, 2001.

66. J. R. Guerci, "Space-Time Adaptive Processing for Radar". Norwood, MA: Artech House, 2003.

67. R. Klemm, "Principles of Space-Time Adaptive Processing", IEE, London, 2006.

68. R. Klemm, "The applications of space-time adaptive processing", IEE, London, 2004.

69. W.L. Melvin. A STAP overview. IEEE Aerospace and Electronic Systems Magazine, 19(1), pp.19-35, 2004.

70. X. Lin and R.S. Blum. Robust STAP algorithms using prior knowledge for airborne radar applications. Signal Processing, 79(3), pp. 273-287, 1999.

71. R. S. Adve. A Two Stage Hybrid Space-Time Adaptive Processing Algorithm. Radar Conference, The Record of the 1999 IEEE, pp. 279-284, 1999.

72. X. Wen, A. Wang, L. Li, and C. Han. Direct data domain approach to space-time adaptive signal processing. Control, Automation, Robotics and Vision Conference, ICARCV, pp. 2070-2075, Dec 2004.

73. D. Cristallini and W. Bürger. A Robust Direct Data Domain Approach for STAP. IEEE TRANSACTIONS ON SIGNAL PROCESSING, 60(3), MARCH 2012.

74. X. Wen, C. Han. Direct data domain approach to space-time adaptive processing. Journal of Systems Engineering and Electronics. 17(3), pp. 59-64, March 2006.

75. S. Park. Prevention of signal cancellation in an adaptive nulling problem. Antennas and Propagation Society International Symposium. Pp. 1040-1043, 1997.

76. Richard Klemm . Principles of Space-Time Adaptive Processing, 3rd Edition, Germany, 2006.

77. G.H. Golub, C.F. Van Loan, “Matrix Computation”, The Johns Hopkins University Press, Baltimore, 1983.

78. Y. Saad, “Numerical Methods for Large Eigenvalue Problems”, Manchester University Press, Manchester, UK, 1992.

79. G.H. Golub, Z. Zhang and H. Zha, “Large sparse symmetric eigenvalue problems with homogeneous linear constraints: the Lanczos process with inner-outer iterations”, 1996.

80. R. Lehoucq and K. Meerbergen. (1997). The inexact Krylov sequence method. [Online], Avaliable: http://ftp.mcs.anl.gov/pub/tech_reports/reports/P612.pdf.

81. R. Morgan and D. Scott. Preconditioning the Lanczos algorithm for sparse symmetric eigenvalue problems, SIAM J. Sci. Stat. Comput, 14(3), pp. 585-593, 1993.

82. Y. Lai, K. Lin and W. Lin., An Inexact Inverse Iteration for Large Sparse Eigenvalue Problems, Numerical Linear Algebra Applications, 4(5), pp. 425-437, 1997.

83. Gene H. Golub and Qiang Ye. In-exact Inverse Iteration for Generalized Eigenvalue Problems. BIT Numerical Mathematics. 40(4), pp. 671-684, 2000.

84. Atkinson, Kendell A. (1988). "Section 8.9". An introduction to numerical analysis (2nd ed.). John Wiley and Sons. ISBN 0-471-50023-2.

85. http://my.safaribooksonline.com/book/-/9780521872652/6dot2-matrix-multiplication-and-inversion/622_the_lu_decomposition.

86. http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf

87. Laura Fischer, Yura Pyatnychko; FPGA Design for DDR3 Memory, March, 2012.

88. Virtex-6 FPGA Memory Interface Solutions User Guide," 1 March 2011. [Online]. Available: http://www.xilinx.com/support/documentation/ip_documentation/ug406.pdf.

89. Bunch, James R.; Hopcroft, John (1974), "Triangular factorization and inversion by fast matrix multiplication", Mathematics of Computation 28: 231–236, ISSN 0025-5718.

90. F. S. Heunis, “Passive Coherent Location Radar using Software-Defined Radio Techniques,” Master’s thesis, University of Cape Town, Private Bag, Rondebosch, 7701, South Africa, May 2010.

91. C. Tong, M. R. Inggs, and G. E. Lange, “Processing design of a networked passive coherent location system,” in Proceedings of the 2011 IEEE Radar Conference, May 2011.

92. F. COLONE, D. W. O’HAGAN, P. LOMBARDO and C. J. BAKER “ A Multistage Processing Algorithm for Disturbance Removal and Target Detection in Passive Bistatic Radar” . IEEE Transaction on Aerospace and Electronic Systems, April, 2009, 698-722.

