Parallel reconfigurable hardware architectures for video processing applications

Karim Mohamed Abedallah Ali

Université de Valenciennes et du Hainaut-Cambrésis

THÈSE

Soutenance prévue publiquement le le 8 février 2018 à l'Université de Valenciennes
pour obtenir le titre de

Doctorat en informatique

par

Karim Mohamed Abedallah Ali

Parallel Reconfigurable Hardware Architectures for Video Processing Applications

Composition du jury

Président :	Smail NIAR	Professeur	LAMIH, Université de Valenciennes
Rapporteurs :	Michael Hübner	Professeur	ESIT, Université de Bochum
	Dirk Stroobandt	Professeur	CSL, Université de Gand
Examinateurs :	Cécile Belleudy	Professeur	LEAT, Université Nice Sophia Antipolis
	Maria Giovanna Sami	Professeur	Polytechnique de Milan
	Nizar Fakhfakh	Docteur	NAVYA Technology
Directeurs :	Jean-Luc Dekeyser	Professeur	LIFL, Université de Lille I
	Rabie Ben Atitallah	MdC, HdR	LAMIH, Université de Valenciennes

UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT-CAMBRÉSIS
ECOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR UNIVERSITÉ LILLE NORD-DE-FRANCE-072
Laboratoire d'Automatique, de Mécanique et d'Informatique industrielles et Humaines 8201
UVHC, Le Mont Houy, 59313 Valenciennes Cedex 9
Tél. : +33 (0)3 27 51 13 50 – Télécopie : +33 (0)3 27 51 19 40

Université de Valenciennes et du Hainaut-Cambrésis
THÈSE

Soutenance prévue publiquement le le 8 février 2018 à l'Université de Valenciennes
pour obtenir le titre de

Doctorat en informatique

par

Karim Mohamed Abedallah Ali

Architectures Parallèles Reconfigurables pour le Traitement Vidéo Temps-Réel

Composition du jury

Président :	Smail NIAR	Professeur	LAMIH, Université de Valenciennes
Rapporteurs :	Michael Hübner	Professeur	ESIT, Université de Bochum
	Dirk Stroobandt	Professeur	CSL, Université de Gand
Examinateurs :	Cécile Belleudy	Professeur	LEAT, Université Nice Sophia Antipolis
	Maria Giovanna Sami	Professeur	Polytechnique de Milan
	Nizar Fakhfakh	Docteur	NAVYA Technology
Directeurs :	Jean-Luc Dekeyser	Professeur	LIFL, Université de Lille I
	Rabie Ben Atitallah	MdC, HdR	LAMIH, Université de Valenciennes

UNIVERSITÉ DE VALENCIENNES ET DU HAINAUT-CAMBRÉSIS
ECOLE DOCTORALE SCIENCES POUR L'INGÉNIEUR UNIVERSITÉ LILLE NORD-DE-FRANCE-072
Laboratoire d'Automatique, de Mécanique et d'Informatique industrielles et Humaines 8201
UVHC, Le Mont Houy, 59313 Valenciennes Cedex 9
Tél. : +33 (0)3 27 51 13 50 – Télécopie : +33 (0)3 27 51 19 40

Acknowledgements

"So all praise is for Allah - Lord of the heavens and Lord of the earth, Lord of all worlds." **The Noble Quran** [**45 :36**].

I would like to thank my supervisors Prof. Jean-Luc Dekeyser and Prof. Rabie Ben Atitallah for giving me that opportunity to work with them. Special thanks to Rabie for his support, patience and encouragement. He was step by step beside me by his advice and help.

I would like to thank my thesis reviewers Prof. Michael Hübner and Prof. Dirk Stroobandt for their time to review this manuscript and for their feedback. I would like to thank Prof. Smail Niar for presiding over my thesis defence jury as well as the other members : Prof. Maria Giovanna Sami and Prof. Cécile Belleudy. I would like also to thank our industrial partner NAVYA Technology represented by Dr. Nizar Fakhfakh.

I would like to thank my colleagues, friends and brothers in LAMIH, residence and Valenciennes for their help and love. Thanks to my Indonesian family for welcoming my guests on the defence day. Thanks to my friend Yunfei for controlling the live stream. Thanks to my dear brothers and sisters who prepared a magnificent celebration for me in the residence. Thanks to Safouene, Riyadh, Imad, Marwan, Moustafa, Abdoulaziz, Mohktar, Ayman, Zeineb, Sara, Afaf and Soufia.

I would like to thank my big family, my parents, my sister for their prayers. Thanks to my brother Mostafa for his help to edit my presentation slides in an attractive way. Finally, I would like to thank everyone who shared with me that success.

Karim Ali
February 2018

Abstract

Embedded video applications are now involved in sophisticated transportation systems like autonomous vehicles. Many challenges faced the designers to build those applications, among them : complex algorithms should be developed, verified and tested under restricted time-to-market constraints, the necessity for design automation tools to increase the design productivity, high computing rates are required to exploit the inherent parallelism to satisfy the real-time constraints, reducing the consumed power to extend the operating duration before recharging the vehicle, etc. In this thesis work, we used FPGA technologies to tackle some of these challenges to design parallel reconfigurable hardware architectures for embedded video streaming applications. First, we implemented a flexible parallel architecture with two main contributions : (1) We proposed a generic model for pixel distribution/collection to tackle the problem of the huge data transferring through the system. The required model parameters were defined then the architecture generation was automated to minimize the development time. (2) We applied frequency scaling as a technique for reducing power consumption. We derived the required equations for calculating the maximum level of parallelism as well as the ones used for calculating the depth of the inserted FIFOs for clock domain crossing.

As the number of logic cells on a single FPGA chip increases, moving to higher abstraction design levels becomes inevitable to shorten the time-to-market constraint and to increase the design productivity. During the design phase, it is common to have a space of design alternatives that are different from each other regarding hardware utilization, power consumption and performance. We developed ViPar tool with two main contributions to tackle this problem : (1) An empirical model was introduced to estimate the power consumption based on the hardware utilization (Slice and BRAM) and the operating frequency. In addition to that, we derived the equations for estimating the hardware resources and the execution time for each point during the design space exploration. (2) By defining the main characteristics of the parallel architecture like parallelism level, the number of input/output ports, the pixel distribution pattern, etc. ViPar tool can automatically generate the parallel architecture for the selected designs for implementation. In the context of an industrial collaboration, we used high-level synthesis tools to implement a parallel hardware architecture for Multi-window Sum of Absolute Difference stereo matching algorithm. In this implementation, we presented a set of guiding steps to modify the high-level description code to fit efficiently for hardware implementation as well as we explored the design space for different alternatives in terms of hardware resources, performance, frequency and power consumption. During the thesis work, our designs were implemented and tested experimentally on Xilinx Zynq ZC706 (XC7Z045-FFG900) evaluation board.

Keywords Video Streaming applications - Parallel Reconfigurable Architectures - High-level Synthesis - Design Space Exploration - FPGA.

Résumé

Les applications vidéo embarquées sont de plus en plus intégrées dans des systèmes de transport intelligents tels que les véhicules autonomes. De nombreux défis sont rencontrés par les concepteurs de ces applications, parmi lesquels : le développement des algorithmes complexes, la vérification et le test des différentes contraintes fonctionnelles et non-fonctionnelles, la nécessité d'automatiser le processus de conception pour augmenter la productivité, la conception d'une architecture matérielle adéquate pour exploiter le parallélisme inhérent et pour satisfaire la contrainte temps-réel, réduire la puissance consommée pour prolonger la durée de fonctionnement avant de recharger le véhicule, etc. Dans ce travail de thèse, nous avons utilisé les technologies FPGAs pour relever certains de ces défis et proposer des architectures matérielles reconfigurables dédiées pour des applications embarquées de traitement vidéo temps-réel. Premièrement, nous avons implémenté une architecture parallèle flexible avec deux contributions principales : (1) Nous avons proposé un modèle générique de distribution/collecte de pixels pour résoudre le problème de transfert de données à haut débit à travers le système. Les paramètres du modèle requis sont tout d'abord définis puis la génération de l'architecture a été automatisée pour minimiser le temps de développement. (2) Nous avons appliqué une technique d'ajustement de la fréquence pour réduire la consommation d'énergie. Nous avons dérivé les équations nécessaires pour calculer le niveau maximum de parallélisme ainsi que les équations utilisées pour calculer la taille des FIFO pour le passage d'un domaine de l'horloge à un autre.

Au fur et à mesure que le nombre de cellules logiques sur une seule puce FPGA augmente, passer à des niveaux d'abstraction plus élevés devient inévitable pour réduire la contrainte de « time-to-market » et augmenter la productivité des concepteurs. Pendant la phase de conception, l'espace de solutions architecturales présente un grand nombre d'alternatives avec des performances différentes en termes de temps d'exécution, ressources matérielles, consommation d'énergie, etc. Face à ce défi, nous avons développé l'outil ViPar avec deux contributions principales : (1) Un modèle empirique a été introduit pour estimer la consommation d'énergie basé sur l'utilisation du matériel (Slice et BRAM) et la fréquence de fonctionnement ; en plus de cela, nous avons dérivé les équations pour estimer les ressources matérielles et le temps d'exécution pour chaque alternative au cours de l'exploration de l'espace de conception. (2) En définissant les principales caractéristiques de l'architecture parallèle comme le niveau de parallélisme, le nombre de ports d'entrée/sortie, le modèle de distribution des pixels, ..., l'outil ViPar génère automatiquement l'architecture matérielle pour les solutions les plus pertinentes. Dans le cadre d'une collaboration industrielle avec NAVYA, nous avons utilisé l'outil ViPar pour implémenter une solution matérielle parallèle pour l'algorithme de stéréo matching « Multi-window Sum of Absolute Difference ». Dans cette implémentation, nous avons présenté un ensemble d'étapes pour modifier le code de description de haut niveau afin de l'adapter efficacement à l'implémentation matérielle. Nous

avons également exploré l'espace de conception pour différentes alternatives en termes de performance, ressources matérielles, fréquence, et consommation d'énergie. Au cours de notre travail, les architectures matérielles ont été implémentées et testées expérimentalement sur la plateforme d'évaluation Xilinx Zynq ZC706.

Mots-clés : Applications vidéo temps-réel - Architectures reconfigurables parallèles - Synthèse de haut niveau - Exploration de l'espace de conception - FPGA.

List of Publications

1. **K. M. A. Ali,** R. Ben Atitallah, S. Hanafi and J. L. Dekeyser, "A Generic Pixel Distribution Architecture for Parallel Video Processing," 2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14), Cancun, 2014, pp. 1-8. doi : 10.1109/ReConFig.2014.7032547
2. **K. M. A. Ali,** R. B. Atitallah, N. Fakhfakh and J. L. Dekeyser, "Using Hardware Parallelism for Reducing Power Consumption in Video Streaming Applications," 2015 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC), Bremen, 2015, pp. 1-7. doi : 10.1109/ReCoSoC.2015.7238104
3. **K. M. A. Ali,** R. B. Atitallah, N. Fakhfakh and J. L. Dekeyser, "Exploring HLS Optimizations for Efficient Stereo Matching Hardware Implementation," 2017 13th International Symposium on Applied Reconfigurable Computing (ARC), Delft, 2017, pp. 168–176. 10.1007/978-3-319-56258-2_15
4. R. B. Atitallah and **K. M. A. Ali,** "FPGA-Centric High Performance Embedded Computing : Challenges and Trends," 2017 Euromicro Conference on Digital System Design (DSD), Vienna, 2017, pp. 390-395. doi : 10.1109/DSD.2017.88
5. Y. B. Jmaa, **K. M. A. Ali,** D. Duvivier, M. B. Jemaa and R. B. Atitallah, "An Efficient Hardware Implementation of TimSort and MergeSort Algorithms Using High Level Synthesis," 2017 International Conference on High Performance Computing & Simulation (HPCS), Genoa, 2017, pp. 580-587. doi : 10.1109/HPCS.2017.92
6. M. Bouain, **K. M. A. Ali,** D. Berdjag, N. Fakhfakh and R. B. Atitallah, "An Embedded Multi-Sensor Data Fusion Design for Vehicle Perception Tasks," 2017 10th International Conference on Computer Science and Information Technology, Florence, Italy, 2017.

Table of contents

Chapter 1

Introduction

1.1 The context of the work

This thesis has a collaboration contract with NAVYA Technology [5]. NAVYA is a French start-up company specialized in manufacturing autonomous electric vehicles for first and last mile transportation. First and last mile transportation problem appears in the places where public transport stations or parking facilities are far away from our final destination while it is still too far to go on foot. In such cases, autonomous vehicles come as a solution to increase accessibility and mobility in large public places like hospitals, universities, airports or large industrial sites where it delivers frequent transportation services between destinations. Usually, an autonomous vehicle is equipped by image sensors to detect, classify and track the observed obstacles in the surrounding area of the vehicle ; in addition to that, other cameras could be installed inside the vehicle for passengers security. This kind of video processing applications can be processed by using industrial PCs. However, with the growing demand to increase the image resolution for better precision, increasing the frame rate, installing more image sensors or applying more complex video algorithms ; industrial PCs reached their processing limits to satisfy that massive data processing request under real-time constraints. Adding another PC could be a solution for increasing the processing capability of the system, but unfortunately those PCs have high power consumption rates, and consequently, they are not advised for battery-based systems like autonomous electric vehicles. For that reason, NAVYA asked to find an efficient solution that fulfils a set of specifications including high processing rate, low power consumption, short developing cycle, lower production costs, etc.

Today, CMOS technology scales down to increase the available number of transistors on a single chip and thus increasing the available computation power [93]. Today, the availability of powerful computing hardware at low cost motivates the industry and academia towards smart cameras and intelligent sensor solutions. In smart camera systems, the images are captured and processed locally at the image sensor node then only the result or regions of interest are communicated to the central server [19] [80]. Increasing the computational power of the sensor node permits the evolution of intelligent sensors where complex algorithms are executed locally at the sensor node then advanced actions are taken upon that. Smart camera solutions have different advantages : (i) Reducing data transfer loads between the sensor nodes and the server. (ii) Increasing the computing power of the smart camera encourages to increase the resolution of the captured images. (iii) Adding more camera nodes is not limited by the processing capabilities of the centralized server ; thus, the system is more flexible and expandable. From the examples for smart camera applications, we can mention video surveillance in public areas [116], crowd behaviour analysis for detecting abnormal activities [26] [102], vehicle tracking [46], car parking occupancy detection [11], intelligent transportation systems [117] [29], assisted living for elder people [31] [32], monitoring systems for kids safety, analyzing customers behaviour in markets, etc.

To deliver the required computing power, there are many technologies in the market that

can be used as a solution like (ASIC, FPGA, DSP, GPU, ...). Field-Programmable Gate Array (FPGA) technology is a competitive solution for building smart camera applications if compared to the other solutions for different reasons :

- One of the main features of FPGA devices is that they are reprogrammable platforms where different hardware architectures could be implemented using the same FPGA platform. Therefore, the hardware architecture can be redesigned and programmed without changing the chip itself to adapt to the rapid changes in the technologies for image sensors and video processing algorithms.
- By exploiting the inherent parallelism in video processing applications, FPGA technology enables us to implement massively parallel architectures due to the huge number of programmable logic available on a single chip. As long as there are enough hardware resources available, the implemented parallel architectures over FPGA platforms are scalable by adding more processing elements.
- Application Specific Integrated Circuits (ASIC) are known by their efficient performance at very low power consumption level, however, their production cost is very expensive with relatively long design cycle to develop the customized integrated circuit. For processor-based solutions, they are characterized by their sequential processing on fixed architectures accessible to everyone with relatively high power consumption rates. While FPGAs represent an in-between solution where they are characterized by their high performance per watt ; thus, FPGAs are good candidate for embedded video processing applications where the constraint of low power consumption should be respected. In addition to that, FPGA is considered as the prototyping platform for ASIC circuits ; so FPGA designs can be ported to ASIC circuits when very low-power consumption circuits are required at mass production.
- Real-time video processing applications require high computing power to run complex video processing algorithms. In the past, this kind of applications was built using general-purpose processor coupled with video accelerators on the same PCB board. In today's market, SoC platforms are introduced by FPGA vendors (Zynq platform from Xilinx and Stratix 10 from Intel FPGA) where ARM processor is coupled with the programmable logic on the same single chip. This combination is more attractive for embedded video applications because neither the entire application is accelerated by using hardware nor entirely processed by a traditional processor. However, only the intensive calculation tasks are selected for hardware acceleration while the rest of the application is kept running on the ARM processors. In addition to that, the presence of general purpose processor on the same SoC allows the usage of Linux operating systems.
- FPGAs are flexible devices since it is possible to apply upgrades remotely over Ethernet connection. In this situation, a new reconfigurable bitstream can be sent to the embedded processor to reprogram the FPGA chip when the sensor settings are changed

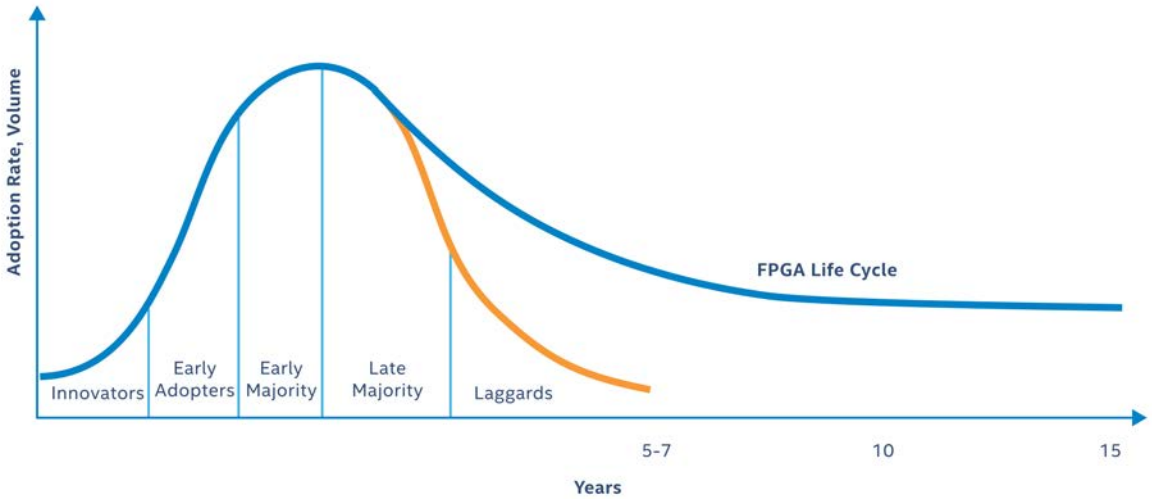

FIGURE 1.1 – FPGA Alignment with Industrial Life Cycles [23]

or when a modified hardware architecture for the applied video processing algorithm is recently developed.

- FPGA technology enables us to integrate several functionalities by implementing them on the same chip. Among the benefits of integration, we can mention : (i) It reduces the board size, chip count and assembly complexity. (ii) It decreases the dynamic power consumption in the system by reducing the I/O communication between the functional blocks. (iii) It increases the system reliability by placing fewer components on the PCB board.
- The life cycle of many microcontrollers and DSP devices range between 5-7 years because their vendors tend to obsolete mature device much sooner while the life cycle of FPGA devices range between 15-20 years as depicted in Fig. 1.1. The long life cycle for FPGA devices helps the customers to avoid the high cost of obsolescence management [105] [94] [23].
- Dynamic Partial Reconfiguration (DPR) is another feature of FPGA platforms where some parts of the hardware can be reconfigured during runtime while the rest of the design is kept unchanged. DPR can be exploited in video processing applications where the same system can support different configurations. In that case, different FPGA reconfigurable bitstreams with varied options for frame rate, resolution and video compression modes can be stored in the system then the best configuration can be selected according to the current system requirements [8].

In addition to the previously mentioned features, FPGA-based solutions are very promising solutions for the domain of autonomous vehicles because FPGA technologies could offer high processing rates at low power consumption rates. In this thesis, we presented

design solutions for some challenges arose while designing parallel reconfigurable hardware architectures for video processing applications as well as we realized those solutions over the latest FPGA technologies.

1.2 Trends and Challenges

1.2.1 Industrial Challenges

- **Time-to-Market**. It is defined as the time taken by a product since being conceived until being available in the market for sale. As a result of the constant competition between companies, time-to-market is considered as a crucial constraint in the product design cycle. It can also be defined as the flexibility to make changes in the design during the development cycle as a response to the customer feedback without additional time delays in the production schedule. In autonomous vehicle industry, different algorithms are developed for detecting traffic lights, pedestrian, obstacles, ..., in the surrounding environment of the vehicle. Knowing how to develop, verify and test those algorithms while respecting time-to-market constraint is so crucial.
- **Production Cost**. Most of the time, the price of the product controls the choice of the customer to buy it or not. Consequently, we are not aiming only to design new products, but also we are aiming to do that at acceptable production costs to have a margin for profit. We can divide costs into : (i) Non-recurring engineering costs which refer to the one-time costs for designing, developing and validating a new product. (ii) Production costs which are paid per unit product including materials, labour, etc. Automating the design process as well as targeting architectures that could be adapted for algorithmic changes are among the reasons to maintain the production costs in the autonomous vehicle industry.
- **Low power consumption solutions**. Autonomous vehicles are 100% electric-based vehicles. The used electricity is usually generated from renewable resources (solar or wind) to increase the economic and environmental value of the electric vehicle industry. Consequently, the designers are devoting their efforts to reduce the power consumed by the vehicle to extend the number of the operational hours before the batteries have to be recharged.

1.2.2 Scientific Challenges

- **Flexible parallel reconfigurable architecture**. Video processing are intensive signal applications where a huge amount of data is transferred from/to the computing nodes. Today the increasing demand for more frame rate or for increasing the image resolution are additional challenges for video processing applications especially when real-time constraints are considered. This challenge is augmented in autonomous vehicle industry

where several image sensors are installed in the vehicle for obstacle detection, tracking and classification. The combination of reconfigurability and parallelism is a key solution for the above-described problem. In this challenge, we search for answers to the following questions :

- How to calculate the parallelism level in order to exploit the inherent parallelism in video processing applications ?
- Processing latency due to inefficient data distribution could limit the performance and the safety conditions of the autonomous vehicle. So, how can we manage input/output pixel distribution to guarantee high processing rates ?
- How can the operating frequency be scaled in order to reduce the power consumption without affecting the system processing rate ?

- **Tools for design automation**. Today, a tremendous number of logic cells exist on a single FPGA chip. Using conventional ways to design, simulate, implement and validate such large FPGA designs is a time-consuming process. For that reason, designers always aim to move the design efforts to the higher abstraction levels to increase the design productivity and to shorten the time-to-market conditions. In FPGA design, there is no unique solution for the design problem, but it is common to have a space of design solutions that are different from each other in terms of hardware utilization, performance, operating frequency and power consumption. Indeed, it is not practical to explore manually a design space consisted of hundreds or thousands of design points. Automating the exploration process is necessary to search rapidly for the solution which better fits with the given system constraints.

1.3 Contributions

To tackle the first challenge for flexible parallel architecture, we had two main contributions :

1. **Generic pixel distribution/collection model**. To address the challenge of Input/Output data distribution in video processing applications, we proposed a generic hardware pixel distribution/collection model for parallel video streaming applications. We defined the required model parameters to have a flexible pixel distribution in both vertical and horizontal directions. After defining the model parameters, the hardware architecture for the pixel distributor/collector is automatically generated to decrease the development efforts.
2. **Hardware parallelism for reducing power consumption**. We implemented a flexible hardware parallel architecture in conjunction with frequency scaling as a technique for reducing power consumption in video streaming applications. We derived the required equations to calculate the maximum level of parallelism to be implemented.

Also, we derived the equations to determine the maximum depth of the used FIFOs for clock domain crossing. The variation in the level of parallelism formed a set of design alternatives which are different concerning hardware utilization and power consumption. Accordingly, the designer is free to select the target design according to what power reduction is required at how much hardware resources it costs.

To tackle the second challenge for design automation, we developed ViPar tool with two main contributions :

3. **Design space exploration.** By varying the design parameters like level of parallelism, operating frequency, etc, a space of different design points is constructed. First, we derived the equations to estimate both resource utilization and performance for each design in the design space. Second, we introduced an empirical model to estimate the power consumption based on Slice, BRAM, and the operating frequency for each design. Finally, the design space was explored for the candidate designs.
4. **Parallel architecture automatic generation.** We used ViPar tool to generate the parallel architecture corresponding to the best candidate points for experimental validation on FPGA. The main characteristics of the parallel architecture like the level of parallelism, the pixel distribution pattern, the number of inputs and output ports, etc, were defined in a specification file ; then, this file is considered as an input for ViPar tool for parallel architecture generation.

Experimental validation. NAVYA Technology proposed Multi-window Sum of Absolute Difference stereo matching algorithm as a video processing application for experimental validation. First, we used high-level synthesis tool to implement the hardware architecture for the stereo matching algorithm. Second, we showed a set of guiding steps in order to obtain an efficient hardware implementation by exploiting the inherent parallelism in that application. We used ViPar to explore the different design alternatives for this application, and the selected designs were validated experimentally over Zynq ZC706 (XC7Z045-FFG900) FPGA board.

1.4 Outline

The rest of this document is organized as follows : Chapter 2 presents the related works. Chapter 3 presents our generic model for pixel distribution/collection and how to reduce the power consumption by using hardware parallelism. Chapter 4 details the hardware implementation for Multi-window Sum of Absolute Difference stereo matching algorithm. The details of ViPar tool and how to estimate the performance parameters for each design point for design space exploration are described in Chapter 5. Finally, conclusions with some proposed future works are presented in Chapter 6.

Chapter 2

Background and Related Works

2.1 Introduction

In this chapter, we will review FPGA-based architectures for video processing applications like soft vector processor, soft VLIW processors, soft GPGPU processors, etc. Then, we will introduce high-level synthesis tools for accelerating the design process. In this point, we will discuss the research efforts done to enhance the quality of the obtained HLS-based designs or to demonstrate the maturity of HLS tools by building efficient solutions for different applications. At high-level design step, the design parameters can be combined in different ways to get different hardware architectures. Thus, we will review the research works related to the design space exploration while using HLS tools and their impact on the design productivity.

2.2 Reconfigurable Architectures for Video Processing Applications

In the literature, hundreds of research works present the implementation of various video processing applications over different hardware architectures range from CPU, GPU, FPGA, DSP and ASIC. One recent study [27] compared between three different hardware platforms : FPGA (Altera Stratix IV E530), CPU (Intel Core i7-960 Quad-Core) and GPU (NVIDIA GeForce GTX 560) to implement three different video processing applications : Sum of Absolute Difference (SAD) stereo matching, 2D convolution and correntropy filter. The implementations were evaluated for images of size 480p, 720p and 1080p to process a kernel of a square size that ranged from 2x2 to 100x100. Two different CPU implementations were presented : One by using OpenCL library on the multi-core CPUs, while the other was a sequential C++ implementation which was considered as a baseline for speed-up comparison. For 2D convolution filter, it was implemented in time-domain (TD) and frequency-domain (FD) for both CPU and GPU platforms. The performance of the SAD algorithm was depicted in Fig. 2.1(a) ; where the results for the GPU implementation was always better than the quad-core CPU. While for FPGA, the computations were independent of the kernel size ; thus, the performance was constant up to the maximum parallel kernel size (64x64). The performance was then decreased for large kernel sizes due to kernel partitioning overhead and final output aggregation. For window size less than 10x10, CPU was faster than FPGA. Also, GPU delivered performance better than FPGA for windows less than 35x35 ; whereas, FPGA outperformed for large kernel sizes. For 2D convolution filter, Fig. 2.1(b) showed that GPU implementations (time-domain and frequency-domain) outperformed FPGA implementation for all kernel sizes. Another experiment was done by implementing the applications over heterogeneous single-chip platforms (CPU/GPU or CPU/FPGA) to compare their performance to the previous PCIe accelerators. Figure 2.1(c) showed that CPU/FPGA implementation had an average speedup of 1.7x over the PCIe

version ; while CPU/GPU experienced the greatest speedup at low kernel sizes then it decreased quickly due to the large data transfer for large kernels. Figure 2.1(d) evaluated the energy consumption to process one frame for SAD and 2D convolution for 720p image size over different platforms. For SAD application, GPU was more energy efficient than FPGA for small kernels up to 25x25 while vice versa for large ones. For 2D convolution, GPU was almost the most efficient for different kernel sizes. Similar comparison studies [47] [49] were performed for different applications implemented over CPU, GPU and FPGA to reach for similar results. We can summarize those results that no platform can outperform for all video processing applications. The tradeoff between the performance metrics in terms of design efforts, execution time, power consumption, solution cost, etc, play an important role to prefer one solution rather than another.

For that reason, heterogeneous platforms (CPU coupled with FPGA or CPU coupled with GPU) emerged as a preferred platform solution for video processing applications to map each task of the application into the appropriate processing engine which gives the best-expected performance. In this work, large kernel sizes are processed under real-time constraints. Thus, reconfigurable hardware is considered the most appropriate technology that fits to our requirements (autonomous vehicles). In this section, we will review some of the proposed reconfigurable architectures for building image/video processing applications. Mainly, these architectures vary in terms of flexibility and performance. From the hardware point of view, we mean by flexibility is the ability to customize the reconfigurable architecture to offer a large space of architectural configurations suitable for building a vast variety of image/video applications. While from the software point of view, flexibility is the ability to code different image/video applications without the need to resynthesis the reconfigurable architecture. As far as we know that designing is the answer for the tradeoff question where flexibility has an inverse impact on the expected application performance such that as the architecture flexibility increases, the performance decreases and vice versa.

Heterogeneous Reconfigurable Platforms. Figure 2.2 shows the architecture of the latest heterogeneous MPSoC from Xilinx [1]. Ultrascale+ Zynq MPSoC EV platform is composed of different processing engines like quad-core ARM Cortex A53-based APU, dual-core ARM Cortex R5-based for real-time processing, ARM Mali-400 MP2 GPU with a geometry processor and two pixel processors, reconfigurable hardware resources range between 177-461 K for FF, 88-230 K for LUT, 18-38 Mb for BRAM and 728-1728 for DSP slices ; in addition to high-speed peripherals and advanced I/O capabilities for efficient communication. This powerful platform is a good candidate for building high-performance real-time video/image processing applications for automotive industry. Before the appearance of a single chip heterogeneous platform, heterogeneity was achieved either by integrating several processing engines over the same PCB board or by dedicating a part of the FPGA fabric to build a soft-core processor like Xilinx MicroBlaze or Altera Nios II.

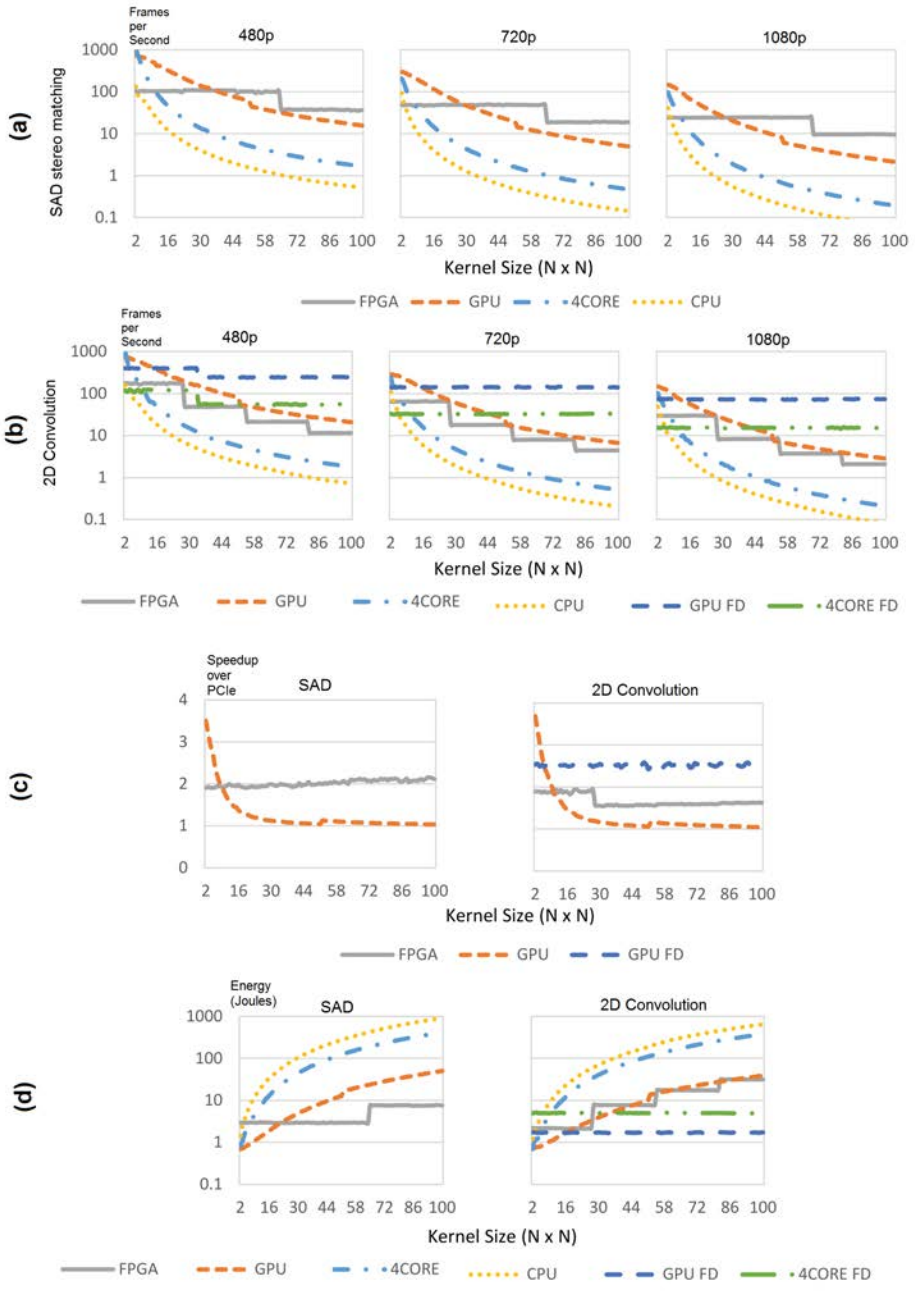

FIGURE 2.1 – (a)Performance of the SAD implementations in frames per second for different kernel and image sizes, (b)Performance of the 2D convolution implementations measured in frames per second for different kernel and image sizes, (c) Speedup of single-chip implementations over their PCIe equivalents for SAD and 2D convolution at different kernel sizes tested on 720p image, (d)System energy consumed to process one frame in SAD and 2D convolution at different kernel sizes tested on 720p image [27].

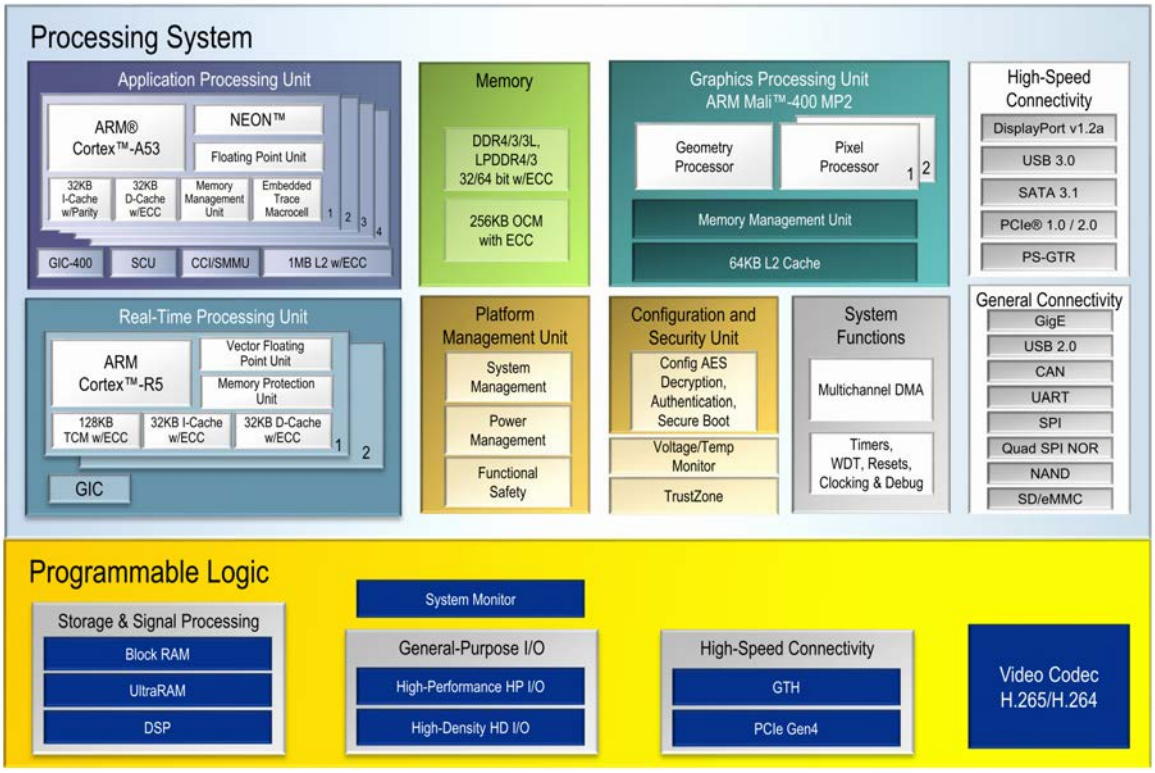

FIGURE 2.2 – Zynq UltraScale+ MPSoC EV block diagram

Soft Vector Processors. VectorBlox MXP [88] is an FPGA-based soft vector processor for performing high parallel data tasks. The architecture is parameterized by allowing the user specifying the number of parallel ALU ranging from 1 to 128 parallel ALU. Programs are written in C/C++ either by using MXP specific library routines or by using MXP intrinsic functions placed inline with the regular C/C++ code. For video processing applications, MXP offers two modules : FrameWriter to write image frames to the external memory or StreamWriter to write few scanlines to the MXP scratchpad. VectorBlox MXP processor can boost the performance by defining up to four custom instructions to save the execution of tens or hundreds of ordinary vector instructions. In the experimental results, authors implemented H.264 deblocking filter by defining custom instructions. In addition to that, they showed the implementation of several video applications like median filter, motion estimation and saliency computation [88] [40].

Authors in [114] developed VESPA as a flexible portable soft vector processor architecture with up to 32 vector lanes. VESPA is portable by being fully implemented in synthesizable Verilog without any FPGA-device dependencies. While it is flexible by parameterizing the set of architectural options like number of vector lanes, number of ALU per bank, number of register file banks, maximum vector length, Icache depth, Icache line size, etc. Theoretically,

application speedup will increase linearly as the number of vector lanes grows, but this is limited due to several factors : (i) The candidate portion for parallelism differs from one application to another. (ii) Adding more vector lanes decreases the overall achieved operating frequency. For example for a vector processor with 32-lane, the authors showed that the clock frequency dropped significantly causing the expected speedup to reduce from 15x to 1lx. In the experiments, the authors used industry-standard embedded microprocessor applications (EEMBC benchmark) like FIR filter, convolution encoder and RGB filter for experimental validation.

Soft VLIW Processors. Authors in [20] proposed a customizable VLIW processor with a variable instruction set for exploiting parallelism. OpenIMPACT tool was used to convert C-coded algorithms into independent assembler intermediate representation ; then, it was analyzed and reorganized in VLIW instructions. For experimental validation, three basic image applications were tested on Virtex-6 FPGA board (Sobel filter, convolution filter and fast discrete cosine transform). The authors then extended their experiments by realizing contactless palmprint extraction algorithm for biometric applications. Their VLIW implementation over FPGA showed an average speedup of 2.7x when compared to a previously DSP-based implementation for the same application.

Soft GPGPU Processors. GPGPU is an array of streaming multiprocessor (SM) where each SM could consist of multiple scalar processor cores (SP). GPGPU has some similarities with vector processor where both target SIMD computation model ; however, GPGPU could support several threads within the same SM while vector processors support only one thread per processor. Another difference is that the memory system in GPGPU is designed to serve numerous running threads with low thread scheduling overhead while vector processors rely on deep pipelining to overcome memory latency.

FlexGrip [13] is a soft GPGPU processor where compiled CUDA binaries are directly executed on the FPGA-based GPGPU without hardware resynthesis. FlexGrip follows Single-Instruction Multiple-Thread (SIMT) model where one instruction is fetched and executed simultaneously by multiple SP cores. It supports the implementation of different number of SP cores per SM processor or different number of SM processor per GPGPU according to the available hardware resources. In the experiments, FlexGrip was implemented for a single SM and 8-SM over Virtex-6 VLX240T board to evaluate five different highly parallel CUDA applications. The results showed a speedup up to 30x for FlexGrip designs when compared to MicroBlaze implementations.

FGPU [10] is another soft GPGPU processor with single level-cache optimized for FPGA. It was developed in VHDL without using FPGA-dependent IP cores to guarantee architecture portability. The authors developed a compiler in order to support applications written in OpenCL [48]. FGPU has an extended MIPS assembly instruction set with additional instructions to support OpenCL execution model. The architecture of FGPU supports several

compute units (CU) where each CU has 8 processing elements sharing the same program counter. FGPU is scalable by paying attention to the ways of implementing the register file, how DSP blocks are used and how the operating frequency is preserved as the architecture scales up. Flexibility is also considered by offering a set of different parameters that can be adjusted to deliver a large space of customizable architectures. In the experimental results, the authors compared 11 applications including three image filters implemented on FGPU to other platforms including : single MicroBlaze soft processor, Cortex-A9 ARM with Neon vector coprocessor and the equivalent HLS implementation. If compared to MicroBlaze implementation, FGPU showed a speedup of 10-47x with 6-22x bigger area but at 3-7x more energy efficient. While both FGPU and HLS-based designs performed so close to each other with less area and power consumption for the HLS-based designs.

Multi-core Reconfigurable Architectures. Authors in [91] developed IPPRO soft scalar RISC processor for image processing applications. Each IPPRO core uses one DSP slice, one BRAM block and 330 slice register. IPPRO cores could be arranged in a multi-core heterogeneous architecture to build SIMD or MIMD computation models. The processor instruction set includes the basic arithmetic and logical operations with different addressing modes. Image processing applications are written in dataflow language named RVC-CAL [66] then the code is converted to IPPRO assembly code. In the experimental results, two different image processing applications were implemented over SIMD-IPPRO architecture : traffic sign recognition [91] and histogram of oriented gradients algorithm [12].

In [115], many-core vision processor architecture was proposed for smart camera applications. The architecture consisted of processing tiles arranged in a grid and connected to each other for pixel exchange while processing neighbourhood image algorithms. Algorithms are coded using a programming model based on tiles where each function is mapped to a single tile. The architecture has several configuration options like for PE type (VLIW4 or RISC), communication topology (4-connected or 8-connected), communication protocol (NoC or P2P), pixel memory (shared or private), etc. Canny edge detection was implemented over Kintex-7 FPGA board by utilizing the proposed many-core vision processor architecture. The application processing chain executed several image algorithms for preprocessing, segmentation and feature extraction. The experimental results showed a fixed frame processing rate of 495 fps for any image size.

Multi-FPGA Architectures. Authors in [103] proposed a modular multi-FPGA platform with four detachable Kintex-7 FPGA modules. I/O interfacing was offered through one FMC interface per module while the FPGA modules communicated with each other through PCIe Gen 3 switch. In the experimental results, the authors showed the implementation of an encrypted H.264 encoder with a peak throughput of 18 frame/sec for HD 1080p image size.

Coarse-Grained Reconfigurable Array (CGRA) Architectures. Pixie [51] is a

heterogeneous Virtual Coarse-Grained Reconfigurable Array implemented on FPGA for image processing applications. VCGRAs [42] are architectures consisting of a large number of processing elements connected to each other by virtual switch blocks for communication. The processing element (PE) can range from a single mathematical function up to a fully RISC processor while the communication network can range from configurable connections between the adjacent PEs up to a fully Network-on-Chip. A tool was developed to create the VCGRA architecture and the interconnection between the PEs. Applications are represented by dataflow graphs where the graph nodes represent PEs while edges show the dependencies. The author demonstrated in the experimental results the implementation of Sobel edge detection on VCGRA grid consisted of 45 PEs and 4 virtual channels.

Dedicated Reconfigurable Architectures. Another class for processing elements is to have dedicated processing elements for a particular image/video processing application. Those architectures are precisely customized for the application and accordingly, optimized for area and power consumption at high-computing performance. This advantage came at the expense of flexibility where architecture redesigning is required for every new application. Architectures can be developed either by coding applications in HDL languages like VHDL and Verilog or by coding them in high-level behavioural languages like C/C++ then high-level synthesis tools are used for realization.

Authors in [95] proposed a multi-core FPGA-based implementation for real-time 2D-clustering image processing. The parallel architecture could be of 4,8 or 16 clustering engines. The critical path was buffered to avoid performance degradation as the number of the clustering cores increased. By profiting from the independent data nature for image/video applications, each clustering core could work in parallel on different parts of the same image. Authors in [100] proposed a dedicated architecture for 8x8 2D-inverse discrete cosine transform (IDCT). The proposed architecture consisted of five pipeline stages with the ability to deactivate the adjacent pipeline stages when having zero additions and multiplications to reduce the dynamic power consumption. The 2D-IDCT was implemented on Virtex-5 FPGA board with a maximum operating frequency of 340 MHz and power consumption of 0.831 W.

Another example for dedicated architectures was proposed in [41] to implement histogram of oriented gradients (HoG) followed by support vector machine (SVM) for real-time multi-scale pedestrian detection. The hardware architecture was fully parallel and pipelined to fulfil the requirements of the real-time detection. In the experimental results, the authors implemented the application over Zynq ZC702 board for an input image of size 1920x1080 at a speed of 60 fps.

2.3 High-Level Synthesis Design Methodology

The idea of generating hardware designs by using High-Level Synthesis (HLS) tools emerged early in the domain of hardware design. It arose from the analogy existed in the software domain where at the beginning, programs were written in machine code then assembly languages were used then finally high-level languages (C, C++, etc) were developed to enhance the software productivity. The first attempts for using HLS tools in hardware design was not successful for different reasons : (i) The quality of the generated results were not promising if compared to the ones obtained by the conventional ways. (ii) Behavioural HDL languages were used as the input language for those tools ; consequently, the user base was limited to include only hardware designers who were already using RTL synthesis tools. In other words, by not considering the users of software background, the HLS market was kept unexpanded. (iii) Intermediate results and design interfaces were not efficiently generated [63] [64].

However, many reasons had motivated researchers from academia and industry to continue enhancing HLS tools. We can mention among these reasons : (i) The huge growth in the silicon capacity pushes towards using HLS tools. (ii) Design productivity is enhanced by reusing behavioural IPs instead of RTL IPs which have fixed architecture and interface. (iii) Recent designs tend to use accelerators and heterogeneous SoCs widely. (iv) Time-to-market constraint usually pushes towards reducing the design time by avoiding long chip design process. (v) In the last ten years, many high-performance applications appeared including financial analysis, scientific computing, bio-informatics and video processing applications. (vi) It is infeasible for software developers to use HDL languages ; thus, offering automated tools to synthesize from C/C++ to RTL was an inevitable design step [25].

The first successful versions of HLS tools appeared in the market in the mid-2000s like Mentor Catapult C Synthesis, AutoESL AutoPilot and Xilinx AccelDSP. Three main reasons shared in that success : (i) Introducing C/C++ and SystemC as an input language for HLS tools ; thus, the developer base was extended. (ii) It was a fast way to get an algorithm synthesized into hardware architecture. (iii) The quality of the synthesized designs was significantly improved by the newly introduced tools [28].

2.3.1 HLS Design Flow

Figure 2.3 depicts the design flow for HLS tools. The design process starts by writing the application in C, C++ or systemC where the development and verification steps are done at a level abstracted from the implementation details. In the C synthesis process, the control and data paths are scheduled and bound to the hardware resources according to the applied optimization directives. After writing the algorithm, the developer can do C simulation to verify the correct semantics of the algorithm before proceeding to the synthesis step. In the

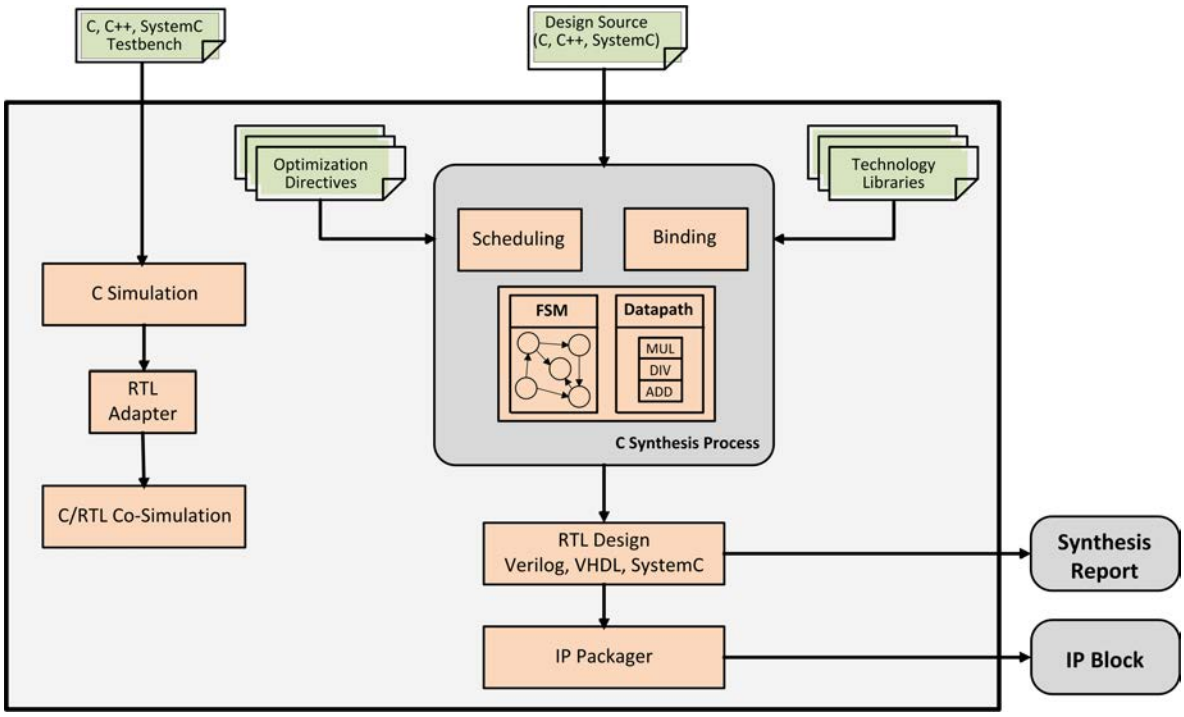

FIGURE 2.3 – HLS tool design flow

binding step, the HLS tool determines what hardware resources will be used to implement each operation. For example, array structures are bound to BRAMs unless the designer used an optimization directive to implement them as individual registers. The tool also offers a set of C libraries that contains optimized functions for hardware implementation. For example, the arbitrary precision data type is one of the predefined libraries. This library allows the designer to define the variables at adjustable data width instead of using the standard data types. For instance, if the range of values for a particular variable is between 0 and 7 ; therefore, we can define it as a 3-bit variable instead of using the standard data type *char* which is equal to 8-bit. In the scheduling step, the tool schedules each operation at which clock cycle it will occur depending on three factors : the length of the clock cycle, the required time to complete that operation and what optimization directive that may be applied for that operation. For example, more operations can be scheduled within the same clock cycle if the clock period is long enough while an operation can be scheduled over several clock cycles for short clock period. After that, the control logic is extracted to create the Finite State Machine (FSM) which controls the execution of the operations in the RTL design. During the synthesis process, the designer should be aware of the following synthesis steps : (i) Each argument in the top-level function is implemented as an individual RTL I/O port. (ii) Each C sub-function is synthesized as a separated RTL module in the final hardware implementation. (iii) By default, loops are kept unrolled ; thus, HLS tool dedicates hardware resources for one loop iteration,

and the hardware will be executed in sequence a number of times equal to the maximum iteration value. By using the unrolling optimization directive, the developer could direct the synthesis tool to perform the loop iterations in parallel. At the end of the synthesis process, *Synthesis Report* is generated to report the performance metrics of the design in terms of hardware resources, execution time in clock cycles and if the timing constraint to schedule the operations within the defined clock period is met or not. After analyzing the report, the designer can modify the optimization directives to refine the implementation for better performance. After obtaining the RTL design, C/RTL co-simulation can be conducted to verify that the RTL implementation is functioning identically to the original C code. Finally, the RTL files are packaged together to produce the IP block [111].

HLS tools do not support every single C/C++ functions for hardware implementation. For example, dynamic memory allocation, function recursion, system calls and File I/O operations are not supported by HLS tools. For that reason, there are two different design methods with HLS synthesis tools : (i) A previous C/C++ reference design is ported to hardware implementation. In this case, the developer starts from a previously written C/C++ code where the role of the designer is to modify the code sections which are not supported for synthesis, then to optimize the code by adding the optimization directives to achieve the design goals. Top-down design flow is more practical in that situation because the design is accelerated as one single function. (ii) Designing from scratch where the designer starts developing his code with full awareness that the written code will be directed for hardware implementation. Top-down, as well as bottom-up design styles, are feasible. In bottom-up, the designer can start accelerating the sub-functions in the application with the ability to expand the accelerator to include the other functions.

2.3.2 HLS Research Directions

We can distinguish two research directions regarding high-level synthesis design methodology : (1) Research contributions tend to enhance the Quality-of-Results (QoR) of the implemented hardware designs through better C-to-RTL transformations (memory partitioning, loop unrolling, loop pipelining, ... etc). (2) Research works which present the hardware implementation for different high-performance applications. In these contributions, HLS designs are/are not compared to hand-written HDL designs implemented for the same application to evaluate the design quality in terms of hardware cost, performance and design time for both approaches.

2.3.2.1 Contributions for efficient hardware implementations (Quality-of-Results)

Memory Partitioning. Accessing data from memory could limit the application performance if multiple memory accesses are requested simultaneously from the same memory

bank. Thus, memory partitioning is used to increase the number of the available physical ports for array access. Consequently, multiple simultaneous memory accesses could be scheduled in parallel for better performance. In [34], three different memory partitioning schemes (lattice-based, hyperplane-based and cyclic) were discussed. Figure 2.4 depicts the three memory partitioning schemes for image resizing (4 :1) application. In Figure 2.4(a), pixels are distributed in cyclic way where multiplexers on the address port are required to resolve the conflict of reading two different elements from the same memory bank. In addition to that, flip-flops are needed to buffer the first two pixels till all data is loaded to start computation. Figure 2.4(b) shows hyperplane distribution manner where for the even index (i), elements are mapped to memory banks B1 and B3 while for the odd index (i), elements are mapped to memory banks B2 and B4. This bank alternation leads to having the same memory port is referenced by two different positions. Figure 2.4(c) depicts the lattice-based distribution where memory conflicts and bank switching are avoided for better data-path and less area overhead. Authors in [57] proposed two different approaches for memory partitioning (vertical and mixed memory partitioning) where different memory accesses in different arrays can be scheduled simultaneously to non-conflicting memory banks. The algorithms were developed using AutoESL and tested experimentally on a set of medical images processed over Virtex-6 FPGA board. The results showed a gain in speed-up, power savings and area reduction of 15.8%, 32.4% and 36% respectively. Authors in [104] proposed a generalized memory partitioning method based on a polyhedral model to solve the problem of bank access conflict where the results showed up to 20% reduction in the used BRAM and Slices.