93. Griffiths, H. D., and Baker, C. J. Passive coherent location radar systems. Part 1: Performance prediction. IEE Proceedings on Radar, Sonar and Navigation, 152, 3 (June 2005), 153—159.

94. Baker, C. J., Griffiths, H. D., and Papoutsis, I. Passive coherent location radar systems. Part 2: Waveform properties. IEE Proceedings on Radar, Sonar and Navigation, 152, 3 (June 2005), 160—168.

95. Lauri, A., Colone, F., Cardinali, R., and Lombardo, P., “Analysis and emulation of FM radio signals for passive radar,” . Presented at the 2007 IEEE Aerospace Conference, Big Sky, MT, Mar. 3—10, 2007.

96. Howland, P. E., Maksimiuk, D., and Reitsma, G. FM radio based bistatic radar. IEE Proceedings on Radar, Sonar and Navigation, 152, 3 (June 2005), 107—115.

97. Kulpa, K. S., and Czekala, Z. Ground clutter suppression in noise radar. Presented at the International Conference on Radar Systems (Radar 2004), Oct. 2004.

98. Axelsson, S. R. J. Improved clutter suppression in random noise radar. Presented at the URSI 2005 Commission F Symposium on microwave Remote Sensing of the Earth, Oceans, Ice, and Atmosphere, Apr. 2005.

99. Gunner, A., Temple, M. A., and Claypoole, R. J., Jr. Direct-path filtering of DAB waveform from PCL receiver target channel. Electronics Letters, 39, 1 (2003), 1005—1007.

100. Kulpa, K. S., and Czekala, Z. Masking effect and its removal in PCL radar. IEE Proceedings on Radar, Sonar and Navigation, 152, 3 (June 2005), 174—178.

101. Cardinali, R., Colone, F., Ferretti, C., and Lombardo, P. Comparison of clutter and multipath cancellation techniques for passive radar. Presented at the IEEE 2007 Radar Conference, Boston, MA, Mar. 2007.

102. Colone, F., Cardinali, R., and Lombardo, P. Cancellation of clutter and multipath in passive radar using a sequential approach. In IEEE 2006 Radar Conference, Verona, NY, Apr. 24—27, 2006, 393—399.

103. Haykin, S. Adaptive Filter Theory (4th ed.). Upper Saddle River, NJ: Prentice-Hall, 2002.

104. D. L. Donoho, “Compressed sensing,” IEEE Trans. Info. Theory, vol. 52, no. 4, pp. 1289–1306, September 2006.

105. E. J. Cand`es, “Compressive sampling,” in Proc. Int. Congress of Mathematicians, Madrid, Spain, Aug. 2006, vol. 3, pp. 1433–1452.

106. Petros Boufounos, Marco F. Duarte, Richard G. Baraniuk, SPARSE SIGNAL RECONSTRUCTION FROM NOISY COMPRESSIVE MEASUREMENTS USING CROSS VALIDATION.

107. D. Needell and J.A. Tropp, CoSaMP: Iterative Signal Recovery from Incomplete and Inaccurate Samples, Information theory and Applications, July 2008, from: http://users.cms.caltech.edu/~jtropp/papers/NT08-CoSaMP-Iterative-preprint.pdf

108. J. A. Tropp and A. C. Gilbert. Signal recovery from random measurements via orthogonal matching pursuit.IEEE Trans. Info. Theory, 53(12):4655–4666, 2007

109. D. L. Donoho, Y. Tsaig, I. Drori, and J.-L. Starck. Sparse solution of underdetermined linear equations by stagewise Orthogonal Matching Pursuit (StOMP). 2007.

110. D. Needell and R. Vershynin. Signal recovery from incomplete and inaccurate measurements via regularized orthogonal matching pursuit. October 2007.

111. E. Cand`es, J. Romberg, and T. Tao. Robust uncertainty principles: Exact signal reconstruction from highly incomplete Fourier information. IEEE Trans. Info. Theory, 52(2):489–509, Feb. 2006.

112. M. A. T. Figueiredo, R. D. Nowak, and S. J. Wright. Gradient projection for sparse reconstruction: Applicationto compressed sensing and other inverse problems. IEEE J. Selected Topics in Signal Processing: Special Issueon Convex Optimization Methods for Signal Processing, 1(4):586–598, 2007.