Loop Transformations. It is hard to write an algorithm without including loop structures. For that reason, HLS tools support the hardware synthesis for loops and offer different loop optimizations like pipelining, unrolling or merging for better performance. In [60], authors developed a parametric polyhedral model to consider the cases of uncertain memory dependency (i.e. the array index is parameterized by undetermined variable) or when it is not uniform by varying between loop iterations. Automatic code transformations to spilt the original loop into sub-loops which are then synthesized normally by the HLS tool. The results showed that the number of cycles per iterations had an average speed-up of 4.3x after applying loop splitting with an average hardware overhead equalled to 95% for LUT, 83% for FF and 12% for DSP. Authors in [86] compared between loop coarsing and loop tiling for image filter applications. In loop tiling, the image is split into separate regions which are then processed in parallel; while in loop coarsing, the input pixel data is aggregated in structures named superpixels where the loop kernels are only replicated inside a single accelerator. The experimental results showed an advantage by using loop coarsing in terms of hardware resources and performance compared to loop tiling. In [92], the authors transformed nested loops of triangular iteration space into rectangular ones for better loop transformation to enhance the overall performance. The reported results showed a performance improvement

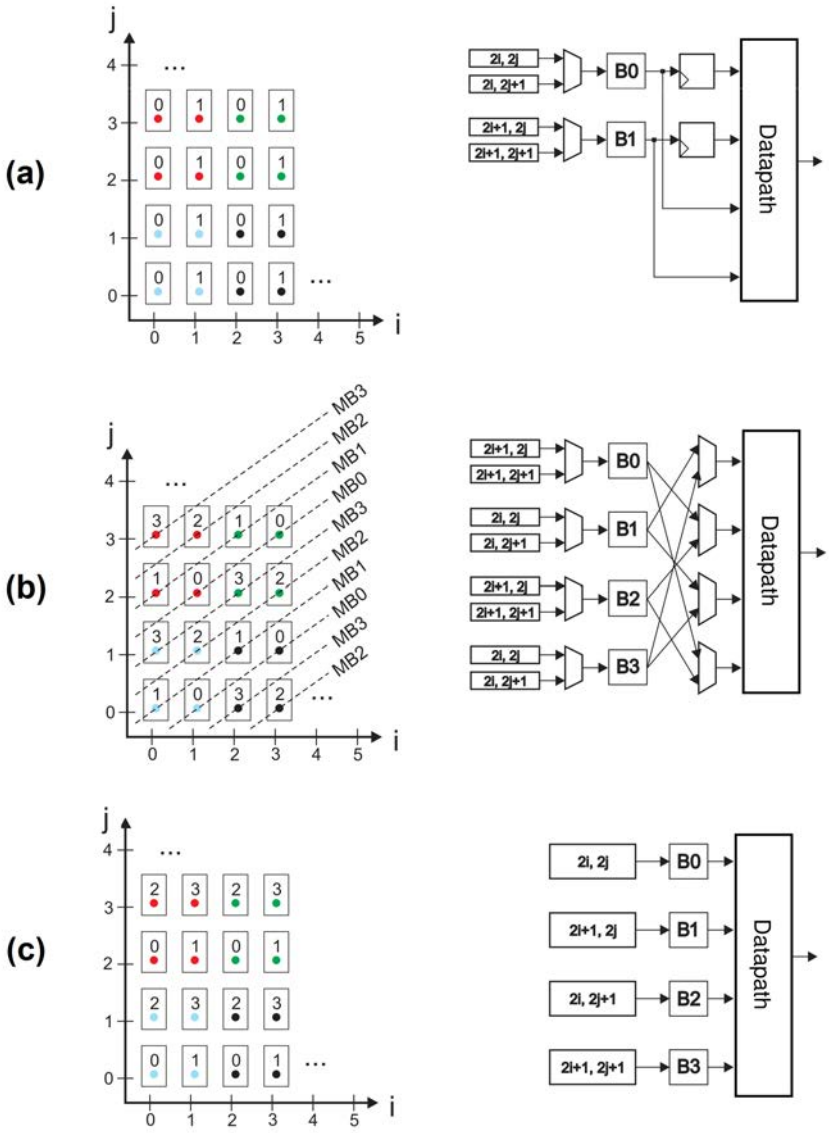

FIGURE 2.4 – Memory partitioning schemes in [34] and their area overhead. (a) Cyclic partitioning. (b) Hyperplane-based partitioning. (c) Lattice-based partitioning.

ranged between 33% to 100% for different benchmarks. While authors in [24] used loop unrolling to decrease the area overhead arose by partitioning multidimensional arrays. Their method was based on reducing the logic used for routing the data from the memory banks to the processing blocks by decreasing the bank switching effect. ElasticFlow architecture was proposed to pipeline nested loop with dynamic-bound inner loops. It distributed the execution of the inner loops over an array of processing units. To keep the memory footprint balanced, a customized banked memory architecture coordinated the memory accesses among the processing units [58].

Dynamic Data Structures. Current HLS tools do not support dynamic memory allocations in addition to that pointer operations are limited ; consequently, dynamic data structures are not supported for hardware synthesis. Authors in [112] proposed an open source synthesizable library named *SynADT* for linked lists, binary trees, hash tables and vectors to allow the usage of abstract data type in HLS. The results showed that for data structures of size 128 KB and 10 MB, HLS data structures were faster by 1.67x and 1.35x over Microblaze processor using default malloc ; while it was 6.7x and 8x slower than an ARM processor running at 677 MHz with 512 KB L2 cache.

Recursion. It occurs when a function makes a call to itself either directly or indirectly via another function. Recursion is not supported for synthesis by the HLS tools ; thus, designers are forced to remove manually any recursion functions before code synthesizing. A C++ library named Embedded Domain Specific Language (EDSL) was developed to write recursive functions. By using EDSL, recursive functions were written by utilizing the C++ front-end of the HLS compiler to build the state machines and stacks while the presented code to the back-end compiler is completely synthesizable. The experimental results showed that using EDSL library for describing recursive function is competitive to the manually converted code in terms of hardware resources and performance [99].

Hardware Debugging Facilities. HLS tools offer software debugging where the HLS code can be executed on a workstation and debugged by using the standard software debugging ways. However, some errors can not be discovered using the software debugging tools like : (i) Errors happened at the interfaces between the blocks in the design. (ii) It is required in some applications to debug the block on runtime by using realistic input data. (iii) The C-code is transformed to HDL code during the HLS synthesis flow. Consequently, software debuggers are not able to discover errors in the hardware design due to tool bugs during transformation. Commercial hardware debugging tools like (Chipscope and Logic Analyzer) provide the visibility of the signals inside the generated RTL design without back-tracing to their source in the original C-code. Accordingly, using those tools by the hardware designers to understand the circuit and do back-tracing is time-consuming process which opposes the idea for which HLS tools are used. While for software designers, debugging at that level is nearly infeasible for them.

Authors in [36] proposed a framework to do in-system debugging by capturing the interaction with the other blocks in the system while automatically recording the hardware signals in a manner similar to software debugging. For signal-tracing, the experimental results showed that the framework increased the length of the execution trace that can be recorded by 127x if compared to the embedded logic analyzer. *Inspect* was another framework integrated with LegUp HLS tool for offering HLS debugging facilities. It had a software perspective to debug HLS-generated blocks by giving the user the familiar debugging execution management (step, run and break) with the ability to inspect variables. Two modes of debugging were

supported either by simulating the generated RTL design or by executing the hardware on FPGA device (hardware debugging). Bugs had been inserted manually into the generated RTL for a set of benchmarks where the automated discrepancy detector could discover them [21].

2.3.2.2 Contributions for demonstrating HLS-based applications

In the last five years, many HLS-based hardware implementations for applications from different domains were published to reflect the degree of maturity that HLS synthesis tools reached. We can highlight the following points regarding those contributions : (i) Authors tended to compare between RTL-based and HLS-based designs for the same application. In those comparisons, RTL-based designs showed better hardware utilization while HLS-based designs showed shorter design time with a short learning curve for the new users. It is worth mentioning that the increase in hardware cost for HLS-based designs is within the acceptable margins. Accordingly, authors usually commented by expressing their acceptance to that increase rather than spending months to obtain a handwritten RTL-based design for the same application. This desire reflects clearly the role of time-to-market constraint which adds another pressure on the design cycle. (ii) HLS design methodology gives the designer the flexibility to compare different design alternatives by applying different optimization directives without spending too much design efforts. For that reason, it is hard to find a research work that built an HLS-based architecture for a particular application without exploring the other possible implementations for best area utilization, best execution time or for best power consumption.

Authors in [74] showed the hardware implementation of two algorithms used in microwave imaging for detecting breast cancer. Hardware acceleration was recommended to improve the processing time and to reduce power consumption. HLS was used to explore around 100 alternative implementations which were different in their cost-performance profile. Storing FPGA bitstreams and software binaries on non-volatile memories is costly. Accordingly, HLS was used to develop four lossless compression decoders to reduce the usage of the external memories. The algorithms were evaluated on Zynq ZC706 operating at 200 MHz where the results showed 30% less for software start-up time and 70% decrease in the time required for partial reconfiguration [113]. Key-value store memcached server was another application developed by HLS tools over Virtex-7 VC709 board. The HLS synthesis results were compared to a previously implemented RTL design to deduce that the latency was reduced by 30% with a decrease in resource utilization reached to 20% for LUT and 35% for FF. In addition to that, HLS code was developed at half the time required by the RTL-based design.

FPGA-based 10 GbE active network probes were designed using HLS tool and open source platform to measure several network parameters like throughput and delay. The design showed 2% more accurate than software solution [81]. Authors in [44] compared the results of designing cryptographic modules based on Advanced Encryption Standard (AES)

using HLS and hand-written VHDL code approaches. Their study demonstrated that both approaches led consistently to comparable results regarding area and clock frequency. In [72], the authors reported their experience while targeting FPGA implementation dedicated to controlling algorithms for power converters using HLS approach. They highlighted that such approach significantly simplified the design and simulation process without requiring specific HDL design skills.

2.3.2.3 Surveys

Authors in [68] made a qualitative comparison for twelve HLS tools according to several criteria including : (i) Source language and ease of implementation : it is recommended to close the gap between algorithm design and hardware design by offering easy-to-use high-level synthesis languages for the people who have no hardware experience. (ii) Tool complexity, user interface and documentation : in order to wide the user base, the tool should have a friendly-style GUIs with enough documentation and tutorials for having a smooth learning curve. (iii) Tools capabilities : HLS tools should support defining variables in flexible data width format for better hardware implementation. The ability to tune the optimization options to explore several implementation varieties is another significant strength for HLS tools. (iv) Verification : HLS tools ease the verification process by generating test benches together with the design. During verification, the testbench includes the source code as a golden reference while applying the test vector to the generated design for comparison. (v) Evaluating the results quality concerning hardware resources and latency. In the experimental results, Sobel edge filter of kernel=3x3 was used as a test application where each criterion was marked by a number ranges between 1 (bad) to 5 (excellent). The evaluation for each HLS tool was represented on a spider web diagram for easy visual comparison. The authors concluded with several recommendations to improve the design productivity of the HLS tools.

In [14], authors did an empirical comparison between four different HLS languages which were used in four different tools (Bluespec System Verilog, Altera OpenCL, LegUp based on C language and Chisel based on Scala functional language). The four tools were used to accelerate three common database operations : sorting, aggregations and joins by implementing four different algorithms (bitonic sorting, spatial sorting, median operator and hash joins). The authors detailed their experience for using these four high-level languages by showing the trade-offs. They concluded that there is no obvious election between them, but it depends on a set of requirements including programmability, area utilization and the obtained performance.

In [106], authors compared three commercial HLS tools (Vivado HLS, Altera OpenCL and BluespecBSV) in addition to two more academic tools (LegUp [22] and ROCCC [101]). The authors gave an overview with some details about their user interaction experience for each tool. During the experiments, two applications (dilation filter of kernel = 3x3 and AES

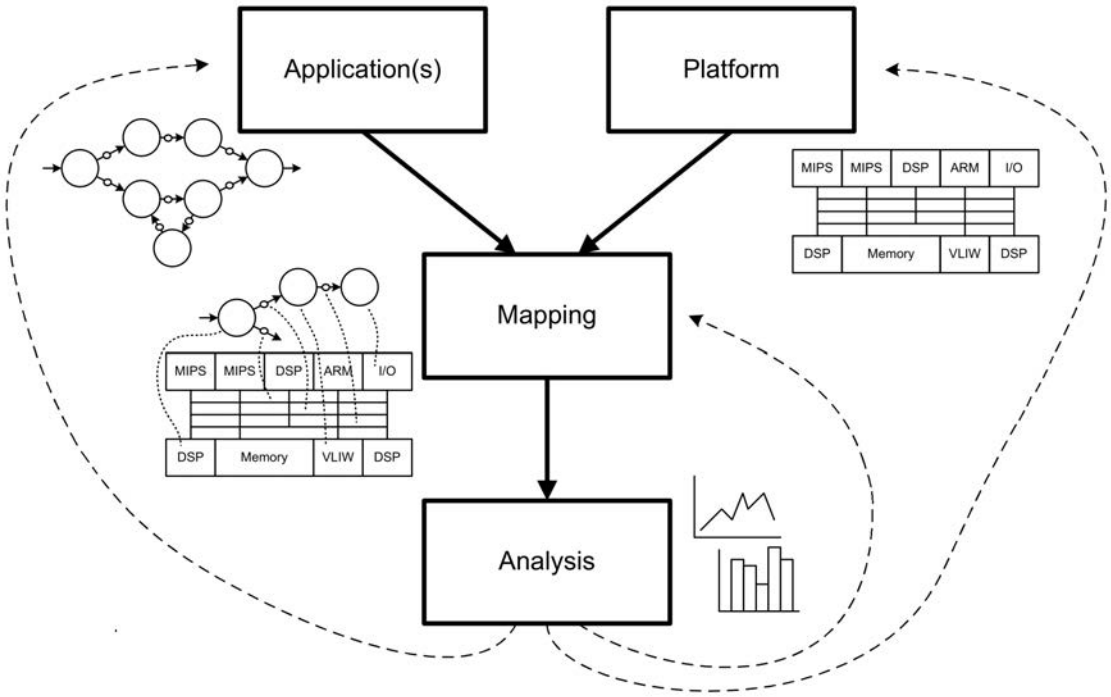

FIGURE 2.5 – Design space exploration with the Y-chart paradigm [53].

encryption algorithm) were used. The implementation of dilation filter was commented by the authors' practical experience for how to compile it by showing the strength and weak points for each tool. Performance, area utilization and power consumption were reported for both applications synthesized by LegUP and ROCCC.

Another recent study extended the set of benchmarks to include different domain applications (CHStone benchmark suite [39]) to evaluate three academic tools (LegUP [22], BAMBU [77] and DWARV [71]) in addition to a commercial HLS tool (the name was not mentioned in the survey). Two sets of experiments (standard-optimization and performance-optimized) were conducted to the HLS tools to report four performance metrics (number of clock cycles required for execution, maximum reported frequency, the total execution time for each application and the hardware utilization in terms of LUT, FF and BRAM). The authors reached the following conclusions : (i) There was no HLS tool which performed the best for all the benchmarks. (ii) The academic tools were not far away in their performance metrics if compared to the commercial ones. (iii) The performance improved on average between 1.6-2x when the code was tuned by the HLS constraints in the performance-optimized experiments.

2.4 HLS Design Space Exploration

Figure 2.5 depicts how the design space is explored in the Y-chart paradigm where one system implementation is selected from a set of alternatives. In this approach, the application and architecture specifications are explicitly separated from each other. The performance of architectures is analyzed for a given set of applications to provide the quantitative data that the designer will use to take a decision. In the mapping step, we define for each function in the application on which platform it will be executed (allocation and binding) and at which moment it will be scheduled (scheduling). The performance metrics are then evaluated regarding area, execution time, power, etc. Iteratively, improvements are applied by changing the architecture or by changing the way the application is described or by changing the way of mapping until finding the solution which satisfies the design requirements [53].

Figure 2.6 shows the abstraction pyramid which describes the architecture modeling at different abstraction levels in connection with three factors : the modeling effort, evaluation effort and accuracy. On the right-hand side of Fig. 2.6, the axis modeling cost indicates that the modeling effort increases as we go down along the abstraction pyramid. Therefore in the performance analysis step, obtaining very accurate performance numbers comes at the expense of considering a more detailed model with long evaluation time. While less accurate numbers can be achieved at a shorter time when high-abstracted models are considered. From the abstraction pyramid, we can conclude that exploring the design space while using high-level synthesis tools, will accelerate the exploration process by early pruning the design space so that by moving down the pyramid, the design space to be evaluated becomes smaller [50]. In addition to that, HLS tools give us the possibility to generate a set of micro-architectures of different area versus performance trade-off without having to modify the original behavioural code itself.

HLS design space exploration (HLS DSE) can be classified in different ways. One classification divides the DSE into two classes : (i) DSE inside the HLS tool. (ii) DSE with HLS tool. The first class focuses on applying DSE to the internal tasks of the HLS tools (allocation, binding and scheduling). Each task is controlled by a set of different factors which have a significant impact on the performance metrics of the resulted hardware. Some research works under this class are [79] and [87]. The second class considers the HLS tools as a black box and explore the design space of the optimization parameters offered by the HLS tools. This class can be further subdivided into : (i) HLS synthesis directives. (ii) Resource sharing. For the first subclass, the HLS directives are inserted into the behavioural code as comments to affect the final synthesized micro-architecture. For example, loops can be pipelined or not with complete or partial unrolling. Arrays can be also completely or partially partitioned with the possibility to be mapped to registers or BRAMs. These optimization varieties generate a design space of different combinations that can be explored for the same application. For resource sharing, a single functional unit (FU) can be shared among different operations in

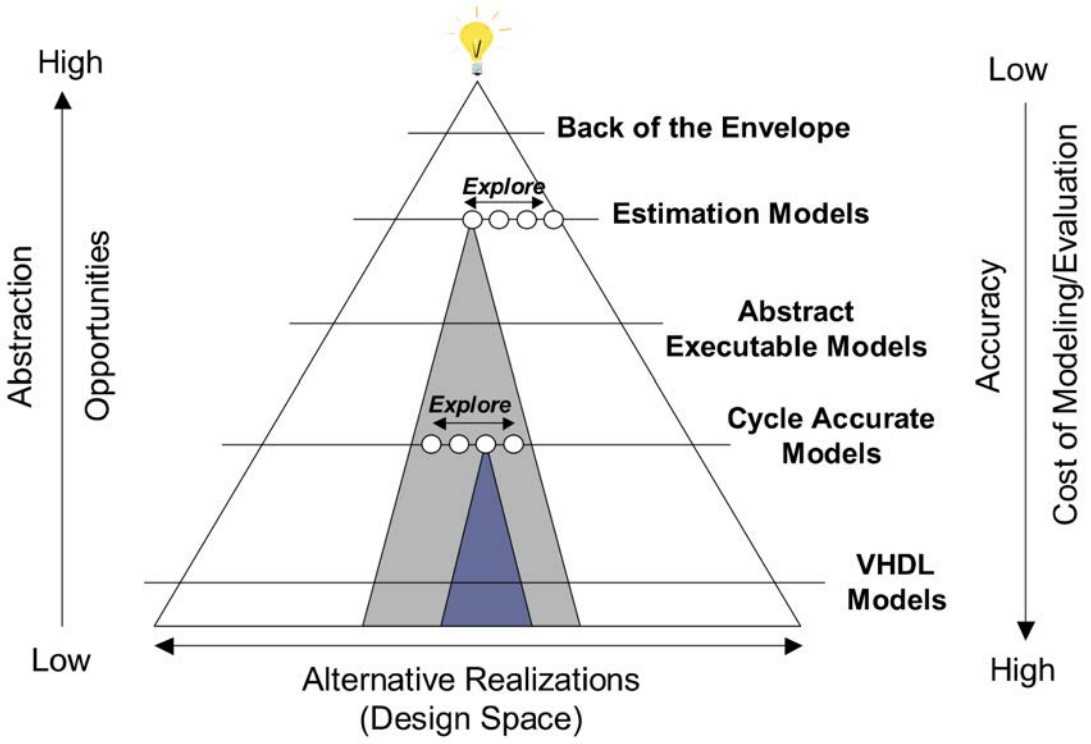

FIGURE 2.6 – Design space exploration with the Y-chart paradigm [50].

the source code. This is achieved by inserting multiplexers at the inputs and outputs of the functional unit. Using resource sharing produces a design space of different implementations which vary from an architecture that has one single shared FU to a fully-parallelized one.

Authors in [82] applied HLS directives then resource sharing to reduce the design space to be explored. They proposed a probabilistic method to accelerate the DSE process by calculating the probability of each generated architecture then continue exploring only the designs of the highest probabilities. Their experimental results showed an acceleration of 12x in the DSE process. Resource sharing acts differently for both ASIC and FPGA. In ASIC, resource sharing reduces the total area of the design while for FPGA, it could act oppositely because of the size of the inserted multiplexers consume a lot of logic resources. For that reason, authors in [83] proposed a method to force the cost of resource sharing to be larger than that of the used multiplexers by fixing the bitwidth of the internal variables. This approach came at the expense of introducing overflow errors in the design. The experimental results showed that the percentage error differed from one application to another. Thus, the designer should estimate if the error percentage is acceptable or not in his application. A framework for HLS DSE was presented in [76] which exploited the loop array-dependency to reduce the DSE time. The results showed that the framework gave the same quality

of result as the exhaustive DSE approach while lowering the exploration time with an average of speed-up of 14x. Another framework used sequential model-based optimization (SMBO) to select the HLS directives automatically. During optimization, a model of the function was constructed using machine learning methods. From the experimental results, the convergence to the optimal HLS directive settings was improved by using transfer-learning mechanism in the SMBO model [61]. Lin-Analyzer [118] did rapid design space exploration for various HLS pragmas like loop pipelining, loop unrolling and array partitioning without the need for doing RTL implementations. Programs were represented in dataflow graphs by using dynamic data dependence graphs (DDDG). DDDG are acyclic directed graphs where nodes represent operations while edges represent data dependence between the nodes. Lin-Analyzer scheduled the graph nodes according to the resource constraints to obtain an early performance estimation. For validation, 10 different applications were tested on Xilinx ZC702 FPGA board. Another classification for HLS DSE is based on the algorithm used during the design space exploration like using genetic algorithm [35], simulated annealing [62], ant colony [82] or machine-learning techniques [59].

2.5 Design Productivity

Design productivity gap arises due to the persistent growth of silicon capacity in comparison with the available design capabilities. Increasing the level of design abstraction by using high-level synthesis tools is an inevitable solution to minimize that gap. Authors in [75] defined design productivity as not only producing designs in short design time but also at acceptable design quality. In that work, they compared CAPH HLS language with VHDL to evaluate CAPH design productivity. Three metrics were introduced to assess an HLS tool : (1) The gain in Non-Recurring Engineering (NRE) to evaluate design efficiency. NRE is defined as the time required for designing, verifying and integrating a new system design. (2) The loss in the design quality in terms of the hardware resources (LUT, FF, BRAM, DSP) and the operating frequency to evaluate the implementation quality. (3) The HLS design productivity is defined as a ratio between design efficiency and implementation quality.

HLS tool vendors work to shorten the design cycle by offering the efficient RTL implementations for a set of functions which can be used directly in building applications. Tool vendors show a special care to provide libraries for image/video processing applications. For example, HLS Video Library from Xilinx offers the RTL design for some existing OpenCV functions [111]. Another example, HDL Coder from MathWorks provides HDL descriptions for different signal and image/video processing functions [65]. Authors in [85] implemented a lightweight C-based library integrated with Vivado HLS to build hardware accelerators for stream-based image processing applications. Point processing applications like brightness

filter as well as local image operations as Sobel filter are supported by that library. A Very High Level Synthesis (VHLS) framework for image processing applications is presented in [18]. Firstly, the algorithm is described in Matlab or OpenCL ; then in the next step, control and data flow graphs are extracted. At the end, HLS tool uses these graphs to obtain the RTL design of the application.

2.6 Positioning

In this section, we will position our contributions with respect to the described state-of-the-art. FPGA technology was chosen among other available solutions as the developing platform for our presented contributions for different reasons : (i) FPGAs are reprogrammable devices that give us the chance to architect our design in an efficient way to implement parallel video designs that fit for autonomous vehicle applications. (ii) FPGAs can deliver high-computing performance to process complex video processing applications under real-time constraints. (iii) FPGA devices are characterized by their low power consumption if compared to GPU or CPU ; accordingly, they are good candidates for battery-based applications. (iv) Using high-level synthesis tools for FPGA design enhance the design productivity. In addition to that, software developers can profit from hardware acceleration without requiring hardware expertise or mastering Hardware Description Languages (HDL).

Throughout the chapter, we reviewed different architectures to build video processing applications on reconfigurable technology like soft vector processors, soft VLIW processors, soft GPGPU processors, dedicated processors, etc. We can classify our architectural contribution under the class of dedicated processors. We made that choice for the following reasons : (i) Performance is our main design goal because we are targeting autonomous vehicle domain where high-performance under real-time constraints are recommended for safety conditions. (ii) Customizing the hardware resources for building exactly the required application to have the power consumption as minimum as possible.

Certainly, losing flexibility was paid for those advantages. However, flexibility could be partially recovered by automating the design process to explore different design alternatives. In order to achieve that, we did the following steps : (i) Performance metrics (area, power consumption and execution time) were estimated at high design level to early explore the design space for the best candidate designs. (ii) We generated the corresponding parallel architectures for the candidate designs for experimental validation. By reviewing high-level synthesis (HLS) tools, it was clear that the last generations succeeded to build efficient hardware implementations in different domains. In addition to that, these implementations were realized during a short design cycle when compared to HDL-based designs. For that reason, we based our implementation for the stereo matching algorithm on HLS tools.

2.7 Conclusion

In this chapter, we have reviewed different reconfigurable solutions for realizing video processing applications. These solutions utilize the reconfigurable fabric in two different ways : (i) To build soft-core processors like vector, GPGPU, VLIW, many-core, etc, to allow software programmability and accordingly, they are more flexible when the application changes. (ii) To build dedicated hardware architecture for a particular application to have area- and power-customized architecture. In addition to that, the reasons behind the rapid development of high-level synthesis tools were explained. We showed how the researchers contributed to either improve the synthesis quality of HLS tools or to demonstrate the implementation of high-performance applications based on HLS approach. In HLS design flow, several architectural optimizations can be combined to deliver different solutions ; thus, design space exploration tools are mandatory at that level to narrow the design space for further synthesis steps.

Chapter 3

Flexible Parallel Architecture for Video Streaming Applications

3.1 Introduction

A huge amount of data is accessed and processed in video processing applications. Memory bandwidth can limit the processing rates if the images are firstly stored to off-chip memories then read for processing. Instead of that, it is recommended to process the input pixel stream once the image sensor generates it. This challenge is augmented by considering today's image sensors which deliver high-resolution images at high frame rates. To address that challenge, reconfigurable computing is suggested to build flexible parallel architectures for video processing applications. In this chapter, we will present a generic model for pixel distribution/collection dedicated for streaming video applications. By the help of this model, the input pixel stream is directly distributed over the parallel processing elements without storing them to an external memory. In this model, we defined the parameters required to have a flexible pixel distribution in both vertical and horizontal directions. To decrease the development efforts, the hardware architecture for the pixel distributor/collector is generated automatically. In the experimental results, we will show the parallel architecture for two different video streaming applications : Video downscaler (16 :1) and convolution filter for an input HD image stream of 60 frame/s.

The proposed parallel video processing architecture will be modified in conjunction with frequency scaling as a technique for reducing power consumption. In that case, the processing elements will operate at frequencies slower than the other parts of the system. However, to keep the same processing rate, the number of the processing elements will increase to compensate for the scaling in the frequency. We will derive the required equations to ease the calculation for the level of parallelism and the maximum depth for the FIFOs used for clock domain crossing. Accordingly, a design space will be formed including all the design alternatives for the application. The preferred design will be selected in aware of how much hardware it will cost at what power reduction it can satisfy. In the experimental results, we will implement two different video streaming applications : Video downscaler (16 :1) and AES encryption algorithm to verify our approach.