113. I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. Comm. Pure Appl. Math., 57:1413–1457, 2004.

114. A. C. Gilbert, S. Guha, P. Indyk, S. Muthukrishnan, and M. J. Strauss. Near-optimal sparse Fourier representations via sampling. In Proc. of the 2002 ACM Symposium on Theory of Computing STOC, pages 152–161, 2002.

115. A. Gilbert, M. Strauss, J. Tropp, and R. Vershynin. Algorithmic linear dimension reduction in the $\ell1$ norm for sparse vectors. Submitted for publication, August 2006.

116. E. J. Cand`es, J. Romberg, and T. Tao, “Stable signal recovery from incomplete and inaccurate measurements,” Comm. Pure Appl. Math., vol. 59, no. 8, pp. 1207–1223, Aug. 2006.

117. J. Tropp and A. C. Gilbert, “Signal recovery from partial information via orthogonal matching pursuit,” Apr. 2005.

118. E. J. Cand`es and T. Tao, “The Dantzig selector: Statistical estimation when p is much larger than n,” Ann. Statistics, 2006.

119. J. Haupt and R. Nowak, “Signal reconstruction from noisy random projections,” IEEE Trans. Info. Theory, vol. 52, no. 9, pp. 4036–4048, Sept. 2006.

120. M. A. T. Figueiredo, R. D. Nowak, and S. J. Wright, “Gradient projections for sparse reconstruction: Application to compressed sensing and other inverse problems,” 2007.

121. S.-J. Kim, K. Koh, M. Lustig, S. Boyd, and D. Gorinevsky, “A method for large-scale "1-regularized least squares problems with applications in signal processing and statistics,” 2007.

122. http://www.ece.rice.edu/~vc3/elec633/CoSaMP.pdf

123. Baraniuk, R.G., Cevher, V., Duarte, M.F., Hegde, C. Model-Based Compressive Sensing, IEEE Transactions on Information Theory, Vol (56) Issue (4), April 2010.

124. Reshma .M, Shiwani Hariraman, Swathi .P; Video Compressed Sensing using CoSaMP Recovery Algorithm , IJEDR Conference Proceeding (NCETSE), 2014.

125. http://publications.lib.chalmers.se/records/fulltext/146656.pdf

126. M. Davenport, D. Needell, and M. Wakin, "Signal Space CoSaMP for sparse recovery with redundant dictionaries," IEEE Transactions on Information Theory, 2013.

127. Emmanuel J. Candes , "The Restricted Isometry Property and Its Implications for Compressed Sensing ", Applied & Computational Mathematics, California Institute of Technology, Pasadena, CA 91125-5000.

128. Petros Boufounos, Marco F. Duarte and Richard G. Baraniuk, Sparse Signal Reconstruction from Noisy Compressive Measurements using Cross Validation, from:http://dsp.rice.edu/sites/dsp.rice.edu/files/publications/conference-paper/2007/sparse-ssp-2007.pdf

129. Irina F. Gorodnitsky, Bhaskar D. Rao, Sparse Signal Reconstruction from Limited Data Using FOCUSS: A Re-weighted Minimum Norm Algorithm, IEEE proceedings on signal processing, March 1997, Vol. 45(3):p. 600-616.

130. Baraniuk, R.G., Cevher, V., Duarte, M.F., Hegde, C. Model-Based Compressive Sensing, IEEE Transactions on Information Theory, Vol (56) Issue (4), April 2010.

131. T. Cormen, C. Lesierson, L. Rivest, and C. Stein. Introduction to Algorithms. MIT Press, Cambridge, MA, 2nd edition, 2001.

132. Thomas Blumensath, Mike E. Davies, Iterative Hard Thresholding for Compressed Sensing; http://www.see.ed.ac.uk/~tblumens/papers/BDIHT.pdf.

133. T. Blumensath, M. Davies, Iterative thresholding for sparse approximations, Journal of Fourier Analysis and Applications 14 (5) (2008) 629–654.

134. I. Daubechies, M. Defrise, and C. De Mol. An iterative thresholding algorithm for linear inverse problems with a sparsity constraint. Comm. Pure Appl. Math., 57:1413–1457, 2004.

135. Riol, O., and Vetterli, M.: 'Wavelets and signal processing', IEEE Signal Processing Magazine, 1991, 8, pp. 14-38.

136. Field, D. J.: 'Wavelets, vision and the statistics of natural scenes', Philosophical Transactions of the Royal Society: Mathematical, Physical and Engineering Sciences, 1999, pp. 2527-2542.