3.2 Cost-effective Solution for Autonomous Vehicles

Figure 3.1 depicts an example of the autonomous shuttle from NAVYA Technology. In this shuttle, multiple image sensors are normally installed for obstacle detection, tracking and classification. In this application domain, image processing is usually done under real-time constraints with an increasing request for higher frame rate and better resolution for better obstacle detection. As previously mentioned in Chapter 1, FPGA technologies are good candidates to fulfil those system requirements through their characteristics which can be summarized in reconfigurability and their ability to realize massively parallel architectures. In this chapter, we will mainly focus on the following aspects : implementing dedicated parallel

FIGURE 3.1 – Autonomous shuttle from NAVYA

architecture, introducing a model for pixel distribution/collection and using parallelism as a technique for reducing power consumption. In order to achieve our objective, we have considered the following setup for experiments.

3.2.1 Experimental Setup

Figure 3.2 depicts the main components of video processing system architecture where a color image sensor VITA-2000 [73] is configured for high definition frame resolution (1080p60). The sensor is connected to Xilinx Zynq ZC706 FPGA board [108] through Avnet IMAGEON FMC module [15]. It is a CMOS image sensor [33] that captures pixels in a monochrome format (10-bit size for each pixel) where the raw captured pixels are then converted to RGB format (24-bit) through the image preprocessing pipeline. *Defective Pixel Correction* (DPC) filter is the first stage in the image preprocessing pipeline where defective pixels are removed. To generate an RGB color image, *Color Filter Array* (CFA) filter [109] is used to restore the other missing two colors based on the neighbouring pixels . Some other filters such as (gamma, noise, edge enhancement, etc) can be added to improve the quality of the input image. For signal synchronization, *Video Timing Controller* (VTC) is connected at both ends of the processing channel for detecting/generating video timing signals. The block named *Parallel image processing elements* represents a set of parallel processing elements that implement a particular image processing algorithm. In the next section, we will explain how the input stream is distributed over the processing elements for parallel processing then how the resulted pixels are gathered back to form the output stream. Before showing the processed image on the screen, *RGB-to-YCbCr422* block converts the RGB pixels into YCbCr 4 :2 :2 format then

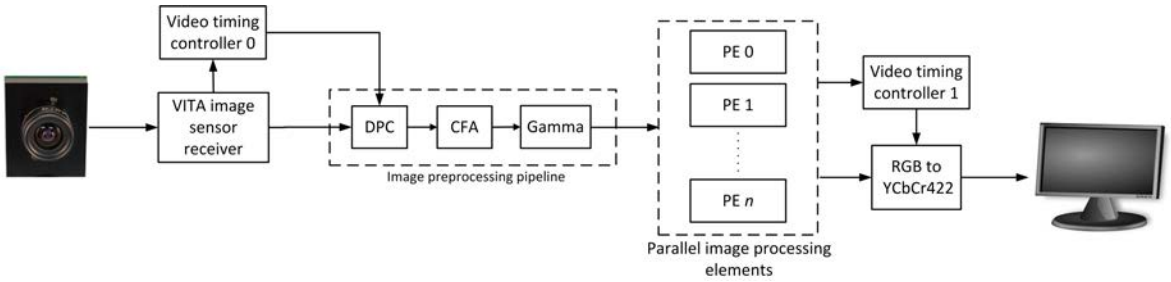

FIGURE 3.2 – Video processing system architecture

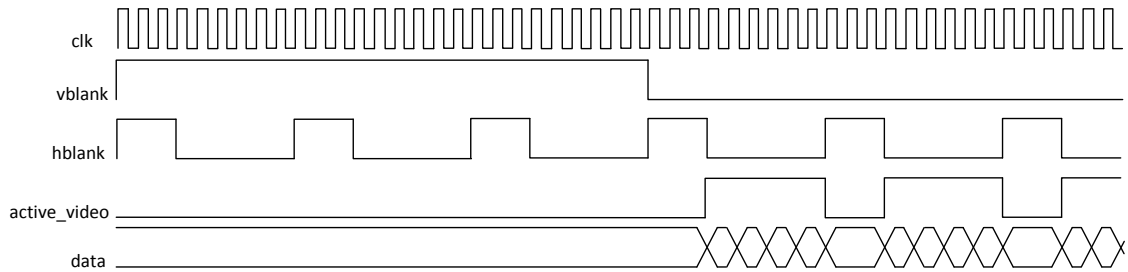

FIGURE 3.3 – Video timing signals

the pixels are streamed to an HD monitor with the correct timing signals. Figure 3.3 shows the timing signals accompanied by the video stream : (i) The start of the frame is observed when *vblank* signal is high. (ii) Horizontal blanking (*hblank*) indicates the start of a line in the frame. (iii) A valid pixel is presented when *active_video* signal is asserted to high. In the following section, we will start by introducing our generic pixel distribution model.

3.3 Generic Pixel Distribution Model

The main concern of this section is to propose a pixel distribution architecture that can deal with various input frames and sizes for macro-block. Firstly, we will introduce the parameters of the generic pixel distribution model. Secondly, the proposed hardware architecture will be detailed, and we will describe the finite state machines that control the architecture. Finally, we will show our experimental results by presenting the synthesis results for the pixel distributor with different input frames and sizes for macro-block. In addition to that, we will demonstrate the implementation of two different video applications : video downscaler (16 :1) and convolution filter of kernel size = 3x3.

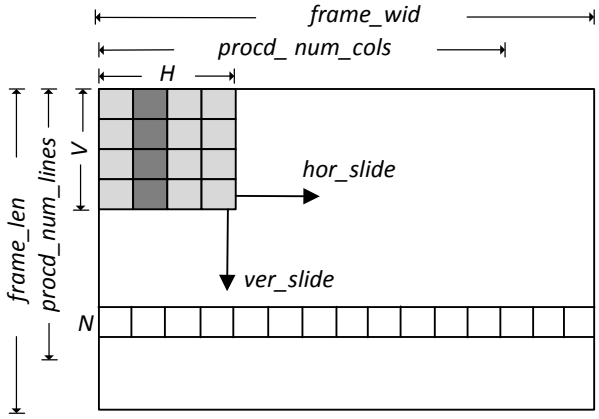

FIGURE 3.4 – Pixel distribution model

3.3.1 Model Parameters

The required parameters to understand the pixel distribution model are illustrated in Fig. 3.4 and described as following :

- A frame is of width *frame_wid* and length *frame_len*.
- A macro-block is the basic processing structure of length V and width H such that $V \geq 1$ and $H \geq 1$.
- A macro-block can move horizontally by a step = *hor_slide* and vertically by a step = *ver_slide* such that $1 \leq ver\_slide \leq V$ and $1 \leq hor\_slide \leq H$.
- *procd_num_lines* is the number of lines processed in one frame. If *procd_num_lines* < *frame_len* then (*frame_len* - *procd_num_lines*) lines aren't processed, such that $procd\_num\_lines = V + \lfloor \frac{frame\_len - V}{ver\_slide} \rfloor * ver\_slide$.
- *procd_num_cols* is the number of pixels processed in one line. If *procd_num_cols* < *frame_wid* then (*frame_wid* - *procd_num_cols*) pixels aren't processed, such that $procd\_num\_cols = H + \lfloor \frac{frame\_wid - H}{hor\_slide} \rfloor * hor\_slide$.
- N is the index of a line in the frame such that $1 \leq N \leq procd\_num\_lines$.
- Since each line is stored in a separate buffer, then V buffers are needed. We define B as the buffer index of a given line such that $B = (N \bmod V)$.

3.3.2 Pixel Distributor Architecture

The role of the pixel distributor is to write the input video stream to the *line buffers* then to distribute the stored pixels in the form of macro-blocks according to the required size (H x V). Figure 3.5 shows the interface and the internal block diagram for the pixel

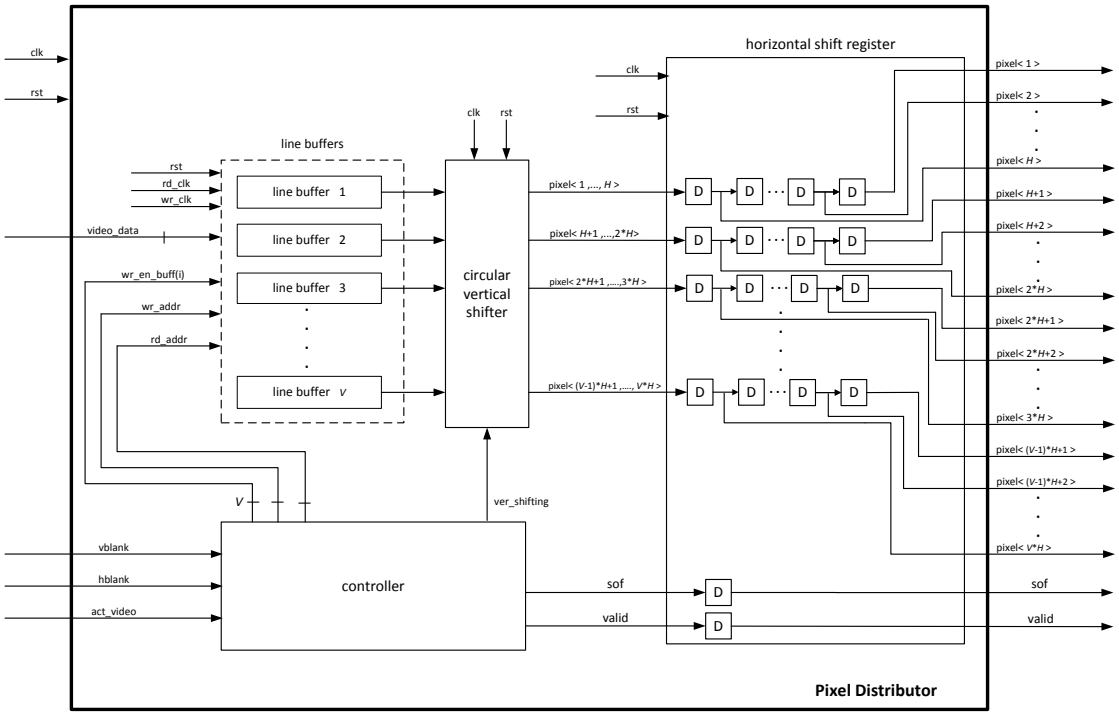

FIGURE 3.5 – Pixel distributor structure

distributor. The interface consists of : (i) The ports for the input video signals which are *vblank*, *hblank*, *act_video* and *video_data*. (ii) The output ports are equal to the number of the pixels in the macro-block (i.e. HxV output ports). (iii) *sof* signal which is asserted to high with the first macro-block to designate the start of the frame. (iv) *valid* signal which is asserted to high with every macro-block to indicate the presence of a valid block at the output ports.

The pixel distributor consists of the following internal blocks : (i) *line buffers* for storing the input pixels. (ii) *circular vertical shifter* for shifting the pixels circularly in the vertical direction. (iii) *horizontal shift register* for shifting the pixels horizontally. (iv) *controller* for asserting the required control signals according to the current state of the system. For example, *controller* asserts one of the *wr_en_buff* signals to enable writing in one of the line buffers at a specified address *wr_addr*, while it loads *rd_addr* for reading operations. Also, *controller* assigns *sof*, *valid* and *ver_shifting* signals to mark the start of the frame, the presence of a valid macro-block or for shifting the pixels vertically.

A column of pixels is passed to *circular vertical shifter* as soon as its last pixel was written to *line buffers*. *horizontal shift register* shifts each pixel horizontally so that after *hor_slide* shifts for the first pixel in the macro-block (i.e. pixel<1>), the *valid* signal is asserted to indicate a valid macro-block at the output ports of the pixel distributor. The image line of

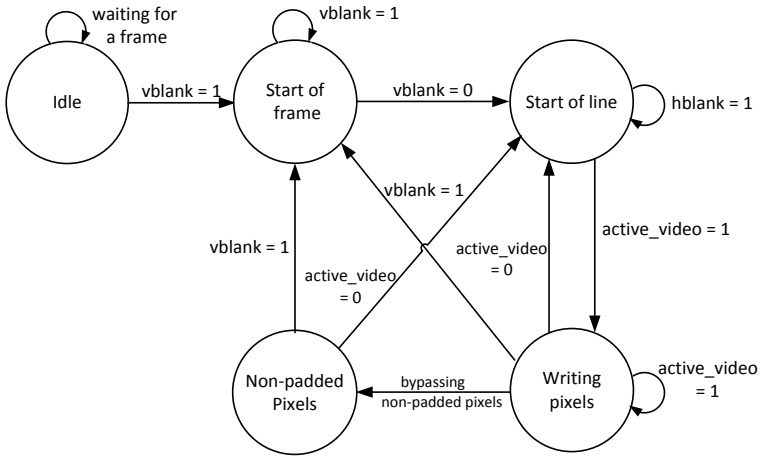

FIGURE 3.6 – Writing process finite state machine

index V+1 will be stored in the first line buffer (Buffer index $B = N \bmod V$). If *ver_slide* $< V$, then the image line of index V+1 will have an order in the macro-block rather than being the first line. In that case, the output of the *line buffers* should be shifted vertically in a circular way to put back the lines of the macro-block in their correct order.

The distribution of pixels is executed within the boundaries of the frame ; therefore, for the neighborhood pixel applications like median filter, the border pixels are not distributed since a part of their macro-blocks lie outside the frame boundaries. In such situation, we decided not to process these border pixels since the percentage of the unprocessed pixels (i.e. the error rate) is within the acceptable range. for example, when the input frame size is 1920x1080, the percentage of the unprocessed pixels is 0.29% for a macro-block of size=3x3, 0.58% for 5x5 and is 0.86% for 7x7.

3.3.3 *Controller* Finite State Machine

Figure 3.6 shows the finite state machine for the writing process, (i) The system starts from the *idle* state waiting for an input stream. (ii) During the *vblank* period, the system waits in the *start of frame* state. (iii) Then, it remains in the *start of line* state while the *hblank* signal is active. (iv) The system rests at *writing pixels* state during the operation of writing pixels. (v) If *procd_num_cols* < *frame_wid* then the system will transit to *non-padded pixels* state to bypass the rest of the pixels of the line ; otherwise, it will transit to *start of line* state to process the next line or to *start of frame* state to process the next input frame.

Figure 3.7 shows the states for the reading process, (i) The system starts at the *idle* state waiting for an input stream. (ii) Then, it remains in the state *first column* until the first column of the macro-block is written to *line buffers*. (iii) In the *reading macro-blocks*

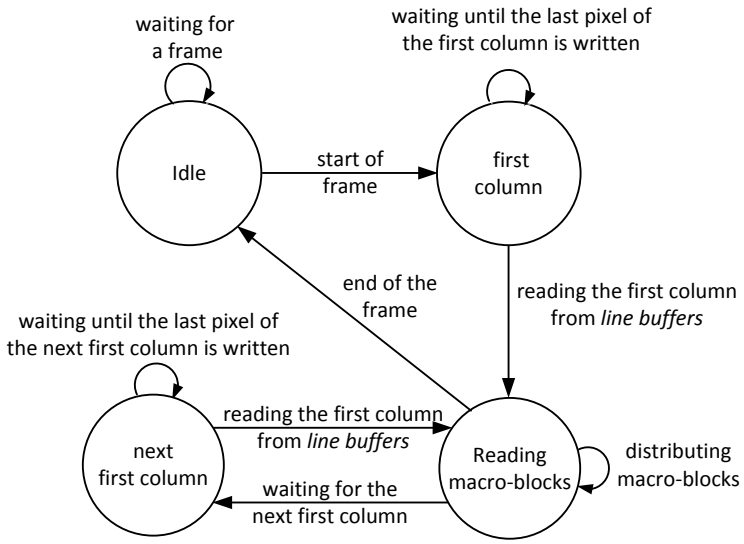

FIGURE 3.7 – Reading process finite state machine

state, the macro-blocks are sent to *circular vertical shifter* as soon as they are written to *line buffers*. (iv) After that, the system will transit to the *next first column* state waiting for the first column of the next set of macro-blocks or it will transit to the *idle* state waiting for the next input frame.

The process of writing/reading pixels to/from the *line buffers* doesn't depend either on the size of the macro-block or on the size of the input frame. Only the number of line buffers depend on the vertical size of the macro-block (V) and the number of the output ports depend on the size of the macro-block (H x V). Therefore, the VHDL files of the pixel distributor can be easily generated for different sizes of macro-block by modifying only the number of buffers in the line buffers and the number of the output ports using a script file.

3.3.4 Parallel Processing

The communication between the pixel distributor and the Processing Element (PE) is done through the *valid* signal. The pixel distributor asserts *valid* signal when a macro-block is available at its output ports (i.e. the input ports of PE). Figure 3.8 depicts the architecture for parallel processing, where a demultiplexer is used to distribute *valid* signal among the parallel processing elements. From the pixel distributor model, the rate of producing macro-blocks is equal to

$$\text{macro-block rate} \quad = \quad \frac{1}{hor\_slide} \tag{3.1}$$

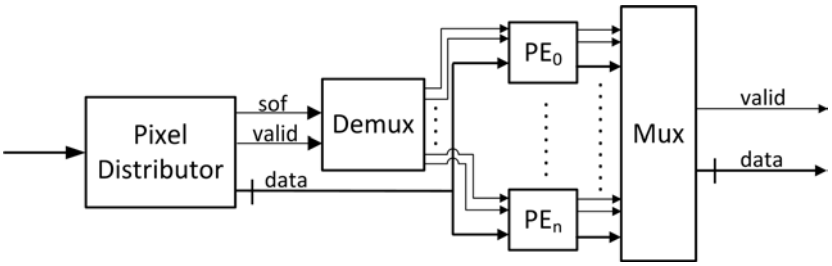

FIGURE 3.8 – Parallel structure

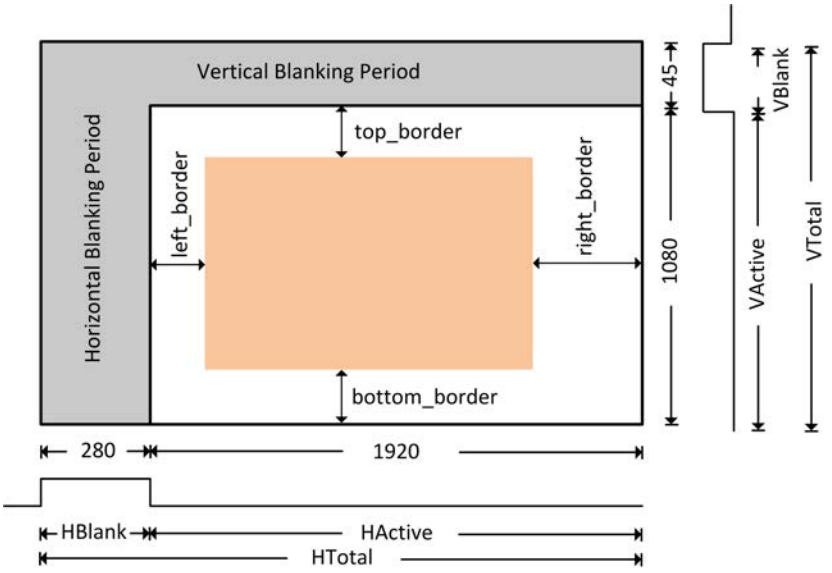

FIGURE 3.9 – Using *pixel collector* to stream 1080p frames

If the computation delay for one processing element is equal to *computation_delay* in clock cycles then the required number of parallel processing elements can be calculated using the following equation :

$$\text{Number of parallel PEs} = \lceil \textit{macro-block rate} * \textit{computation\_delay} \rceil \quad (3.2)$$

$$= \left\lceil \frac{\textit{computation\_delay}}{\textit{hor\_slide}} \right\rceil \quad (3.3)$$

3.3.5 Pixel Collector

The processed image is streamed on HD 1080p monitor ; consequently, *pixel collector* has to generate frames according to HD 1080p timing signals (i.e. HBlank = 280, HActive = 1920, HTotal = 2200, VBlank = 45, VActive = 1080 and VTotal = 1125). The processed frame could be equal to 1080p frame size as in the case of grayscale filter or smaller than it as in the case

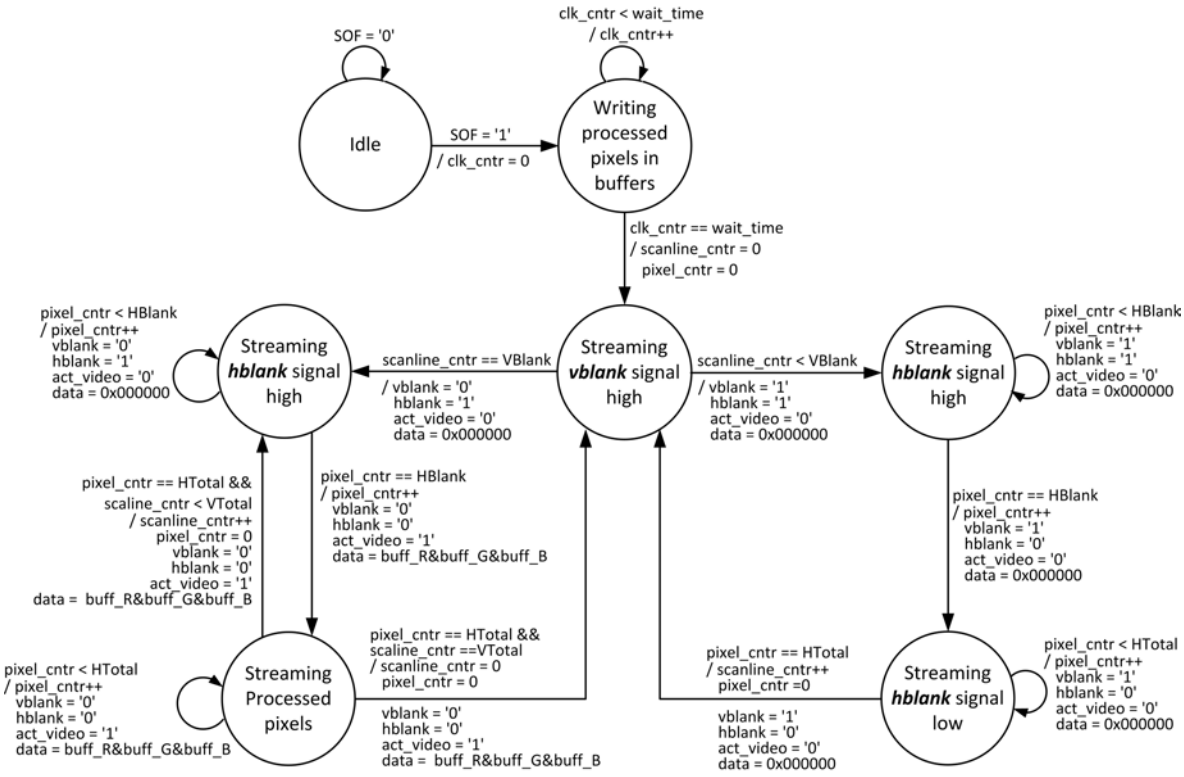

FIGURE 3.10 – Finite state machine for *pixel collector*

of video downscaler (16 :1) where the processed frame size is 480x270. Border parameters (*left_border, right_border, top_border* and *bottom_border*) are defined to pad small frames to 1080p format as depicted in Fig. 3.9 where the shaded orange area represents the actual image size. Vertical and horizontal blanking periods are streamed before the image frame for synchronization such that the total number of the streamed pixels is equal to 2200x1125 pixels after adding the blanking periods for HD 1080p frame. Fig. 3.10 shows the finite state machine which controls the process of how the pixel collector streams the processed frame in 1080p frame format. At the beginning, the pixel collector is at *idle* state waiting to receive the first processed pixel when *sof* signal is asserted high. Before starting pixel streaming, the pixel collector waits in a state till enough pixels are stored in its internal buffers to guarantee continuous pixel stream. As depicted in Fig. 3.9, VBlank signal is set to high to indicate the start of a new frame where VBlank period is counted in image lines. The system transfers between three states until VBlank period is streamed (*Streaming vblank signal high*, *Streaming hblank signal high* and *Streaming hblank signal low*). Each image line is started by HBlank period to signal the start of a new line. After that, the control is transferred to *Streaming processed pixels* state where pixels are read from the internal buffers and streamed to the output port with high *act_video* signal.

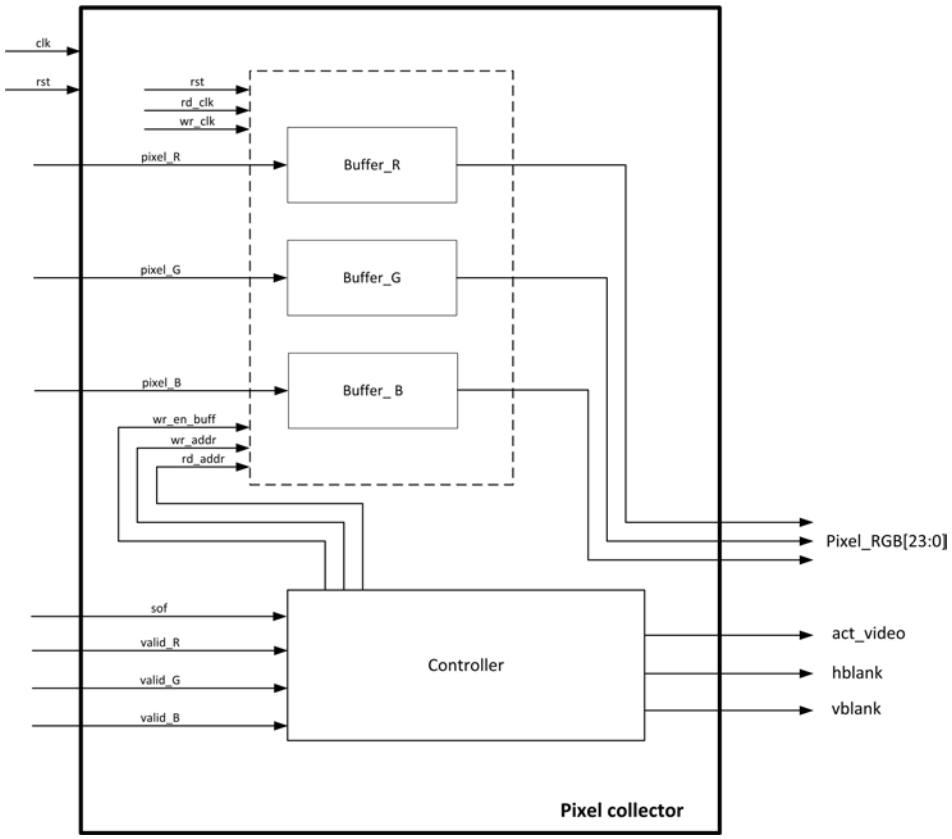

FIGURE 3.11 – The internal structure of *pixel collector*

Fig. 3.11 depicts the hardware architecture of the pixel collector where three internal buffers (*Buffer_R*, *Buffer_G*, *Buffer_B*) are used to store the three colour components of each processed pixel. The *controller* plays the role of asserting the right control signals according to the current state of the system. For example, to store the pixels in the internal buffers, the controller loads *wr_addr* by the correct address and enables *wr_en_buff* while it loads the address to *rd_addr* for reading operation. When the pixels are streamed to the output, the controller is responsible for generating the correct timing signals (*vblank*, *hblank* and *act_video*) according to its finite state machine.

3.3.6 Experimental Results

Firstly, we will highlight the code generation phase for *pixel distributor*. Secondly, we will present the synthesis results of the pixel distributor for different macro-block sizes as well as for different frame sizes. Finally, two video processing applications were implemented : video downscaler (16 :1) and convolution filter of kernel size = 3x3 to illustrate the proposed parallel structure described in section 3.3.4.

Generated files	Description	Number of code lines		
		4x4	**8x8**	**16x16**
pixel_distributor.vhd	pixel distributor top level	500	670	1300
cir_ver_shifter.vhd	circular vertical shifter	80	95	127
hor_shift_reg.vhd	horizontal shift register	83	190	600
buff.vhd	line buffers component	60	60	60
Total generated Lines of Code		723	1015	2087

TABLE 3.1 – Generated VHDL code files for the pixel distributor

3.3.6.1 Code Generation

We developed a tool that takes the length H and the width V of the macro-block as inputs to generate the required VHDL code files for the pixel distributor. By using a host machine equipped with Intel(R) Core i7 processor and 16 GB RAM, where more than 700 lines were generated automatically for a pixel distributor of macro-block size = 4x4. This is a significant improvement in comparison with manual coding for the same distribution design which could take hours for development and verification.

By using the same tool, when the size of macro-block is changed ; the designer does not need to redesign the pixel distributor but few input parameters can be modified in the tool, and after few seconds the required VHDL files are generated. The tool generates a set of files containing the description for *circular vertical shifter*, *horizontal shift register*, *line buffer* as well as the top-level module for *pixel distributor*. The tool helps the designer to obtain the required files particularly when the number of code lines increases with larger macro-block sizes. For instance, the code size grows from more than 700 lines for macro-block = 4x4 to more than 2000 lines for macro-block = 16x16 as shown in Table 3.1.

3.3.6.2 Pixel Distributor Synthesis Results

Table 3.2 shows the synthesis results for the pixel distributor over Zynq XC7Z045-FFG900 evaluation board. The pixel distributor was synthesized for the following model parameters (macro-block size = 4x4, hor_slide = 1 and ver_slide = 1) for different frame sizes (HD1080, HD720, SVGA and VGA). The results showed that the size of the controller in terms of Register and LUT differs according to the frame size. This occurs because the size of the internal counters used by the controller during reading/writing process depends on the size of the input frame. While for the other components, they are almost occupying the same area because their size depends only on the macro-block size which was fixed to 4x4 during this experiment.

Table 3.3 shows the synthesis results for the pixel distributor for fixed frame size (HD1080) with hor_slide = 1, ver_slide = 1 and different sizes of macro-block (1x3, 2x2, 3x1, 4x4,....). From the results, we can notice that *circular vertical shifter* has almost the same area when V

	1920x1080			1280x720			800x600			640x480		
	Slice Reg	Slice LUT	BRAM_18K	Slice Reg	Slice LUT	BRAM_18K	Slice Reg	Slice LUT	BRAM_18K	Slice Reg	Slice LUT	BRAM_18K
Circular vertical shifter	2	36	0	2	34	0	2	36	0	2	34	0
Horizontal shift register	130	0	0	130	0	0	130	0	0	130	0	0
Controller	95	173	0	92	174	0	87	159	0	84	163	0
Line buffers	0	0	4	0	0	4	0	0	4	0	0	4
Total	227	209	4	224	208	4	219	195	4	216	197	4
Freq(MHz)	268.53			267.95			266.17			269.40		

TABLE 3.2 – Synthesis results for pixel distributor for model parameters (macro-block = 4x4 with hor_slide = 1 and ver_slide = 1) with different frame sizes

Macro-block size (H x V)	Circular vertical shifter		Horizontal shift register		Controller		Line buffers	Total			Freq (MHz)
	Slice Reg	Slice LUT	Slice Reg	Slice LUT	Slice Reg	Slice LUT	BRAM_18K	Slice Reg	Slice LUT	BRAM_18K	
1 x 3	2	29	26	0	83	145	3	111	174	3	289.27
2 x 2	1	9	34	0	94	170	2	129	179	2	269.98
3 x 1	0	0	26	0	92	176	1	118	176	1	269.98
3 x 8	3	134	194	0	88	170	8	285	304	8	259.87
4 x 4	2	36	130	0	95	173	4	227	209	4	268.53
4 x 6	3	107	194	0	95	178	6	292	285	6	267.59
5 x 4	2	35	162	0	95	183	4	259	218	4	257.69
5 x 7	3	144	282	0	94	183	7	379	327	7	244.92
6 x 4	2	36	194	0	95	175	4	291	211	4	268.53

TABLE 3.3 – Synthesis results for pixel distributor for HD1080 frame, hor_slide = 1, ver_slide = 1 and different macro-block sizes

parameter is the same for macro-block of sizes 4x4, 5x4 and 6x4. For *horizontal shift register*, it has the same area for distributors of the same number of output pixels like in the case of 3x1 and 1x3 or in the case of 4x6, 6x4 and 3x8. Based on the synthesis results, the maximum operating frequency for *pixel distributor* is higher than the one required for HD1080 running at 60 frame/sec (i.e. 148.5 MHz).

3.3.6.3 Video Downscaler (16 :1)

Video downscaler (16 :1) scales HD1080 frame (1920x1080) to one sixteenth of its size (480x270). Figure 3.12 shows the architecture of the parallel image processing elements where

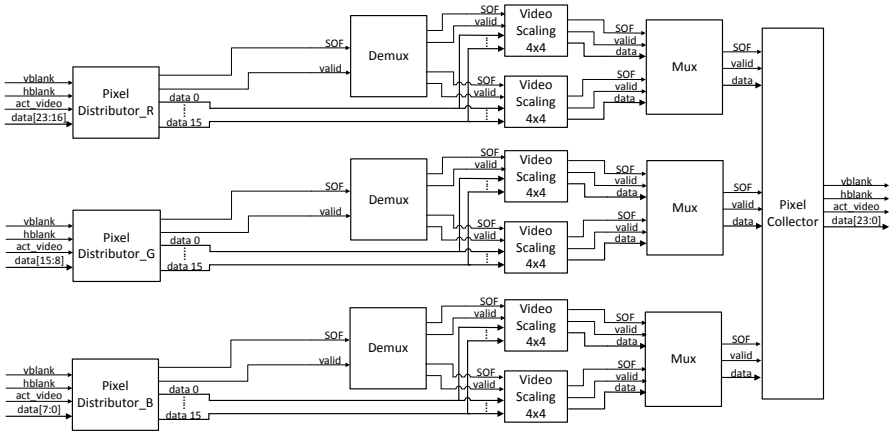

FIGURE 3.12 – Parallel architecture for video downscaler (16 :1)

video downscaler (16 :1) has a separate processing channel for each color component (red, green and blue). The pixel distributor was configured with the following model parameters (macro-block size = 4x4, hor_slide = 4 and ver_slide = 4). The computation delay for *VideoScaling*_4x4 IP is 8 clock cycles, and by applying equation 3.3, we can deduce that the required number of parallel PEs is 2. Thus, we had two *VideoScaling*_4x4 IPs working simultaneously for each processing channel. The demultiplexer is used to branch the control signals (*valid* and *sof*) over the IPs for parallel processing. While the multiplexer is used to gather both the processed pixels and the control signals from the parallel processing elements to send them to the pixel collector. In the pixel collector, pixels are stored in order, and when there are enough stored pixels in the buffer, it starts streaming the video frame with the corresponding video timing signals (vblank, hblank and active_video) to the HDMI output port. The scaled output frame can be placed in the middle of the screen by setting the border parameters of the pixel collector to ($left\_border = 720$, $right\_border = 720$, $top\_border = 405$, $bottom\_border = 405$).

Table 3.4 shows the synthesis results for video downscaler (16 :1). The video downscaler occupies 4.8% and 9.3% of the total available resources for Register and LUT respectively. The parallel processing channels consume nearly 9.7% of the total used Register and 8.6% of that used for LUT. *Pixel distributor* utilizes around 3.2% of the total design area for both Register and LUT ; thus, it represents a low hardware design cost. For BRAM utilization, video downscaler (16 :1) shows around 22% of the total available BRAM on the board since *pixel collector* keeps the pixels for one scaled frame (480x270) before starting streaming the output.

	Slice Reg	Slice LUT	BRAM_18K	BRAM36	DSP48E1
Video processing system architecture					
Video timing controller 0	1209	1230	0	0	0
VITA image sensor receiver	5941	6331	0	13	4
Image preprocessing pipeline	10565	9733	19	10	9
RGB-to-YCbCr422	254	202	0	0	4
Video timing controller 1	1093	1114	0	0	0
Total	19062	18610	19	23	17
Video downscaler (16 :1)					
Pixel distributor (R,G,B)	669	639	12	0	0
Demux (R,G,B)	3	9	0	0	0
Video scaling (R,G,B)	1140	756	0	0	0
Mux (R,G,B)	90	42	0	0	0
Pixel collector (R,G,B)	154	307	0	96	0
Total	2056	1753	12	96	0
Total application area	21118	20363	31	119	17
Resource utilization (%)	4.83	9.32	2.8	22	1.89
Convolution filter					
Pixel distributor (R,G,B)	507	630	9	0	0
Demux (R,G,B)	9	27	0	0	0
Conv. filter (R,G,B)	4410	2844	0	0	0
Mux (R,G,B)	210	99	0	0	0
Pixel collector (R,G,B)	132	325	0	72	0
Total	5268	3925	9	72	0
Total application area	24330	22535	28	95	17
Resource utilization (%)	5.56	10.31	2.57	17.61	1.89

TABLE 3.4 – Synthesis results for the video downscaler (16 :1) and convolution filter

3.3.6.4 Convolution Filter

Based on the same video processing architecture shown in Fig. 3.2, a convolution filter [38] with kernel [-1, -1, -1, -1, 9, -1, -1, -1, -1] is applied to HD1080 input frame captured by the VITA image sensor. In this application, a processing channel is dedicated to each color component where Fig. 3.13 shows the processing channel for the red color component and similarly it will be for the green and blue color channels. The input stream is distributed by *pixel distributor* in the form of macro-blocks of size = 3x3 with hor_slide = 1 and ver_slide = 1. The computation delay for *Conv*_3x3 block is 6 clock cycles and by using equation

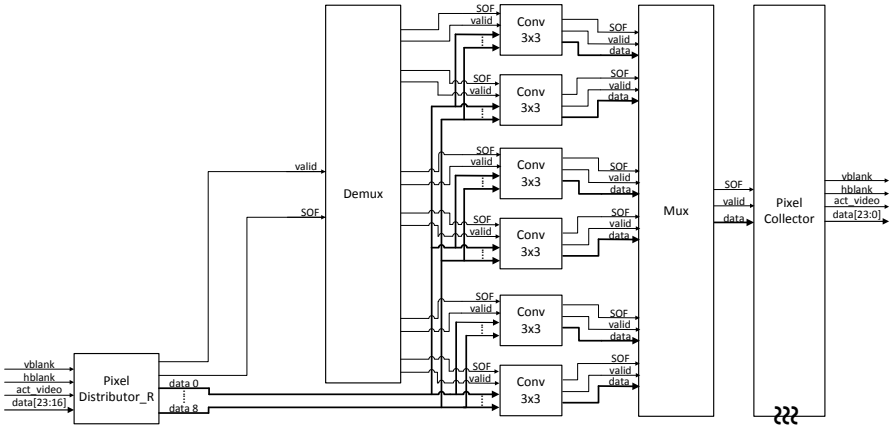

FIGURE 3.13 – The red color processing channel for convolution filter

3.3, the required number of parallel *Conv_*3x3 elements for each channel will be 6 IPs in order to process the distributed macro-blocks. Demultiplexer and multiplexer are used for branching and gathering pixels and control signals through the parallel architecture. Due to the limitation described in subsection 3.3.2, the border pixels are not processed so the pixel collector produces the output frame with a contour of black pixels. The border parameters of the pixel collector were set to the following values ($left\_border = 1$, $right\_border = 1$, $top\_border = 1$, $bottom\_border = 1$).

As shown in Table 3.4, the convolution filter has 5.5% of the total available Register and 10.3% of that available for LUT. The parallel processing channels occupy 21.7% and 17.4% of the total design utilization for Register and LUT respectively. This percentage rises due to the presence of 6 parallel *Conv_*3x3 elements working at the same time for each processing color channel. The pixel distributor shows less than 3% of both resources which proves the low hardware cost of our solution. For BRAM utilization, the collector starts streaming at the time it receives the first processed pixel ; however, the frame is started by a VBlank period so the processed pixels have to be stored in buffers during that period. For this reason, the convolution filter takes around 17.6% of the total available BRAM resources.

3.4 Using Hardware Parallelism for Reducing Power Consumption

Fig. 3.14 depicts three processing channels (red, green and blue) where the pixel distributor distributes the input pixel stream in the form of macro-blocks of size $H \times V$, where H is the horizontal size and V is the vertical one. It stores the pixels in its internal buffers during the first (V-1) rows of the macro-block (i.e. *idle time*) while during the last row, it starts to

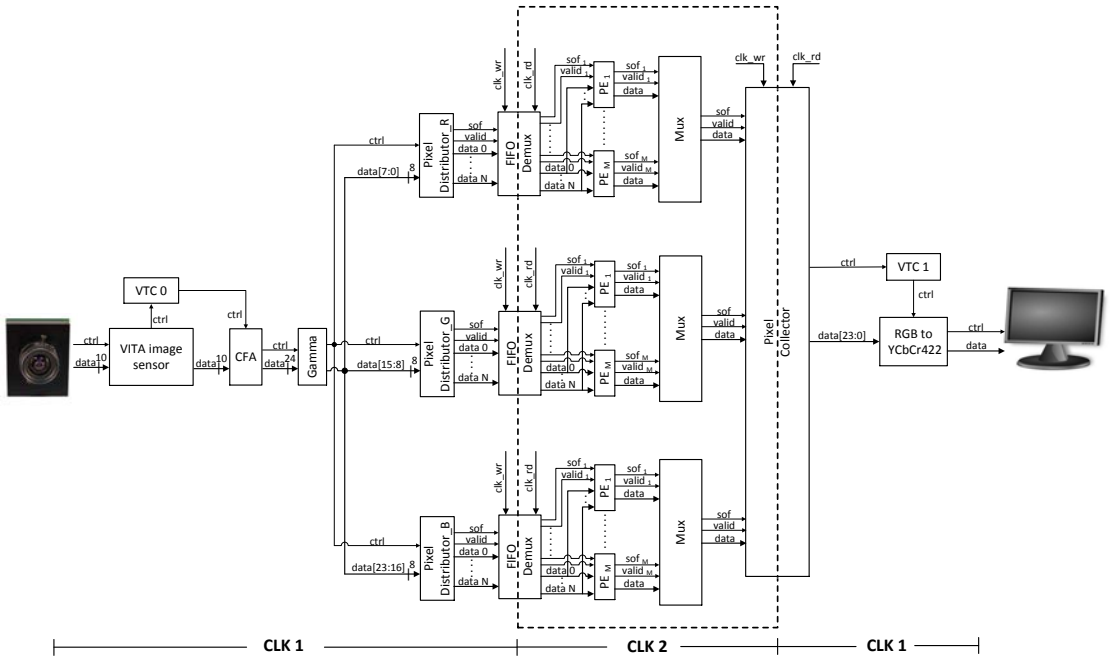

FIGURE 3.14 – Video processing architecture

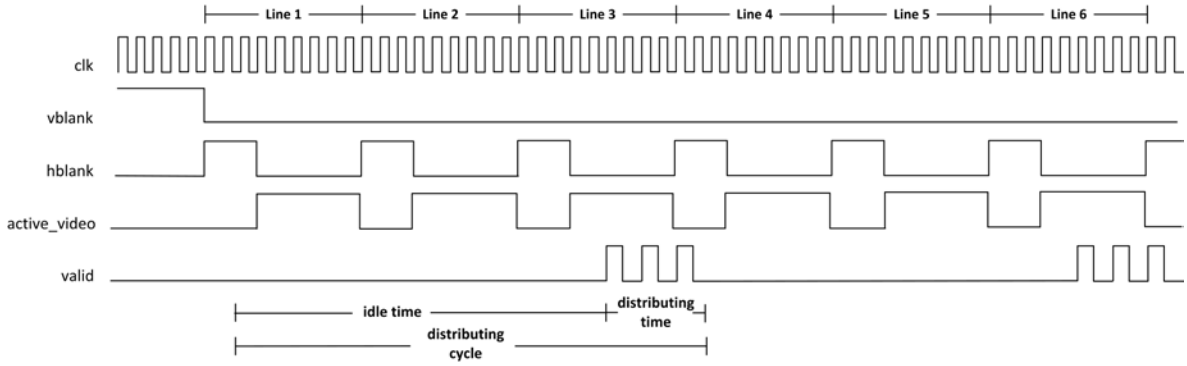

FIGURE 3.15 – *valid* signal during the *distributing_cycle* for macro-blocks of $H = 2$ and $V = 3$

distribute the pixels in the form of macro-blocks with *valid* signal assigned high with each block (i.e. *distributing_time*) as shown in Fig. 3.15.

The parallel Processing Elements (PEs) are operating at clock frequency FREQ2 which is slower than the one used by the other parts of the system (FREQ1). Therefore, a FIFO is required to store the macro-blocks during their transfer from one clock domain to another. FIFO is typically implemented using a dual-port BRAM where we have two input clock frequencies : *clk_wr* for writing and *clk_rd* for reading. The block named *FIFO_DeMux* has two roles : (i) to store the macro-blocks when they are transferred from clock domain CLK1 to clock domain CLK2. (ii) to distribute the macro-blocks among the processing elements (PE_1, PE_2, PE_3 ,....., PE_n). Multiplexers are used to gather the processed data from the parallel PEs ; then they are later written to the pixel collector. When the pixel collector has enough pixels, it starts streaming them to *RGB-to-YCbCr*422 block where pixels are converted to YCbCr 4 :2 :2 format ready for streaming on HD monitor. The communication between the blocks is done through the signals named *valid* and *sof* such that *valid* signal is asserted high when there is an available pixel at the output port, while *sof* signal is flagged only if this pixel represents the start of the frame.

3.4.1 Level of Parallelism and FIFO Depth Calculations

3.4.1.1 Level of Parallelism

If *pixel distributor* sends pixels to the FIFO at a rate faster than the receiving side can handle, then the depth of FIFO will grow indefinitely. As shown in Fig. 3.15, to bound the maximum depth of FIFO, the produced macro-blocks during the *distributing_time* should be processed within the time of the *distributing_cycle* otherwise the maximum depth of FIFO will grow up.

Taking this constraint into consideration, we can calculate the maximum computation delay (*max_comp_delay*) available for each processing element as following :

$$max\_comp\_delay = \frac{distributing\_cycle * N\_PE}{N\_mblocks * rd\_clk} \tag{3.4}$$

$$= \frac{V * line\_period * wr\_clk * N\_PE}{N\_mblocks * rd\_clk} \tag{3.5}$$

$$= \frac{V * line\_period * \frac{1}{FREQ1} * N\_PE}{N\_mblocks * \frac{1}{FREQ2}} \tag{3.6}$$

$$= \frac{V * line\_period * \mathrm{FREQ2} * N\_PE}{N\_mblocks * \mathrm{FREQ1}} \tag{3.7}$$

Where V is the vertical dimension of the macro-block, *line_period* is the time required to stream one line of pixels in the horizontal direction, *distributing_cycle* is the time required to stream V lines of pixels, *N_PE* is the number of parallel processing elements, *N_mblocks* is the number of macro-blocks per *distributing_cycle*, *wr_clk* is the clock period for FIFO

writing clock frequency (FREQ1) and *rd_clk* is the clock period for FIFO reading clock frequency (FREQ2).

From the same equation, by fixing the computation delay (*comp_delay*), we can calculate the required level of parallelism (i.e. *N_PE*) to be :

$$\text{Level of Parallelism} = \frac{comp\_delay * N\_mblocks * rd\_clk}{distributing\_cycle} \tag{3.8}$$

$$= \frac{comp\_delay * N\_mblocks * rd\_clk}{V * line\_period * wr\_clk} \tag{3.9}$$

$$= \frac{comp\_delay * N\_mblocks * \frac{1}{\text{FREQ2}}}{V * line\_period * \frac{1}{\text{FREQ1}}} \tag{3.10}$$

$$= \frac{comp\_delay * N\_mblocks * \text{FREQ1}}{V * line\_period * \text{FREQ2}} \tag{3.11}$$

3.4.1.2 FIFO Depth

Since we can not simultaneously read and write to the same FIFO position ; therefore, a constant value equal to 2 will be added to guarantee a minimum non-zero FIFO depth. At every *rd_clk* (CLK2), one PE can be activated ; so for calculating the maximum FIFO depth, we will have two cases according to how much slower is *rd_clk* (CLK2) than *wr_clk* (CLK1) :

- **Case 1**. Fig. 3.16 shows the case when not yet all PEs are activated by the end of the *distributing_time* (i.e. *N_PE* * *rd_clk* > *distributing_time*). In this case, FIFO depth is calculated as follows :

$$\text{FIFO depth} = N\_mblocks - N\_act\_PE + 2 \tag{3.12}$$

$$= N\_mblocks - \frac{distributing\_time}{rd\_clk} + 2 \tag{3.13}$$

$$= N\_mblocks - \frac{N\_pixels\_line * \frac{1}{FREQ1}}{\frac{1}{FREQ2}} + 2 \tag{3.14}$$

$$= N\_mblocks - \frac{N\_pixels\_line * \text{FREQ2}}{\text{FREQ1}} + 2 \tag{3.15}$$

  Where *N_mblocks* is the number of macro-blocks per *distributing_cycle*, *N_act_PE* is the number of active processing elements by the end of the *distributing_time*, and *N_pixels_line* is the number of pixels per *line_period*.

- **Case 2**. Fig. 3.17 shows the other case when all PEs are activated at least once during the *distributing_time* (i.e. *N_PE* * *rd_clk* ≤ *distributing_time*)

$$\text{FIFO depth} = N\_mblocks - \frac{distributing\_time * N\_PE}{rd\_clk * comp\_delay} + 2 \tag{3.16}$$

$$= N\_mblocks - \frac{N\_pixels\_line * \frac{1}{FREQ1} * N\_PE}{\frac{1}{FREQ2} * comp\_delay} + 2 \tag{3.17}$$

$$= N\_mblocks - \frac{N\_pixels\_line * \text{FREQ2} * N\_PE}{\text{FREQ1} * comp\_delay} + 2 \tag{3.18}$$

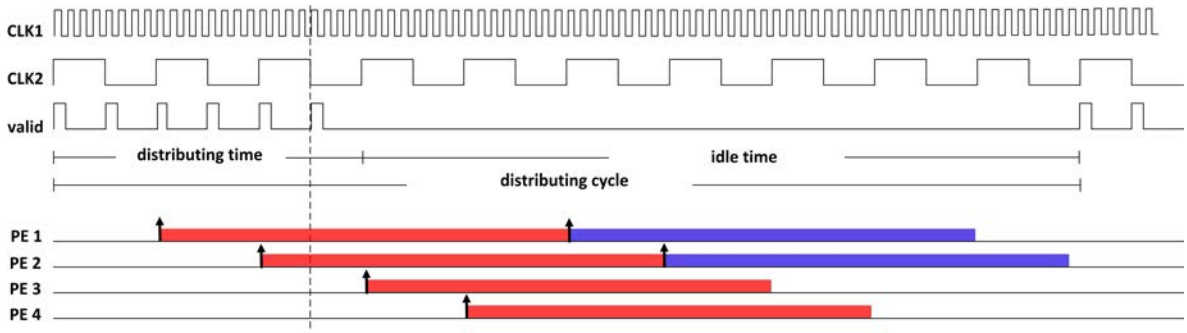

FIGURE 3.16 – Case 1 : when all PEs are not yet activated where *N_mblocks* = 6 blocks, *N_pixels_line* = 20 pixels, FREQ1 = 8F and FREQ2 = F.

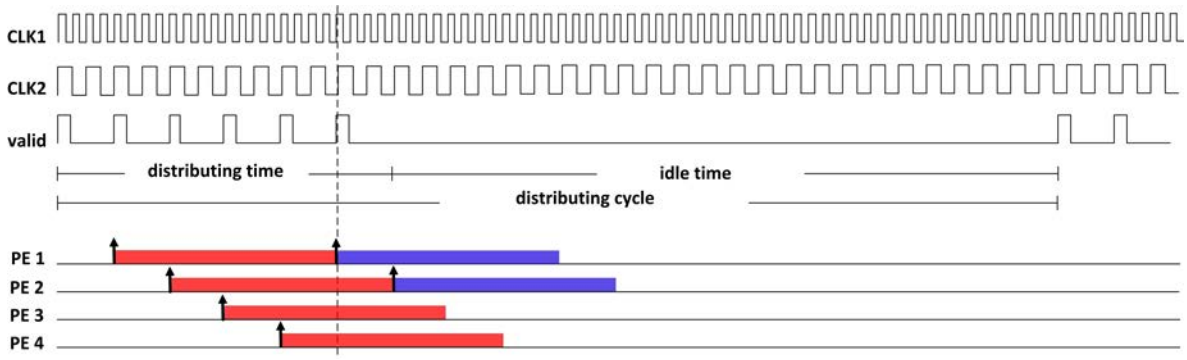

FIGURE 3.17 – Case 2 : when all PEs are activated at least once where *N_mblocks* = 6 blocks, *N_pixels_line* = 20 pixels, *N_PE* = 4, *comp_delay* = 8 cycles, FREQ1 = 2F and FREQ2 = F

.

where *comp_delay* is the number of clock cycles required by PE to process one macro-block.

3.4.2 Experimental Results

In this section, we will discuss the implementation of two different video processing applications : video downscaler (16 :1) and AES encryption algorithm. By applying the equations obtained in the previous section, we were able to obtain different design alternatives varying in the depth of FIFO and in the level of parallelism. For each design alternative, the power was estimated by Xilinx XPower Analyzer and measured practically using TI Fusion Digital Power Designer. The preferable design is then selected based on the percentage decrease in power compared to the hardware cost needed to implement this solution.

Design point	Level of parallelism	FREQ1 (MHz)	FREQ2 (MHz)	FIFO depth
video downscaler (16 :1) Application				
D1	1	148.5	148.5	0
D2	1	148.5	74.25	242
D3	1	148.5	37.125	362
D4	2	148.5	74.25	2
D5	2	148.5	37.125	242
D6	2	148.5	18.5625	362
D7	4	148.5	37.125	2
D8	4	148.5	18.5625	242
D9	4	148.5	9.28125	362
AES Encryption Application				
D1	3	148.5	148.5	0
D2	3	148.5	74.25	242
D3	3	148.5	37.125	362
D4	6	148.5	74.25	2
D5	6	148.5	37.125	242
D6	6	148.5	18.5625	362
D7	12	148.5	37.125	2
D8	12	148.5	18.5625	242
D9	12	148.5	9.28125	362

TABLE 3.5 – The design points for video downscaler (16 :1) and AES encryption applications

3.4.2.1 Design Points

The application was synthesized using the parallel video processing architecture depicted in Fig 3.14 over Zynq XC7Z045-FFG900 platform. The image sensor was configured for 60 frame/sec such that FREQ1=148.5 MHz while FREQ2 was a divisor of FREQ1 according to the selected design point. For video downscaler (16 :1), the pixel distributor distributed the HD frames in the form of macro-blocks of size = 4x4 to the processing elements of computation delay equal to 4 clock cycles. For AES encryption application, the HD frame was encrypted through a non-pipelined 128-bit AES encryption IP of computation delay equal to 12 clock cycles. We chose Electronic Codebook cipher mode (ECB) since it is the simplest AES encryption mode where the plaintext is separately encrypted using the same 128-bit cipher key [30].

Table 3.5 listed a set of different design points. These points could be obtained using equation 3.11 by either varying the level of parallelism or the operating frequency FREQ2. For both applications, the design point D1 is considered as the reference design point because it has the minimum required level of parallelism as well as it operates at the same clock frequency (i.e. FREQ1 = FREQ2 = 148.5 MHz).