137. Antonini, M., Barlaud, M., Mathieu, P., and Daubechies, I.: 'Image coding using wavelet transform', IEEE Transactions on Image Processing, 1992, 2, pp. 205-220.

138. Knowles, G. 1990. VLSI architecture for the discrete wavelet transform. Electron Letters, 26, 15: 1184-1185.

139. Chakabarti, C. and Vishwanath, M. 1995. Efficient realizations of the discrete and continuous wavelet transforms: from single chip implementations to mappings on SIMD array computers. IEEE Transactions on Signal Processing, 43, 3: 759-771.

140. Ali Al-Haj. Fast Discrete Wavelet Transformation Using FPGAs and Distributed Arithmetic. International Journal of Applied Science and Engineering, 2003. 1, 2: 160-171.

141. D. Crookes, "Architectures for high performance image processing: The future", J. of Systems Architecture, 45(10), Apr. 1999, p.739;

142. A. Benkrid, K. Benkrid, D. Crookes, “Design and implementation of a generic 2D orthogonal discrete wavelet transform on FPGA”, 11th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 9-11 April 2003, p.162;

143. Bahoura M. and Ezzaidi H. “Pipelined Architecture for Discrete Wavelet Transform Implementation on FPGA”, 22nd International Conference on Microelectronics (ICM 2010).

144. Afandi et. Al. “3D Haar Wavelet Transform with Dynamic Partial Reconfiguration for 3D Medical Image Compression” Biomedical Circuits and Systems Conference, 2009. BioCAS 2009. IEEE

145. A. Ahmad, B. Krill, A. Amira, and H. Rabah, “Efficient architectures for 3D HWT using dynamic partial reconfiguration,” Journal of Syst. Arch., vol. 56(8), pp. 305–316, 2010.

146. M.-M. Salem, M. Appel, F. Winkler, B. Meffert, FPGA-based smart camera for 3D wavelet-based image segmentation, in: Proceedings of the Second ACM/IEEE International Conference on Distributed Smart Cameras (ICDSC 2008), California, USA, 2008, pp. 1–8.

147. Gerald, K.: ‘A Friendly Guide To Wavelets’, 1994.

148. Y. Bar-Shalom and W. D. Blair, Multitarget-Multisensor Tracking: Applications and Advances, vol. III, Archtech House, Norwood, MS, 2000.

149. S. Blackman and R. Popoli, Design and Analysis of Modern Tracking Systems, Artech House, Norwood, MA, 1999.

150. http://pskla.kpi.ua/lib/2009/Simon.pdf

151. Y. Boers and J.N. Driesses, Multitarget Particle Filter Track Before Detect Application, IEEE Proceedings Radar, Sonar and Navigation, 2003, Vol. 151: p. 351-357.

152. M. Sanjeev Arulampalam, Simon Maskell, Neil Gordon and Tim Clapp, A Tutorial on Particle Filters for Online Nonlinear/Non-Gaussian Bayesian Tracking, IEEE Transactios on Signal Processing, Feb. 2002, Vol. 50(2): p.174-188.

153. Niclas Bergman, Recursive Bayesian Estimation: Navigation and Tracking Applications, 1999, http://www.control.isy.liu.se/research/reports/Ph.D.Thesis/PhD579.pdf

154. G. Kitagawa, "Non-Gaussian state-space modelling of non-stationary time series (with discussion)," Journal of the American Statastical Association, 82(400), pp. 1032-1063 (December 1987).

155. N. Gordon, D. Salmond, and A. Smith, "Novel approach to nonlinear/non-Gaussian Bayesian state estimation," IEEE Proceedings-F, 140(2), pp. 107-113 (April 1993).

156. D. Moolenaar, L. Nachtergaele, F. Catthoor, and H. De Man, "System level power exploration for MPEG-2 decoder on embedded cores: a systematic approach" Proc. IEEE Wsh. on Signal Processing Systems, Nov. 1997.

157. K. Flautner, NS. Kim, D. Blaauw, and T. Mudge, "Drowsy Caches: Simple Techniques for Reducing Leakage Power", IEEE Symposium on Computer Architecture, May, 2002.

158. N. Kim, K. Flautner, D. Blaauw, and T. Mudge, “Circuit and micro-architectural techniques for reducing cache leakage power”, IEEE Trans on VLSI, pp.167-184, Feb 2004.

159. Artix7 FPGAs from Xilinx, Inc. (www.xilinx.com).

160. http://electroscience.osu.edu/9219.cfm