Design point	Slice	Register	LUT	LUTRAM	BRAM_18K	BRAM_36K	DSP48E1
Video Downscaler (16 :1) Application							
Base	8860	17273	17046	1168	29	114	16
D1	183	573	342	0	0	0	0
D2	1125	3537	2645	0	0	6	0
D3	1799	4989	4154	0	0	6	0
D4	613	1665	1034	264	0	0	0
D5	1042	3471	2830	0	0	6	0
D6	1508	4557	3767	0	0	6	0
D7	1004	2889	1797	264	0	0	0
D8	1612	5406	4026	0	0	6	0
D9	2165	6849	5005	0	0	6	0
AES Encryption Application							
Base	8873	17376	17027	1168	77	18	16
D1	5660	7518	14645	0	0	0	0
D2	6378	10482	17113	0	0	6	0
D3	7157	11934	18435	0	0	6	0
D4	11564	16635	30147	264	0	0	0
D5	12643	19164	32025	0	0	6	0
D6	12998	20610	33149	0	0	6	0
D7	21881	32451	59936	264	0	0	0
D8	22539	34986	62033	0	0	6	0
D9	23534	36429	62929	0	0	6	0

TABLE 3.6 – The Synthesis results for each design point for both video downscaler (16 :1) and AES encryption

3.4.2.2 Synthesis Results

The selected synthesis/implementation strategy can affect the power consumption of the implemented design [69]. Taking this into consideration, it is worth to mention our selected options for synthesis and implementation during our experiments. PlanAhead 14.3 tool was used during the design process where for both applications, *PlanAhead Defaults* was used as a synthesis strategy. While the implementation strategy was as follows : (i) For video downscaler, we used *ISE Defaults* for all designs except for designs D8 and D9, it was *ParHighEffort* to meet the timing constraints. (ii) For AES encryption, we used by default *ParHighEffort* strategy while *MapTiming* was used for design D2 to avoid timing constraints violation.

Table 3.6 shows the hardware cost for each design point. For each application, the row named *base* represents the required resources for implementing the basic blocks which exist in

Design point	Video Downscaler (16 :1) Application		AES Encryption Application	
	Measured Power (in mW)	Percentage power decrease (%)	Measured Power (in mW)	Percentage power decrease (%)
D1	1288.95	0	1038.36	0
D2	1212.94	5.9	1046.87	-0.82
D3	1116.93	13.35	1020.26	1.74
D4	1126.36	12.61	1023.54	1.43
D5	1111.96	13.73	1005.16	3.2
D6	1067.65	17.17	991.15	4.55
D7	1059.1	17.83	989.26	4.73
D8	1055.32	18.13	992.04	4.46
D9	1036.56	19.58	982.36	5.39

TABLE 3.7 – The measured power for different design points for video downscaler and AES encryption

every single design point like VITA image sensor, VTC, CFA, GAMMA, pixel distributors or pixel collector. While the row named after each design point represents the needed resources for implementing that specified design. Therefore, the total resources used for realizing a single design point is equal to the sum of the *base* row in addition to the row representing that design point. For example, the total design cost for D1 for video downscaler (16 :1) is : Slice = 9043, Register = 17846 and LUT = 17388.

From the synthesis results, we can get some observations that will later help us to understand how the power is consumed in the system : (i) It is obvious that the used BRAMs for video downscaler application was more than that used for AES application. This occurred because video downscaler (16 :1) needs to store more pixels before start streaming the video frames. (ii) The required level of parallelism for AES application is higher than that needed for video downscaler as mentioned in Table 3.5. Consequently, the total used logic for AES application will be greater than that used for video downscaler (16 :1).

3.4.2.3 Power Analysis

The power consumption for each design point was estimated using XPower Analyzer [110] to understand how the power was broken down between the different hardware resources. The power was also measured for verification through the power controller UCD90120A mounted on the evaluation board using Fusion Digital Power Designer [98]. During our experiments, we considered the Register number as the cost function to implement a certain design choice. For sure, we can choose any other hardware resource as the cost, or we can even have multiple factors in the cost function (for example, the summation of both register and LUT number as the cost function).

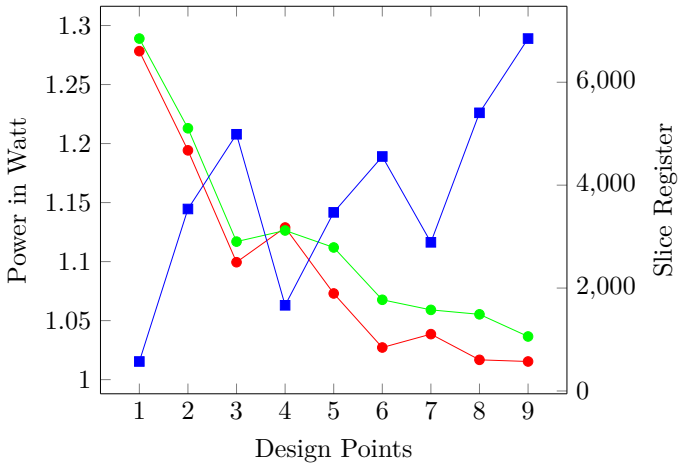

FIGURE 3.18 – The trade-off between the estimated power —●—, the measured power —●— and the slice register cost —■— for each design point for video downscaler (16 :1)

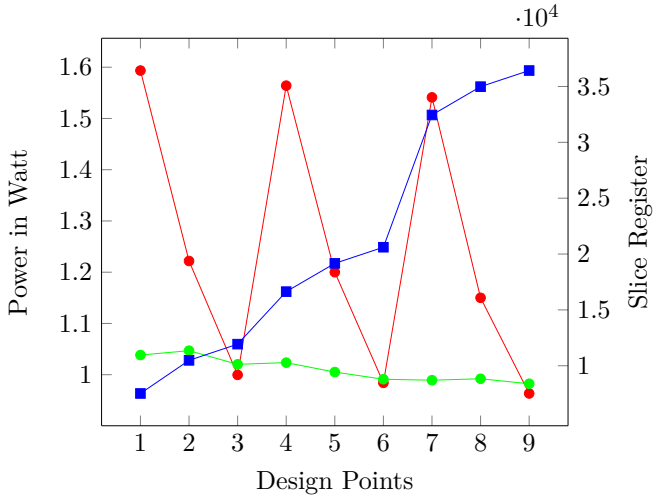

FIGURE 3.19 – The trade off between the estimated power —●—, the measured power —●— and the slice register cost —■— for each design point for AES encryption

In Fig. 3.18, the estimated and measured power for video downscaler application was plotted against the number of Register required for each design point. Experimentally, the power consumption decreased from 1.29 W for D1 to be 1.04 W at D9 with a percentage power reduction equal to 19.6% as listed in Table 3.7. According to the available register resources, the designer can select which design alternative to use and what percentage decrease in power to gain as listed in Table 3.6 and Table 3.7. For example in video downscaler (16 :1), the percentage decrease in power consumption for D7 was 17.8% at register cost = 2889 and for D6 was 17.1% at register cost = 4557 so D7 is always better than D6 since it achieved more power reduction at lower register cost. Also, we can consider D7 as a design choice better than other points like D8 or D9 because the percentage decrease in power between these points

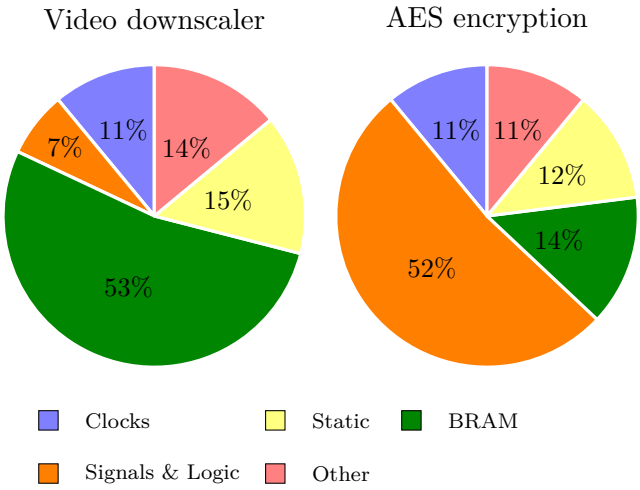

FIGURE 3.20 – The power consumed by different resources to implement the reference design D1 for both video downscaler and AES encryption

and D7 is not so significant (0.3% for D8 and 1.7% for D9) if compared to the percentage increase in the register cost (87% for D8 and 137% for D9).

For AES encryption application, Fig. 3.19 depicts the estimated and measured power versus the Register cost for different design points. From the experimental measurements, the percentage decrease in power compared to that for the reference design was in the range of -0.8% up to 5.4% as reported in Table 3.7. One reason for having such power increase at D2 is because that the used implementation strategy was changed to satisfy the timing constraints. It relies on the designer decision either to profit from the maximum possible power reduction of 5.4% at register cost = 36429 or to stay at some moderate hardware cost like at D6 with register cost = 20610 and power reduction of 4.5%.

Fig. 3.20 depicts the power estimations for the reference design D1 for both applications. When we look deep into how the power consumption is broken down between the different hardware resources. We can easily deduce that the significant power fraction was consumed by the BRAM in the case of video downscaler while it was from the Signals & Logic in the AES application. This can help us to explain why the maximum possible power reduction was large for video downscaler (19.6%) while it was small for AES encryprtion (5.4%) : (i) For video downscaler, the significant portion of the used BRAM were counted from the base design resources as well as the significant fraction of the power was consumed by them. Therefore ; when FREQ2 was scaled over the BRAMs, the total power consumption was decreased as well. Table 3.5 showed that scaling down FREQ2 was accompanied by an increase in the level of parallelism as well as the depth of FIFOs and consequently the used hardware resources increased. But fortunately, the achieved power reduction was not too much affected by the power consumption arising from the added logic and thus we obtained a percentage decrease reached up to 19.6%. (ii) For the AES encryption application, the number of the used BRAM

was not too much compared to the used logic, so the significant portion of the consumed power was due to the used logic. Accordingly, as the level of parallelism increased, the used logic increased as well. Unfortunately, scaling FREQ2 in this case was not enough to compensate the increase in the power consumption due to the added logic and to show in return a significant decrease in the total power consumption. Therefore, although D1, D4 and D7 operate at different clock frequencies equal to 148.5 MHz, 74.25 MHz and 37.125 MHz respectively, they reported a small percentage decrease in power reduction because of the added logic resulting from increasing the level of parallelism.

It is notable that the percentage error between the estimated and measured power was small for the video downscaler while it was significant for the AES encryption. This behaviour from XPower Analyzer can be explained in the highlight of Fig. 3.20. For video downscaler application, the power consumption was dominated by the BRAM while the Signals & Logic dominated it for AES application. If we suppose that XPower Analyzer can assume better activity rates for BRAMs than that assumed for Logic ; therefore, the power estimations for video downscaler will be more close to the real measurements than that in the case of AES application.

3.4.2.4 Performance

To satisfy the timing condition of 60 frame/sec, the output video channel was constrained to clock frequency FREQ1 = 148.5 MHz. We also limited the maximum depth of the FIFOs by processing the produced macro-blocks within their distributing cycle as mentioned before in section 3.4.1.2. According to these constraints, not every pair (level of parallelism, scaled frequency FREQ2) could suit as a design point for our application. As a result form that, regardless what level of parallelism is applied or what value for FREQ2 is chosen, the performance is kept constant at 60 frame/sec for all design points.

3.5 Conclusion

In this chapter, we presented a generic pixel distribution/gathering model dedicated for streaming video applications with low hardware cost. The pixel distributor has a flexible model where the required VHDL files can be obtained by setting the size of the macro-block and the sliding manner in the code generation tool without spending more redesign efforts. After that, we presented how the parallel hardware architecture is modified in conjunction with frequency scaling to reduce power consumption. The equations required to calculate the level of parallelism and the depth of the FIFOs were derived. Then, with the help of these equations, a design space including all the possible design alternatives was obtained. The designer is free to choose whichever design alternative to use based on the tradeoff between the hardware cost and the defined goal for power consumption. Two video processing applications were

implemented to verify our approach where the results for the measured power showed up to 19.6% power reduction for video downscaler and up to 5.4% for AES application.

Chapter 4

Efficient Hardware Implementation for Multi-Window SAD Algorithm

4.1 Introduction

Using High-Level Synthesis (HLS) tools in Electronic Design Automation (EDA) aims at moving the design efforts to higher abstraction levels. Among the reasons that have motivated researchers to continue improving HLS tools : the huge growth in the silicon capacity, the emergence of IP-based design approaches, trends towards using hardware accelerators on heterogeneous SoCs, the time-to-market constraint which usually presses to reduce the design time. Translating C/C++ code to RTL design with the help of HLS tools does not mean an efficient hardware design. However, a set of high-level optimization steps could be applied in order to obtain an efficient design. In this chapter, we will present the flow of steps to modify the high-level code targeting hardware platforms as well as to exploit the inherent parallelism in the application. We will show the impact of applying each optimization step on the overall design efficiency in terms of hardware utilization and performance. It is worth to mention that these optimization steps could be advised not only for stereo matching algorithms but also for any other video processing application. In the context of our collaboration with NAVYA company, Multi-window Sum of Absolute Difference (Multi-window SAD) stereo matching algorithm was chosen as our industrial case study. In the experimental results, we will show its hardware implementation for input grey images of size 640x480.

4.2 Stereo Matching Algorithm

Stereo matching is the problem of finding the depth of objects using two or more images. These images are taken from different positions by different cameras at the same time. Stereo matching is a correspondence problem where for every pixel (X_R) in the right image, we try to find its best matching pixel (X_L) in the left image at the same scanline. Figure 4.1a shows how the depth of objects is calculated in the stereo matching problem. Assuming two cameras of focal length (f) at the same horizontal level, separated from each other by a distance *baseline* (b). Pixel (P) in the space will be located at point (X_R) and point (X_L) in the right and left image respectively. The difference between the two points on the image plane is defined as *disparity* (d) as depicted in Fig. 4.1b. Therefore ; the depth of pixel (P) from the two cameras can be calculated from the following equation :

$$\text{depth} = \frac{baseline * focal\ length}{disparity} = \frac{b * f}{(X_R - X_L)} \tag{4.1}$$

Several algorithms were proposed in the literature to find the best matching [84]. In this thesis, Multi-Window Sum of Absolute Difference algorithm (Multi-Window SAD) was implemented [43]. In Multi-Window SAD, absolute difference between pixels of the right and left images are aggregated within a window such that the window of minimum aggregation is considered as the best matching among its candidates. In order to overcome the error that

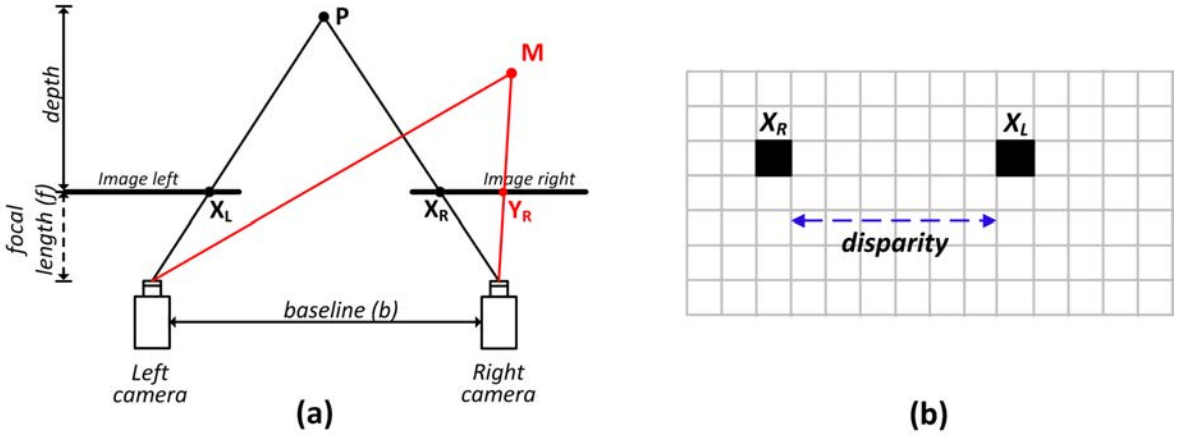

FIGURE 4.1 – (a) Calculating the depth of an object in stereo matching problem (b) Disparity is defined as the distance in pixels between (X_R) and (X_L)

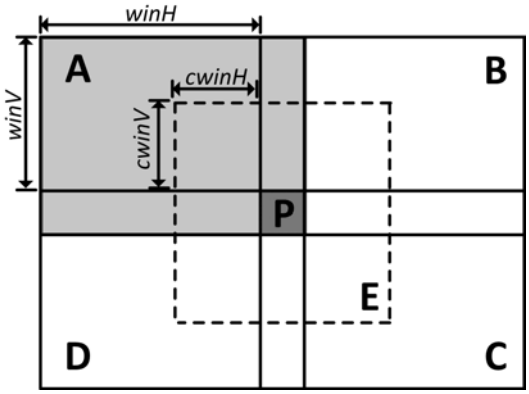

FIGURE 4.2 – 5-window SAD configuration

appears at the regions of depth discontinuity, the correlation window can be divided into smaller windows, and only non-error parts are considered in calculations. Figure 4.2 shows 5-window SAD configuration : pixel (P) lies in the middle of window (E) where it is surrounded by another four windows named (A, B, C and D). The four windows are partially overlapped at the border pixel (P). These windows are of height = winV+1 and width = winH+1 while window (E) is of height = 2*cwinV+1 and width = 2*cwinH+1. We defined *window score* as the aggregation of the absolute difference of the pixels within that window. In 5-window SAD, the correlation score at pixel (P) is equal to the score of window (E) in addition to the minimum two scores of the other four windows (A, B, C and D). The score is calculated at different disparities where the disparity of the minimum score is considered as the best matching among the candidates as depicted in Fig. 4.3. Occluded objects are common to happen in stereo matching problem where sometimes the objects are only captured by one

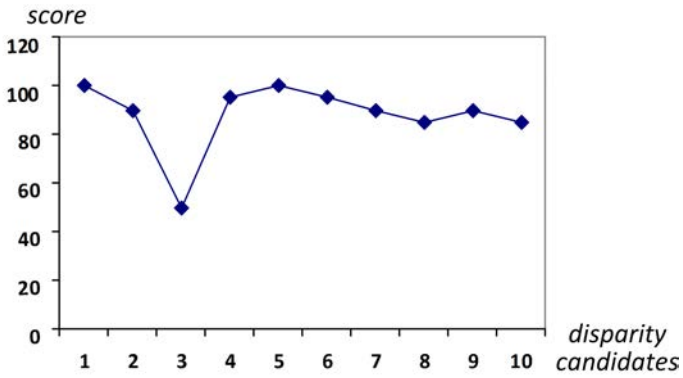

FIGURE 4.3 – Minimum score at disparity distance = 3

camera. For example, pixel (M) in Fig.4.1a was only captured by the right camera. Therefore, Left/Right consistency check is done in order to get rid of occluded objects from the final disparity image. During our experiments, we will use Vivado 2015.2 design suite to implement our application over Zynq ZC706 FPGA evaluation board (XC7Z045-FFG900) with input grey images of size 640x480. The system was configured for 5-window SAD with the following parameters : winH = 23, winV = 7, cwinH = 7, cwinV = 3 and maximum disparity = 64.

The multi-window SAD algorithm is described in Listing 4.1 where it scans all the image pixels at every disparity value ranges from zero to the maximum value (DISP_MAX). The calculation of window (E) is usually performed independently from the others because the size of window (E) is different from other windows (A, B, C and D). The for-loop described between Lines 2-23 is repeated a number of times equal to the maximum disparity value (DISP_MAX). The addition operation is considered as the core computation operation since we sum up the absolute difference of pixels. If we assume an image of size N×N with aggregation window of size M×M then $(M^2\text{-}1)N^2$ addition operations are needed for window aggregation at every single pixel. While by applying box-filtering [67] in both horizontal and vertical direction ; the number of additions will be reduced to $4N^2$. Box-filtering is applied in both horizontal and vertical directions (Lines 5-10) to obtain the score of windows (A, B, C, D and E) at each pixel. The minimum score *min_score* is equal to the sum of the score at window (E) in addition to the best minimum two score values of the other four windows (Lines 12-17). The new score value is compared to the previously calculated value at the same pixel such that if it is smaller, then both the score array *bestscoreR*[ ] and the disparity map array *DISP_IMG_R*[ ] are updated ; otherwise, they are kept unchanged (Lines 18-20). Occluded objects are common to happen in the stereo matching problem ; therefore, Left/Right consistency check is applied to get rid of occluded pixels from the final disparity map. Left/Right consistency check needs to calculate the disparity map twice ; in the first calculation, the right image is considered as

```
1  Algorithm: Mutli_Window_SAD
2     for every disparity(d) where d = 0 -> DISP_MAX:
3        for every element(x,y) where x = 0 -> imgW, y = 0 -> imgH:
4           Abs[y][x] = abs(IMG_R[y][x]-IMG_L[y][x+d]);
5        for every element(x,y) where x = winH -> imgW-winH, y = 0 -> imgH:
6            row[y][x] =  row[y][x-1] + Abs[y][x] - Abs[y][x-winH];
7           rowE[y][x] = rowE[y][x-1] + Abs[y][x] - Abs[y][x-2*cwinH];
8        for every element(x,y) where x = 0 -> imgW, y = winV -> imgH-winV:
9            scr[y][x] =  scr[y-1][x] +  row[y][x] -  row[y-winV][x];
10          scrE[y][x] = scrE[y-1][x] + rowE[y][x] - rowE[y-2*cwinV][x];
11       for every element(x,y) where x = winH -> imgW-winH, y = winV -> imgH-winV:
12          scoreA =  scr[y           ][x           ];
13          scoreB =  scr[y           ][x+winH      ];
14          scoreC =  scr[y+winV      ][x+winH      ];
15          scoreD =  scr[y+winV      ][x           ];
16          scoreE = scrE[y+winV-cwinV][x+winH-cwinH];
17          min_score = scoreE + MIN_2_values{scoreA,scoreB,scoreC,scoreD};
18          if min_score < bestscoreR[y+winV][x+winH]:
19             bestscoreR[y+winV][x+winH] = min_score;
20             DISP_IMG_R[y+winV][x+winH] = d;
21          if min_score < bestscoreL[y+winV][x+winH+d]:
22             bestscoreL[y+winV][x+winH+d] = min_score;
23             DISP_IMG_L[y+winV][x+winH+d] = d;
24    for every element(x,y) where x = 0 -> imgW, y = 0 -> imgH:
25       dispVal = DISP_IMG_R[y][x];
26       if abs(DISP_IMG_L[y][x+dispVal]-dispVal) > 1:
27          DISP_IMG_R[y][x] = 0;
```

Listing 4.1 – Pseudo code for Multi-Window SAD algorithm

the reference while vice versa happens in the second one (Lines 18-23). Hence the disparity maps are stored in *DISP_IMG_R*[ ] and *DISP_IMG_L*[ ] then for each pixel, we check if the value of the left disparity map is the same as its matching pixel in the right disparity map. If it is the case then we validate the pixel matching ; otherwise, the disparity value at that pixel is uncertain and is replaced by zero (Lines 24-27).

4.3 High-level Synthesis Optimizations

Two classes of optimizations could be applied to modify the software code.

- **Optimizations targeting hardware implementation**. This type of optimizations modifies the software code to fit for hardware implementation. For example, using arbitrary data precision instead of the standard data types for efficient hardware size or modifying the way of processing to be in the form of image strips to reduce the size of used arrays or adding AXI-Stream control signals to the output ports for communication control.
- **Optimizations for exploiting the inherent parallelism in the application.** This type of optimizations modifies the software code to exploit the parallelism in the

Design	Slice	FF	LUT	BRAM_18K	Freq. (MHz)	exec. time (ms)
SW version	380 ms on core i7@ 2.7 GHz and 16 GB of RAM					
#1	X	2637	5918	7392	100	X
#2	898	1743	2735	155	100	30080
#3	859	1758	2659	113	100	22410
#4	1400	2552	3738	75	100	8163
#5	983	1525	2567	47	100	5786
#6	996	1575	2619	49	100	6307

TABLE 4.1 – Synthesis results reported by Vivado HLS for each optimization step

application at different levels (pipeline-level, task-level or data-level parallelism).

The software code can be modified in two ways : (i) Code restructuring : some optimizations are achieved by rewriting some parts of the SW code. For example, rewriting some loops in separate sub-functions to enforce their execution in parallel or defining variables in arbitrary precision data types to decrease area utilization. (ii) Directives offered by the High-level Synthesis tool like loop pipelining, loop unrolling or array partitioning. These directives are one of HLS tools strength points where they are easily added to the software code to perform sophisticated optimizations without the need to modify the code by hand. The order of applying the optimization steps in the design flow should be taken into consideration in order to avoid reapplying some optimization steps twice. For example, the optimizations targeting hardware implementation should be applied prior to that for exploiting the parallelism.

4.3.1 Optimizations Targeting Hardware Implementation

The behavioural code was written in HLS-friendly syntax with neither file read/write, nor dynamic memory allocation nor system calls. The SW code was executed on standard PC to test its correct functionality. Then it was synthesized by Vivado HLS to obtain the first synthesizable design (Design #1). Table 4.1 listed an overuse for BRAM (BRAM_18K=7392) for Design #1 where the FPGA platform has maximum BRAM_18K=1090. This limitation will lead to the first optimization step which is dividing an image into strips during processing to reduce the required memory usage.

4.3.1.1 Dividing an image into strips

For window-based image processing algorithm, dividing an image into strips is an inevitable step due to the limited number of on-chip memories (BRAMs). In strip processing, loop boundaries and array dimensions are updated to reflect a strip size processing area instead of full image size (i.e. the image height changed from *imgH* to *stripH* where *stripH* =

```
1  for every disparity(d) where d = 0 -> DISP_MAX:
2     for every element(x,y) where x = 0 -> imgW, y = 0 -> 2*winV+1:
3        Abs[y][x] = abs(IMG_R[y][x]-IMG_L[y][x+d]);
4     for every element(x,y) where x = winH -> imgW-winH, y = 0 -> 2*winV+1:
5         row[y][x] =  row[y][x-1] + Abs[y][x] - Abs[y][x-winH];
6     for every element(x,y) where x = winH -> imgW-winH,
7                                  y = winV-cwinV -> winV+cwinV+1:
8        rowE[y-winV+cwinV][x] = rowE[y-winV+cwinV][x-1] + Abs[y][x] - Abs[y][x-
       cwinH];
9     for every element(i) where i = 0 -> imgW:
10       for every element(y) where y = 0 -> winV+1:
11          scr_AB[i] += row[y][i];
12       for every element(y) where y = winV+1 -> 2*winV+1:
13          scr_CD[i] += row[y][i];
14       for every element(y) where y = 0 -> 2*cwinV+1:
15          scr_E[i] += row_E[y][i];
16    for every element(x) where x = winH -> imgW-winH:
17       scoreA = scr_AB[x           ];
18       scoreB = scr_AB[x+winH      ];
19       scoreC = scr_CD[x+winH      ];
20       scoreD = scr_CD[x           ];
21       scoreE =  scr_E[x+winH-cwinH];
22       min_score = scoreE + MIN_2_values{scoreA,scoreB,scoreC,scoreD};
23       if min_score < bestscoreR[x+winH]:
24          bestscoreR[x+winH] = min_score;
25          DISP_IMG_R[x+winH] = d;
26       if min_score < bestscoreL[x+winH+d]:
27          bestscoreL[x+winH+d] = min_score;
28          DISP_IMG_L[x+winH+d] = d;
29 for every element(x) where x = winH -> imgW-winH:
30    dispVal = DISP_IMG_R[x];
31    if abs(DISP_IMG_L[x+dispVal]-dispVal) > 1:
32       DISP_IMG_R[x] = 0;
```

Listing 4.2 – Pseudo code for aggregating the pixels in horizontal then vertical direction (Design #2)

$2\times winV+1$) where the code will be repetitively executed until all the strips in one frame are completely processed.

In multi-window SAD algorithm, the pixels can be summed in three different ways : (i) Listing 4.2 shows that Design #2 aggregates the pixels in the horizontal direction (Lines 4-8) then the result is aggregated in the vertical one to get the score of the window (Lines 9-15). (ii) While in Design #3, the aggregation is done vertically along the column length (Listing 4.3, Lines 4-10) then horizontally along the scanline (Listing 4.3, Lines 11-14). (iii) However, in Design #4, the pixels are aggregated within the window boundary in both directions to get the score value (Listing 4.4, Lines 4-11). Figure 4.4 shows how the pixels are aggregated in Design #4 to obtain the score of a window of dimensions $(X1, Y1)$. A circular buffer of size $(X1+1)$ is used to keep the sum of pixels of each column (C_0, C_1, C_2, ...). Usually *last_ptr* points to the position where the last column is added to the buffer ; while *first_ptr* points

```
1  for every disparity(d) where d = 0 -> DISP_MAX:
2     for every element(x,y) where x = 0 -> imgW, y = 0 -> 2*winV+1:
3        Abs[y][x] = abs(IMG_R[y][x]-IMG_L[y][x+d]);
4     for every element(x) where x = 0 -> imgW:
5        for every element(y) where y = 0 -> winV+1:
6           col_AB[x] += Abs[y][x];
7        for every element(y) where y = winV+1 -> 2*winV+1:
8           col_CD[x] += Abs[y][x];
9        for every element(y) where y = winV-cwinV -> winV+cwinV:
10          col_E[x]  += Abs[y][x];
11    for every element(x) where x = winH -> imgW-winH:
12       scr_AB[x] = scr_AB[x-1] + col_AB[x] - col_AB[x-winH];
13       scr_CD[x] = scr_CD[x-1] + col_CD[x] - col_CD[x-winH];
14       scr_E[x]  =  scr_E[x-1] +  col_E[x] -  col_E[x-2*cwinH];
```

Listing 4.3 – Pseudo code for aggregating the pixels in vertical then horizontal direction (Design #3)

```
1  for every disparity(d) where d = 0 -> DISP_MAX:
2     for every element(x,y) where x = 0 -> imgW, y = 0 -> 2*winV+1:
3        Abs[y][x] = abs(IMG_R[y][x]-IMG_L[y][x+d]);
4     for every element(i) where i = winH -> imgW-winH:
5        for every element(x,y) where x = 0 -> winH+1, y = 0 -> winV+1:
6           scr_AB[i] += Abs[y][i+x-winH];
7        for every element(x,y) where x = 0 -> winH+1, y = winV+1 -> 2*winV+1:
8           scr_CD[i] += Abs[y][i+x-winH];
9        for every element(x,y) where x = winH-cwinH -> winH+cwinH+1,
10                                  y = winV-cwinV -> winV+cwinV+1:
11           scr_E[i] += Abs[y][i+x-winH];
```

Listing 4.4 – Pseudo code for aggregating the pixels within window boundary (Design #4)

to the position next to it as depicted in Fig. 4.4. Listing 4.5 explains how to compute the score of a window of dimensions $(X1, Y1)$. Initially, *last_ptr* points to the first place in the buffer while *first_ptr* points to the next one (Line 7). Every iteration along the image scanline *imgW*, a new column (C) is summed up to the end of the circular buffer (Lines 11-13). The score for window (W_1) is computed by utilizing Box-Filtering technique (Line 14) as depicted in Fig. 4.4. Finally, the values of the pointers are updated to the new positions (Lines 16-23).

Table 4.1 reports the estimated hardware utilization for the three designs. By comparing, we can observe that Design #4 is more efficient in terms of BRAM usage as well as for execution time. Design #4 has 51% and 33% less BRAM utilization than Design #2 and Design #3 with 72% and 59% better performance respectively. Therefore, we will consider Design #4 as a base for the next optimization steps.

4.3.1.2 Using arbitrary precision data types

HLS tools support arbitrary precision data types by defining variables with smaller bit width. Instead of using the native C-based data types of width 8, 16, 32 and 64 bit ; we

```
1  // BUFF_SIZE = X1+1
2  // STARTLINE = 0      // ENDLINE = Y1
3  // stripImgR[stripH][imgW] is the right image strip
4  // stripImgL[stripH][imgW] is the left image strip
5  // array SCORE[] keeps the score of windows
6  int cir_buff[BUFF_SIZE];
7  int sum = 0, scr = 0, scrTmp = 0, first_ptr  = 1, last_ptr  = 0;
8  for(int i = 0; i < BUFF_SIZE; i++)
9     cir_buff[i] = 0;
10 for(int i = 0; i < imgW; i++)
11    for(j = STARTLINE; j <= ENDLINE; j++)
12       sum += abs( stripImgR[j][i] - stripImgL[j][i+d] );
13    cir_buff[last_ptr ] = sum;
14    SCORE[i] = scrTmp = scr + cir_buff[last_ptr] - cir_buff[first_ptr];
15    scr = scrTmp; sum=0;
16    if(first_ptr == BUFF_SIZE-1)
17       first_ptr  = 0;
18    else
19       first_ptr ++;
20    if(last_ptr  == BUFF_SIZE-1)
21       last_ptr  = 0;
22    else
23       last_ptr ++;
```

Listing 4.5 – Calculating the score of a window of dimensions (X1,Y1)

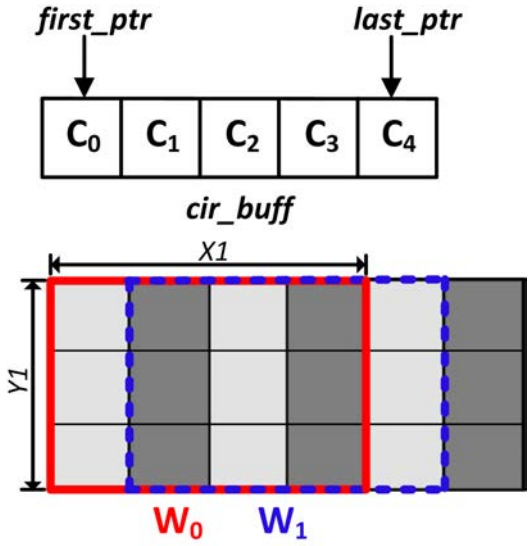

FIGURE 4.4 – Pixel Aggregation in Design #4

can define our variables with adjustable bit width to produce systems of the same accuracy but with less area utilization. In Vivado HLS, the header file *ap_int.h* should be included to define variables with arbitrary precision data types. The format for integer data type is defined as ap_[u]int<W> where u is used to define unsigned values while W is the number

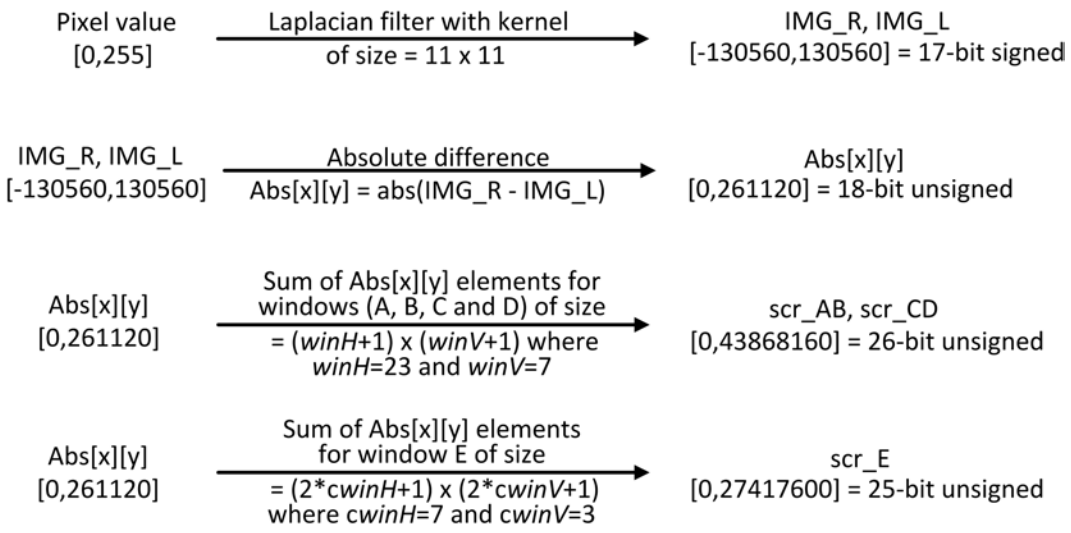

FIGURE 4.5 – Bit-width analysis to define variables with arbitrary precision

of bits of the variable which could be up to 1024 bit-wide [111]. By knowing the lower and upper limits for each variable, we can assign exactly the required number of bits. Figure 4.5 shows how bit-width analysis is applied to define variables with arbitrary precision size. In order to enhance the quality of the final obtained disparity image, the 8-bit input image is firstly filtered by a Laplacian filter for edge detection. In our design, we used a Laplacian filter of window size = 11×11 and kernel value as presented in equation 4.2. After that, the filtered images (IMG_R[y][x] or IMG_L[y][x]) are used to calculate the absolute sum of windows such that the size of windows (A,B,C and D) is 24×8 while the size of window (E) is 15×7. In Table 4.1, Design #5 showed around 31% reduction for LUT and 40% reduction for FF after applying arbitrary precision data types.

$$\text{Kernel\_11x11} = \begin{bmatrix} 0 & 0 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 0 & 0 \\ 0 & 1 & 2 & 5 & 6 & 8 & 6 & 5 & 2 & 1 & 0 \\ 1 & 2 & 8 & 12 & 13 & 13 & 13 & 12 & 8 & 2 & 1 \\ 1 & 5 & 12 & 12 & 0 & -9 & 0 & 12 & 12 & 5 & 1 \\ 2 & 6 & 13 & 0 & -37 & -60 & -37 & 0 & 13 & 6 & 2 \\ 2 & 8 & 13 & -9 & -60 & -88 & -60 & -9 & 13 & 8 & 2 \\ 2 & 6 & 13 & 0 & -37 & -60 & -37 & 0 & 13 & 6 & 2 \\ 1 & 5 & 12 & 12 & 0 & -9 & 0 & 12 & 12 & 5 & 1 \\ 1 & 2 & 8 & 12 & 13 & 13 & 13 & 12 & 8 & 2 & 1 \\ 0 & 1 & 2 & 5 & 6 & 8 & 6 & 5 & 2 & 1 & 0 \\ 0 & 0 & 1 & 1 & 2 & 2 & 2 & 1 & 1 & 0 & 0 \end{bmatrix} \tag{4.2}$$

```
#define stripH 2*winV+1
struct out_axis{
   unsigned char disparity[imgW];
   ap_uint<1> TLast[imgW];
};
void Mutli_Window_SAD(
   int IMG_R[stripH][imgW],
   int IMG_L[stripH][imgW],
   struct out_axis *result
){
   #pragma HLS INTERFACE s_axilite port=return
   #pragma HLS INTERFACE axis port=IMG_R
   #pragma HLS INTERFACE axis port=IMG_L
   #pragma HLS INTERFACE axis port=out_axis
   ..............
   ..............
   ..............
   for(int i = 0; i < imgW; i++){
      if(i < imgW-1){
         result->disparity[i] = DISP_IMG_R[i];
         result->TLast[i] = 0;   }
      else if( i == imgW-1){
         result->disparity[i] = DISP_IMG_R[i];
         result->TLast[i] = 1;   }
      }
}
```

Listing 4.6 – Adding TLAST signal to the result output port

4.3.1.3 Choosing the I/O interface protocol

The generated HLS hardware block can be connected to the other blocks in the design through various types of I/O protocols. Vivado HLS tool offers different ways of communication protocols where the designer is free to choose the one which fits better with his design requirements. During the synthesis process, the top-level function arguments are synthesized as RTL ports where three different classes of ports can be defined :

- **Clock and Reset ports.** HLS tool defined two input ports for clock and reset signal.
- **Block-Level interface protocol.** The block-level port is used to control and check the current state (start, ready, busy or done state) of the HLS block. The port has various configuration options : (i) *ctrl_hs* where the signals (*start*, *idle*, *done* and *ready*) are defined as separated ports. (ii) *axilite_ctrl_hs* where the signals of *ctrl_hs* protocol are grouped into one AXI4-Lite interface. (iii) *ctrl_none* which implements the design without any block-level I/O interface protocol.
- **Port-Level interface protocol.** The port-level port is created for each argument in the top-level function and for the return argument if it exists. The port has various configuration options : (i) *ap_memory* where array-based arguments are implemented by default as an *ap_memory* interface. It is a standard block RAM interface with *data*, *address*, *chip-enable* and *write-enable* ports. (ii) *ap_hs* includes two-way handshaking

with valid and acknowledge signals. (iii) AXI4 interfaces for function arguments including AXI4-Stream (*axis*), AXI4-Lite (*axilite*) and AXI4 master (*m_axi*) for port-level interface.

In our design, the HLS block is connected to a Zynq platform where pixels flow between the Processing System (PS) and the Programmable Logic (PL) through DMA-based communication system ; thus, we chose AXI-Stream for the port-level interface. While AXI-Lite was selected for the block-level interface protocol to control the operation of the hardware block. The defined AXI-Stream protocol by Vivado HLS tool comes only with the fundamental AXI-Stream signals (TDATA, TREADY, TVALID) and it is the role of the designer to define any other required AXI-Stream signals. For DMA-based communication, TLAST signal is needed. Listing 4.6 shows how the top-level function was modified to add TLAST signal to the AXI-Stream communication. For the output argument, it was redefined as a structure consisted of two elements : disparity array and TLAST signal (Lines 2-5). The function arguments were defined as AXI-Stream (axis) interface (Lines 11-14). While streaming the last output element, TLAST is asserted high to signal the end of communication (Line 24). In Table 4.2, Design #6 listed the hardware cost for adding the TLAST signal.

4.3.1.4 Grouping pixels at the I/O ports

The High Performance (HP) bus between the Processing System and Programmable Logic in the Zynq platform is 64-bit data width. If the sent/received pixels are of size less than 64-bit, then the designer can benefit from the available bus to reduce the required communication time by merging pixels during data transfer. This operation requires an additional attention from the designer while separating the pixels at the input ports or merging them at the output ports. In our design, the input pixel is 32-bit width while the output disparity is only 8-bit. Thus, we can merge up to 2 pixels at the input port and up to 8 pixels at the output port. Listing 4.7 shows how the disparity pixels are merged up to 8 pixels before transmission. Design #7 showed 7% improvement in the execution time as listed in Table 4.2.

4.3.2 Optimizations for Exploiting Parallelism

Video processing applications are good candidates for parallel implementation. In this section, we will exploit the inherent parallelism in Multi-window SAD algorithm at different levels to improve the application processing time.

4.3.2.1 Task-level parallelism

In task-level parallelism, independent data tasks can be executed concurrently. For 5-window SAD algorithm, the score of window (B) is used after ($winH+1$) pixel shift as a score for a new window (A) along the same scanline. The same case applied for windows (C)

```
1  #define imgW   640
2  #define imgW_8 80
3  struct out_axis{
4     unsigned long int disparity_64[imgW_8];
5     ap_uint<1> TLast[imgW_8];
6  };
7  void Mutli_Window_SAD(
8     int IMG_R[stripH][imgW],
9     int IMG_L[stripH][imgW],
10    struct out_axis *result
11 ){
12    ..............
13    ..............
14    // sending pixels at the output port
15    for(int i = 0; i < imgW_8; i++){
16       if(i < imgW_8-1){
17          result->disparity_64[i] =
18                ( (unsigned long int) DISP_IMG_R[8*i  ]       )
19             + ( (unsigned long int) DISP_IMG_R[8*i+1] << 8  )
20             + ( (unsigned long int) DISP_IMG_R[8*i+2] << 16 )
21             + ( (unsigned long int) DISP_IMG_R[8*i+3] << 24 )
22             + ( (unsigned long int) DISP_IMG_R[8*i+4] << 32 )
23             + ( (unsigned long int) DISP_IMG_R[8*i+5] << 40 )
24             + ( (unsigned long int) DISP_IMG_R[8*i+6] << 48 )
25             + ( (unsigned long int) DISP_IMG_R[8*i+7] << 56 );
26          result->TLast[i] = 0;   }
27       else if( i == imgW_8-1){
28          result->disparity_64[i] =
29                ( (unsigned long int) DISP_IMG_R[8*i  ]       )
30             + ( (unsigned long int) DISP_IMG_R[8*i+1] << 8  )
31             + ( (unsigned long int) DISP_IMG_R[8*i+2] << 16 )
32             + ( (unsigned long int) DISP_IMG_R[8*i+3] << 24 )
33             + ( (unsigned long int) DISP_IMG_R[8*i+4] << 32 )
34             + ( (unsigned long int) DISP_IMG_R[8*i+5] << 40 )
35             + ( (unsigned long int) DISP_IMG_R[8*i+6] << 48 )
36             + ( (unsigned long int) DISP_IMG_R[8*i+7] << 56 );
37          result->TLast[i] = 1;   }
38       }
39 }
```

Listing 4.7 – Merging 8 pixels at the output port

and (D). Thus, only three score calculation loops are needed for windows (A/B, C/D and E). In order to execute data-independent loops in parallel, we have : (i) To duplicate the common input pixels between the three loops if exist. (ii) To rewrite them in separated functions to allow the HLS tool to schedule them in parallel. Listing 4.8 explains how task-level parallelism is applied. The input right/left strips are distributed on the local arrays (IMG_R_AB[], IMG_L_AB[], IMG_R_CD[], ..., ..., ..., IMG_L_E[]) (Lines 19-47). The common image lines between windows are duplicated to allow data-independent windows calculations. For example, the common image lines between windows A/B and E are duplicated to the local arrays for both of them (Lines 30-37). After pixel distribution, three function calls are executed

Design	Optimization	Slice	FF	LUT	BRAM 18K	exec. time (ms)
#6	Adding TLAST	996	1575	2619	49	6307
#7	Grouping pixels	1135	1820	3080	49	5865
#8	Task-level parallelism	1110	2002	3339	67	2658
#9	Calculating 4 disparity lines	2790	4578	7796	102	815
#10	Calculating 8 disparity lines	5012	8502	14027	204	432
#11	Calculating 12 disparity lines	6594	12563	18476	252	339
#12	Loop pipelining	1161	2004	3546	67	1174
#13	Removing false dependency	1115	2030	3433	67	1002
#14	Data-level parallelism	2771	6365	8155	59	313

TABLE 4.2 – Synthesis results reported by Vivado HLS for each optimization step

to calculate the score for windows (A/B, C/D and E) where the HLS tool will schedule their executions in parallel since there is no common data between them (Lines 49-51). In Table 4.2, Design #8 reported the effect of applying task-level parallelism where the processing time was improved by around 50%.

4.3.2.2 Pipeline-level parallelism

In pipeline-level parallelism, the computation is divided into stages where it is possible to execute the pipelined stages in parallel. We applied pipeline-level parallelism in two different ways : (i) By restructuring the code manually. (ii) By applying HLS directives like LOOP_PIPELINE. Figure 4.6 depicts that there is only one image line difference between two adjacent strips. For calculating one disparity line, a strip of height = 2**win_V*+1 is needed ; while for four adjacent disparity lines, a strip of height = 2**win_V*+4 is required. Thus, we can increase the height of strip to benefit from the sent pixels to calculate several disparity lines. We tried to calculate 4, 8 and 12 disparity lines while using the same pipeline for Designs #9, #10 and #11 respectively as listed in Table 4.2.

The other way of performing pipeline-level parallelism is by adding LOOP_PIPELINE directive to the defined for-loops in the algorithm. This loop transformation is done automatically by the help of the tool without the need to modify the code. Design #12 in Table 4.2 reported the hardware cost and the execution time after applying pipelining for Design #8. When LOOP_PIPELINE directive is applied, the HLS tool tries to schedule all the loop iterations just one clock cycle far from each other (i.e. Iteration Interval (II) = 1) but sometimes due to inter loop-dependency, II = 1 can not be achieved. It is the role of the designer to check the positions where II > 1 are reported then to direct the tool to remove the false loop dependency if exists by introducing LOOP_DEPENDENCE directive.

Lines 23-28 from Listing 4.2 are copied in Listing 4.9 to show a case for false loop dependency which can be removed to allow better loop pipelining. When pipeline

```
1  #define imgW 640
2  #define imgW_2 320
3  #define start_E winV-cwinV
4
5  void scr_window_AB_CD(int img_R[winV+1][imgW], int img_L[winV+1][imgW],
6                        int score[imgW] )
7  { ............................. }
8
9  void scr_window_E(int img_R[2*cwinV+1][imgW], int img_L[2*cwinV+1][imgW],
10                   int score[imgW] )
11 { ............................. }
12
13 void Multi_Window_SAD( int IMG_R[stripH][imgW], int IMG_L[stripH][imgW],
14                         struct out_axis *result
15 ){
16  int IMG_R_AB[winV+1  ][imgW], IMG_L_AB[winV+1  ][imgW];
17  int IMG_R_CD[winV+1  ][imgW], IMG_L_CD[winV+1  ][imgW];
18  int IMG_R_E[2*cwinV+1][imgW], IMG_L_E[2*cwinV+1][imgW];
19  //distributing input scanlines
20  for(int num = 0; num < 2*winV+1; num++)
21  {
22   if(num < winV-cwinV){
23    for(int i = 0; i < imgW_2 ; i++){
24     IMG_R_AB[num][2i]   = (int)(IMG_R[num][i]      );
25     IMG_R_AB[num][2i+1] = (int)(IMG_R[num][i] >> 32);
26     IMG_L_AB[num][2i]   = (int)(IMG_L[num][i]      );
27     IMG_L_AB[num][2i+1] = (int)(IMG_L[num][i] >> 32);
28    }
29   }
30   else if(num < winV+1){
31    for(int i = 0; i < imgW_2 ; i++){
32     IMG_R_AB[num][2i]   = IMG_R_E[num-start_E][2i]   = (int)(IMG_R[num][i]    );
33     IMG_R_AB[num][2i+1] = IMG_R_E[num-start_E][2i+1] = (int)(IMG_R[num][i]>>32);
34     IMG_L_AB[num][2i]   = IMG_L_E[num-start_E][2i]   = (int)(IMG_L[num][i]    );
35     IMG_L_AB[num][2i+1] = IMG_L_E[num-start_E][2i+1] = (int)(IMG_L[num][i]>>32);
36    }
37   }
38   else if(num == winV+1){
39    ........................
40   }
41   else if(num <= winV+cwinV+1){
42    ........................
43   }
44   else{
45    ........................
46   }
47  }
48
49  scr_window_AB_CD(IMG_R_AB, IMG_L_AB, scr_AB );
50  scr_window_AB_CD(IMG_R_CD, IMG_L_CD, scr_CD );
51  scr_window_E(IMG_R_E, IMG_L_E, scr_E);
52
53 }
```

Listing 4.8 – Code restructuring for applying task-level parallelism

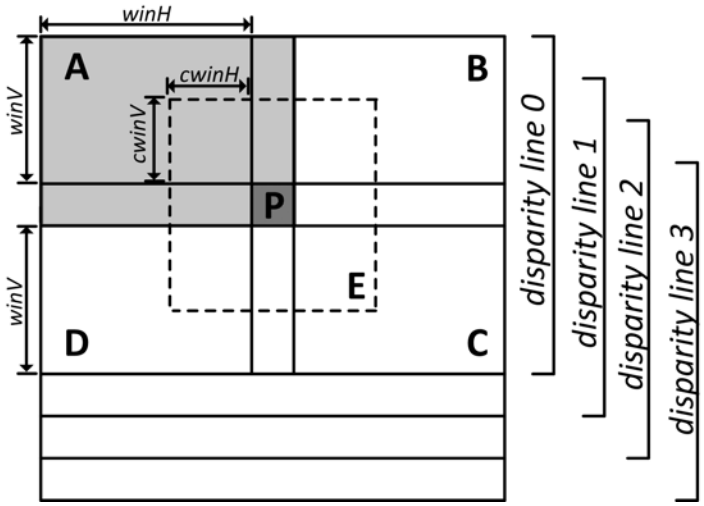

FIGURE 4.6 – Enlarging the strip height to calculate 4 disparity lines

```
1  for every disparity(d) where d = 0 -> DISP_MAX:
2     ......................
3     for every element(x) where x = winH -> imgW-winH:
4        ..........................
5        if min_score < bestscoreR[x+winH]:
6           bestscoreR[x+winH] = min_score;
7           DISP_IMG_R[x+winH] = d;
8        if min_score < bestscoreL[x+winH+d]:
9           bestscoreL[x+winH+d] = min_score;
10          DISP_IMG_L[x+winH+d] = d;
11 ......................
```

Listing 4.9 – A case for false loop dependency

directive is added to the for-loop defined between Lines 3-10; the HLS tool signalled a warning for bestscoreL[ ] array `"Warning: unable to enforce a carried dependency constraint (II=1, distance=1")`. By examining the for-loop, we could notice that the HLS tool considered (d) as a variable of undetermined value during the loop iterations between Lines 3-10. Thus, it assumed a dependency between reading from bestscoreL[ ] at Line 8 and writing to it at Line 9. Accordingly, the HLS tool scheduled the pipeline with II=2 to avoid any conflicts between the pipelined loop iterations when read and write operations are requested in the same clock cycle to the same array location as shown in Fig. 4.7(a). But this dependency is a false one because variable (d) does not change its value during the iterations of the loop defined between Lines 3-10. So, we are confident to define a false dependency directive for bestscoreL[ ] array as following : `"#pragma HLS DEPENDENCE variable=bestscoreL array inter false"` to obtain II = 1 as depicted in Fig. 4.7(b). In Table 4.2, Design #13 showed 15% gain in execution time than Design #12 when false inter loop dependency is removed.

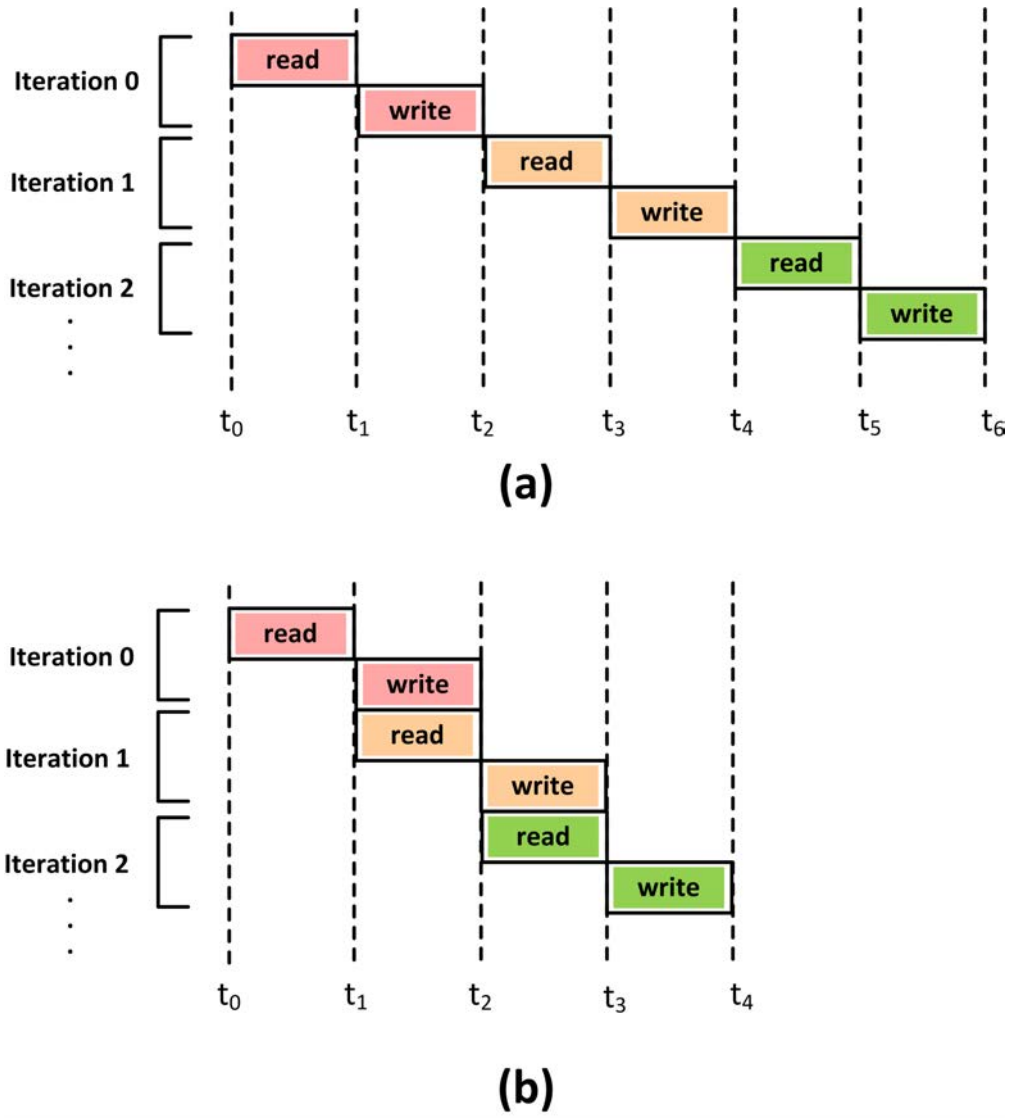

FIGURE 4.7 – (a) Loop scheduling when Iteration Interval (II) = 2. (b) Loop scheduling when Iteration Interval (II) = 1 after applying false loop dependency

4.3.2.3 Data-level parallelism

When the computation process is repeated without true loop-carried dependency between the iterations ; then, it can be duplicated to operate on different set of data in parallel. We applied data-level parallelism in two different ways : (i) By applying ARRAY_PARTITION and LOOP_UNROLL directives. (ii) By increasing the number of parallel processing elements.

The goal of array partitioning is to boost the system throughput at the expense of increasing the used hardware resources. ARRAY_PARTITION directive partitions an array

either partially into small arrays or entirely as individual elements to increase the available number of read/write ports. In this design, we used this directive in two different cases : (i) The local arrays defined at (Lines 16-18, Listing 4.8) are two-dimensional arrays of ($winV+1$) rows and $imgW$ columns. By applying ARRAY PARTITION directive with options "`type=block, factor=8, dimension=1`", the arrays will be partitioned into smaller arrays, where each image line will be stored in an individual BRAM structure. By this configuration, one column of pixels can be read in the same clock cycle. (ii) The circular buffer defined at (Line 6, Listing 4.5) will be partitioned completely into individual registers " `type=complete, dimension=1`" to ease the process of data accessing and due to its smaller size.

LOOP_UNROLL directive duplicates the computation process to operate on a different set of data by creating multiple copies of the loop body. The loop can be partially unrolled by creating N copies of the loop body if factor N is defined ; otherwise, the loop is fully unrolled by default. In our design, the arrays are implemented as BRAMs with physical dual-port memory. Consequently, we can profit from the available dual-port of the BRAMs to unroll the loops with factor = 2. Design #14 reported 70% improvement in the execution time with 2.3x and 3.1x increase in LUT and FF respectively as listed in Table 4.2.

The other way of exploiting data-level parallelism is by increasing the number of parallel processing elements. This can be achieved by defining a new top-level function that includes multiple instances of Design #14 operating in parallel. In the next chapter, we will introduce ViPar tool which will explore the design space for the parallel architecture that better satisfies the system constraints.

4.3.3 Experimental Results

Zynq ZC706 FPGA evaluation board (XC7Z045-FFG900) is used to process 5-window SAD algorithm for input images of size 640×480. The left/right images are captured by a stereoscopic camera system STERSEE which is attached to the FPGA board through an Ethernet port as shown in Fig. 4.8. The hardware cores in the design are initialized and controlled by the Processing System through AXI-Lite communication interface (*S_AXI_LITE* port is not connected in the figure for simplicity). The pixels transfer between Processing System (PS) and Programmable Logic (PL) through High Performance (HP) buses with the help of AXI-DMA cores. The input images are preprocessed by a Laplacian filter (*HLS_Laplace_Filter* core) of kernel size = 11×11 to enhance the quality of the obtained disparity image. After that, the preprocessed images are sent into strips to *HLS_SAD* core for stereo matching processing, and the result is written back to the DDR memory attached to the Processing System. For demonstration, the disparity image can be sent to the output screen through *AXI-VDMA* and *HDMI* cores.

Each hardware core with AXI_Lite interface is treated as a memory-mapped IO device with an address space defined in *xparameters.h* file. In the beginning, the Processing System

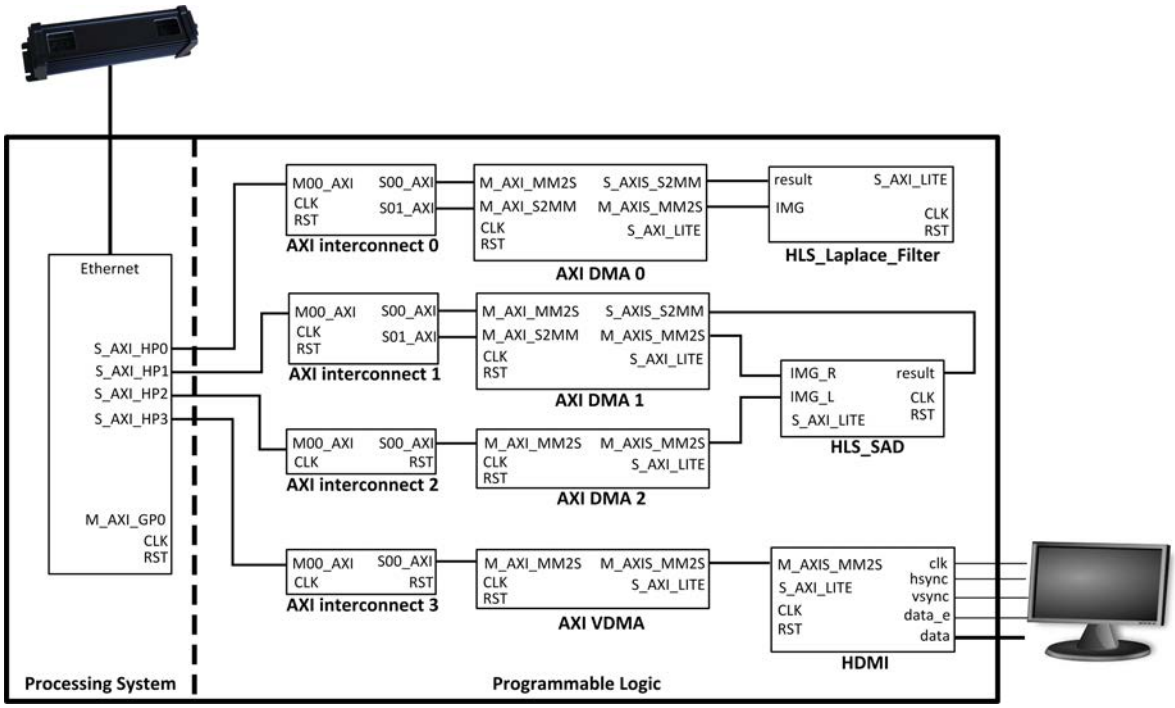

FIGURE 4.8 – System architecture block diagram

```
1  int HLS_SAD_init(HLS_SAD *stereo_ptr)
2  {
3    HLS_SAD_Config *cfg_ptr;
4    int status;
5    cfg_ptr = HLS_SAD_LookupConfig(HLS_SAD_DEVICE_ID);
6    if (!cfg_ptr)
7    {
8      print("ERROR: Lookup for accelerator configuration failed.\n\r");
9      return XST_FAILURE;
10   }
11   status = HLS_SAD_CfgInitialize(stereo_ptr, cfg_ptr);
12   if (status != XST_SUCCESS)
13   {
14     print("ERROR: Could not initialize accelerator.\n\r");
15     return XST_FAILURE;
16   }
17   return status;
18 }
```

Listing 4.10 – The initialization for HLS_SAD core

(PS) obtains the configuration information of the device by the help of its ID (Line 5-10, listing 4.10). But when Memory Management Unit (MMU) is used, the effective address of the hardware core is different from that defined in *xparameters.h* file and a translation is required. After initializing the cores, the left/right images are sent to *HLS_Laplace_Filter*

```
1  int main()
2  {
3    XAxiDma axiDma_0, axiDma_1, axiDma_2;
4    HLS_Laplace_Filter laplaceFilter;
5    HLS_SAD stereoHW;
6    int Lap_IMG_R[307200], Lap_IMG_L[307200];
7    unsigned char disparity[307200];
8
9    HLS_Laplace_Filter_Start(&laplaceFilter);
10   // Applying Laplace filter to the right image
11   status = XAxiDma_SimpleTransfer(&axiDma_0, (u32)&imageR[0], 307200*sizeof(char)
       , XAXIDMA_DMA_TO_DEVICE);
12   Xil_DCacheFlushRange((unsigned)&imageR[0], 307200 * sizeof(char));
13   status = XAxiDma_SimpleTransfer(&axiDma_0, (u32)&Lap_IMG_R[0], 307200*sizeof(
       int), XAXIDMA_DEVICE_TO_DMA);
14   Xil_DCacheInvalidateRange((unsigned)&Lap_IMG_R[0], 307200*sizeof(int));
15   while(!HLS_Laplace_Filter_IsReady(&laplaceFilter))
16     ; // blocked till processing ends
17
18   HLS_Laplace_Filter_Start(&laplaceFilter);
19   // Applying Laplace filter to the left image
20   status = XAxiDma_SimpleTransfer(&axiDma_0, (u32)&imageL[0], 307200*sizeof(char)
       , XAXIDMA_DMA_TO_DEVICE);
21   Xil_DCacheFlushRange((unsigned)&imageL[0], 307200 * sizeof(char));
22   status = XAxiDma_SimpleTransfer(&axiDma_0, (u32)&Lap_IMG_L[0], 307200*sizeof(
       int), XAXIDMA_DEVICE_TO_DMA);
23   Xil_DCacheInvalidateRange((unsigned)&Lap_IMG_L[0], 307200*sizeof(int));
24   while(!HLS_Laplace_Filter_IsReady(&laplaceFilter))
25     ; // blocked till processing ends
26
27   // Processing stereo matching algorithm in strips
28   for(line=0; line<468; line=line+4)
29   {
30     HLS_SAD_Start(&stereoHW);
31     // Sending the right preprocessed strip through AXI_DMA_1
32     status = XAxiDma_SimpleTransfer(&axiDma_1, (u32)&Lap_IMG_R[640*line], 11520 *
        sizeof(int), XAXIDMA_DMA_TO_DEVICE);
33     Xil_DCacheFlushRange((unsigned)&Lap_IMG_R[640*line], 11520 * sizeof(int));
34     // Sending the left preprocessed strip through AXI_DMA_2
35     status = XAxiDma_SimpleTransfer(&axiDma_2, (u32)&Lap_IMG_L[640*line], 11520 *
        sizeof(int), XAXIDMA_DMA_TO_DEVICE);
36     Xil_DCacheFlushRange((unsigned)&Lap_IMG_L[640*line], 11520 * sizeof(int));
37     // Receiving the disparity lines through AXI_DMA_1
38     status = XAxiDma_SimpleTransfer(&axiDma_1, (u32)&disparity[4480+640*line],
       2560 * sizeof(unsigned char), XAXIDMA_DEVICE_TO_DMA);
39     Xil_DCacheInvalidateRange((unsigned)&disparity[4480+640*line], 2560 * sizeof(
       unsigned char));
40     // Waiting until the result is written back
41     while(!HLS_SAD_IsReady(&stereoHW))
42       ; // spin
43   }
44 }
```

Listing 4.11 – A snapshot of the SW code running on the Processing System during stereo matching processing

core for preprocessing. *AXI_DMA_0* is used during the pixel transfer where the transmitted image is of type char, and the resulted image is of type int (Lines 11-14, Listings 4.11). *HLS_Laplace_Filter* starts processing when it receives a start signal from the Processing System through AXI-LITE bus (Line 9). During filter processing, the system is blocked checking the status of the core if it is ready for the next input or not (Lines 15-16). During strip processing, the *HLS_SAD* core is called several times till all the image strips are processed (Lines 27-43). Listing 4.11 shows the case where the size of strip is enlarged to process 4 disparity lines using the same pipeline (Line 28). The filtered right strip is sent to the *HLS_SAD* core through *AXI_DMA_1* (Lines 31-33) while the left strip is sent through *AXI_DMA_2* core (Lines 34-36). The resulted disparity is stored back through *AXI_DMA_1* in disparity[ ] array (Lines 37-39). During stereo processing, the system is blocked waiting for *ready* signal to start the next strip processing (Lines 41-42).

4.4 Conclusion

High-level synthesis tools shorten the design efforts by translating high-level code descriptions into hardware designs. However, it is the role of the designer to present the high-level codes in a format acceptable by the HLS tool. In this chapter, we presented a set of guidelines that the designer is advised to follow to modify the high-level code for producing an efficient hardware implementation. As a particular case, we showed these guidelines for 5-window SAD stereo matching algorithm. Two classes of optimizations were introduced : optimizations for targeting an efficient hardware implementation and optimizations for exploiting the levels of parallelism inherent in the application. For each optimization step, we showed its impact on the overall design quality concerning hardware cost and execution time. Finally, we presented the architecture of the hardware implementation for the stereo matching algorithm as well as the software part.

Chapter 5

ViPar : A Tool for Design Space Exploration

5.1 Introduction

In the last chapter, a set of optimization steps were applied to the high-level code for Multi-Window SAD algorithm to obtain an efficient hardware implementation. Applying pipeline-level and data-level parallelism alongside executing the application at different operating frequencies will produce a space of various design alternatives that need to be explored. Designing, implementing and doing measurements for performance and power for every single design is not advisable under time-to-market constraints. Instead of that, it is preferable to do fast estimations for area utilization, power and performance for all design alternatives then according to the system constraints, full implementations will be built only for the candidate designs.

For a given design, defining the priority of constraints could vary from one application to another. For example, power consumption is a key factor for battery-based systems while hardware resources matter if several functionalities would be embedded on the same chip. In some other cases, timing is crucial for safety-critical applications while Quality-of-Service is vital for interactive or multimedia applications. During the design phase, it is the role of the designer to define the priorities of system constraints then to explore the design space for the implementation that could efficiently satisfy the application requirements.

In this chapter, the design space was built by varying the level of parallelism at different operating frequencies for three initial designs obtained from pipeline-level parallelism (*pipe_4*, *pipe_8* and *pipe_12*). The obtained designs have various trade-offs regarding hardware resources, power consumption, execution time and operating frequency. Our objective is to explore the possible hardware designs then choose the one that fit with our requirements. We developed ViPar tool to automate the design space exploration process through the following steps : (1) We estimated both resource utilization and performance for each design based on the values obtained at parallelism level = 1. (2) We introduced an empirical power model to estimate the power consumption for each design based on resource utilization and operating frequency. (3) We generated automatically the high-level description codes for the parallel video processing architectures. In the experimental results, we will compare the estimated values with the measured ones in order to evaluate how much the estimations are correct. We will show an example of how to select a design as a solution according to the given system constraints.

5.2 ViPar Tool

Design productivity of complex systems is increased significantly by using high-level synthesis tools. During the design phase, it is much easier to write, modify and verify complex algorithms described in high-level languages. Figure 5.1 depicts an array-based video processing architecture where N processing elements are running in parallel. Each processing

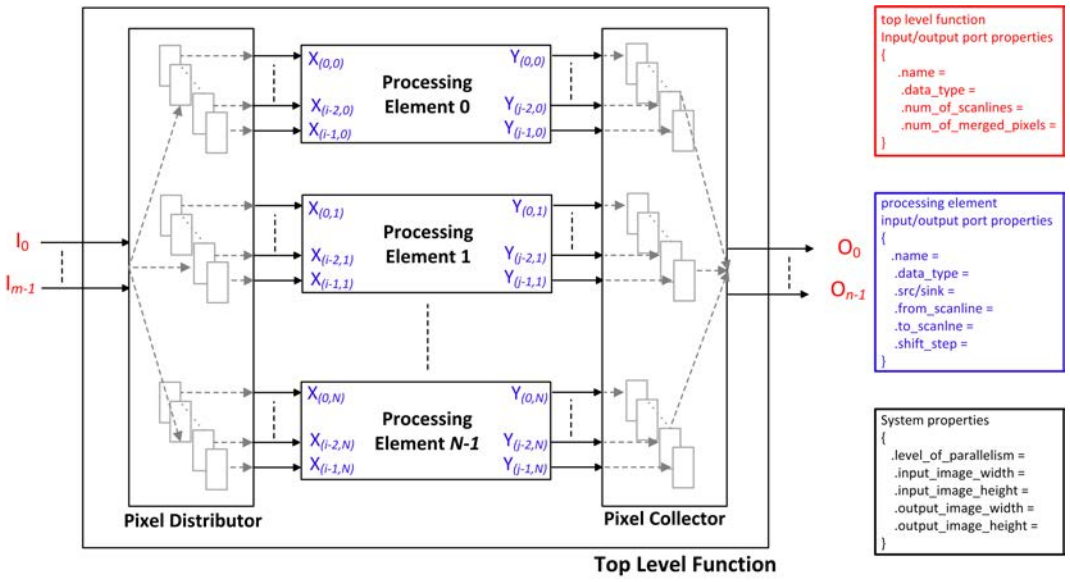

FIGURE 5.1 – Array-based video processing architecture

element has i input ports (X_0, X_1, ..., X_{i}) and j output ports (Y_0, Y_1, ..., Y_{j}). The input pixel streams (I_0, I_1, ..., I_{m}) are copied and distributed to individual array structures through *Pixel Distributor*. After processing, *Pixel Collector* stores the pixels in arrays before streaming them out in order through system output ports (O_0, O_1, ..., O_{n}).

When the architecture depicted in Fig. 5.1 is described in high-level language; we have to code how the pixels will be distributed over the parallel processing elements and how they will be collected back to stream the output. To allow data-level parallelism execution, the same image scanline could be mapped to several input ports of the same or different processing elements. It could be feasible to write the distribution/collection subroutines manually for architectures of few processing elements, but it will be a real challenge to do so for an architecture of large number of processing cores. Another challenge arises when a large number of parallel architectures are examined in order to find an efficient implementation that fulfils the design requirements. Consequently, coding these architectures manually is a time-consuming and an error-prone process. To address the above challenges, we developed ViPar tool which explores the design space then automatically generates the high-level codes for the best candidate parallel architectures. The generated code is then compiled by Vivado HLS to obtain the corresponding RTL design. Our objective is to show how ViPar increases the design productivity by exploring and building video processing architectures of large number of parallel processing elements.

FIGURE 5.2 – Design flow with ViPar tool

Synthesis/Implementation Strategy Name	Description
Vivado Implementation Defaults	Balances runtime with trying to achieve timing closure.
Performance_Explore	Uses multiple algorithms for optimization, placement, and routing to get potentially better results.
Area_Explore	Uses multiple optimization algorithms to get potentially fewer LUTs.
Power_DefaultOpt	Adds power optimization (power_opt_design) to reduce power consumption.

TABLE 5.1 – Different strategies with different objectives

5.2.1 ViPar Tool Design Flow

Figure 5.2 depicts the design flow for ViPar tool where the initial inputs are the metrics of the design at parallelism level = 1. These design metrics include area utilization, power consumption and performance. For area estimation, we keep increasing the level of parallelism till one of the resources (Slice or BRAM) is completely utilized. The upper bound for resource utilization depends on the selected FPGA chip during the exploration process. The produced set of design alternatives are then estimated for power and performance as we will explain in Sections 5.3, 5.4 and 5.5. According to the system constraints, the design space is minimized to the set of the candidate designs. Then high-level code generation tool is used to generate the design C++ files for each design candidate automatically. The generated High-level description codes in addition to the HLS optimizations/User constraints are considered as the inputs for the High-level Synthesis Tool to obtain the RTL design. Later, the RTL design is implemented to obtain the design bit stream. The design metrics are measured experimentally to verify how far the estimations are from the real values and to make sure that the system constraints are indeed fulfilled.

5.3 Area Estimation

Two factors affect the resource utilization of the implemented design which are :

- **Synthesis/Implementation strategy**. Vivado Design Suite offers a set of pre-defined strategies in order to obtain the hardware implementation. It is a multi-objective problem where each strategy has a certain objective to optimize (power, area, performance, etc.). Table 5.1 shows some of these strategies and their objectives.
- **Operating frequency**. The implementation tool tries to achieve timing closure alongside satisfying the objective of the applied strategy. At higher operating frequencies, the tool allocates more hardware resources in order to satisfy the timing constraints.

Parallelism Level	Slice		FF		LUT		BRAM_18K	
	Base	Parallel	Base	Parallel	Base	Parallel	Base	Parallel
1	2624	7910	8645	20258	6643	24520	19	112
2	2611	15811	8645	40440	6638	49126	19	224
3	2714	23834	8645	60623	6643	73850	19	336
4	2707	30933	8645	80799	6646	98582	19	448
5	2866	37460	8645	100979	6645	123409	19	560
6	2686	43651	8645	121177	6640	149088	19	672
7	2118	48904	8645	141370	6605	175503	19	784
8	1736	52734	8645	161550	6559	199714	19	896

TABLE 5.2 – Base and parallel hardware cost for designs at different parallelism levels

Default synthesis strategies are frequently used in the design flow unless they failed to satisfy the timing constraints. In such cases, *Performance_Explore* synthesis strategies could be used for example in order to get better results. Figure 5.3 shows the relation between LUT and FF utilization at different parallelism levels. In this figure, *Default* synthesis strategies were used at two different operating frequencies 100 and 200 MHz while *Performance_Explore* synthesis strategy was used at 100 MHz. We can deduce the following observations : (i) Utilization for LUT and FF increases linearly with the increase of the parallelism level. (ii) At the same parallelism level, the observed utilization varies either because of using different synthesis strategies or because of using different operating frequencies. (iii) The difference in LUT/FF utilization between designs implemented at the same parallelism level is small at low levels then becomes more significant at high levels of parallelism.

The hardware cost in terms of Slice, FF, LUT and BRAM for the parallel architecture depicted in Fig. 5.1 could be divided into two parts :

- **Base cost**. It represents the required resources for implementing the basic blocks which exist in every single design alternative. These basic blocks could be like AXI-DMA blocks for pixels transfer, AXI-interconnect blocks, AXI-VDMA block for AXI4-Stream video target peripherals, etc.
- **Parallel cost**. It represents the required hardware resources for implementing the processing elements of the parallel architecture depicted in Fig. 5.1.

Table 5.2 lists the hardware cost for an application implemented at different parallelism levels. At each parallelism level, the hardware cost for Slice, FF, LUT and BRAM is separated into *Base cost* and *Parallel cost*. It is worth mentioning that these designs were implemented using *Default* synthesis strategy at operating frequency of 100 MHz. From the synthesis results, we can observe that the *Base cost* values for FF, LUT and BRAM are almost the same value regardless which parallelism level is implemented. This observation agrees with our definition of *Base cost* as it is the required resources for implementing the basic blocks

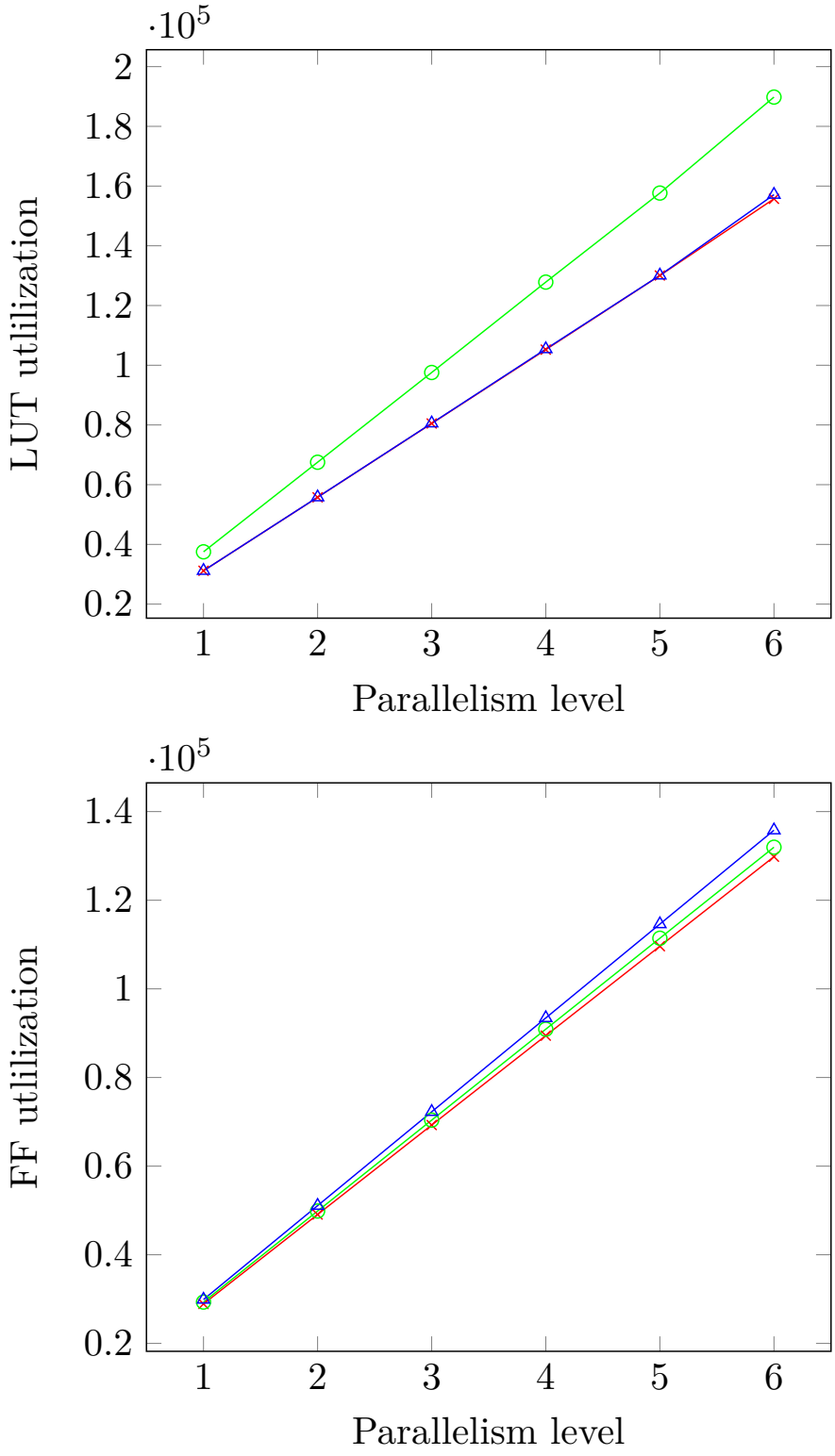

FIGURE 5.3 – LUT/FF utilization at different parallelism level for *Default* strategy at 100 MHz —×—, *Default* strategy at 200 MHz —△— and *Performance_Explore* strategy at 100 MHz —○—

which exist in every single design.

For Slice utilization, it varies with a percentage ranging between 0.5 to 33.8%. The reason for this wide variation is that the slices are partially utilized by the synthesis tool in order to decrease the congestion of wires in the design to satisfy the timing constraints. But when the design size increases, the Slice occupancy increases as well and consequently fewer Slices are dedicated to the *Base cost*. In fact, Slice is not considered as an independent hardware resource ; for example, in Zynq ZC706 FPGA board (XC7Z045-FFG900), each Slice is composed of 8 FFs and 4 LUTs. So if we tried to obtain the Slice number by dividing the used LUT by 4 or by dividing the used FF by 8 then we will discover that the resulted values for Slice utilization do not match with those listed in Table 5.2. For instance, at parallelism level =3, the numbers of used LUT and FF for *Base cost* are 6643 and 8645 respectively. By dividing those values to get the Slice number then it will 6643/4 = 1661 or 8645/8 = 1081 ; however the reported Slice value in Table 5.2 was 2714. This difference can be explained that the synthesis tool tends to use more Slices to avoid full Slice occupancy so that the timing constraints are met.

For the parallel hardware cost, Fig. 5.3 depicts that it increases linearly with the increase of the level of parallelism. Therefore, we can assume that the base hardware cost is the same for all designs while the parallel hardware cost is increasing linearly. Our estimations are based on the area utilization obtained at parallelism level = 1 ; therefore, in order to obtain correct estimations at higher parallelism levels, the initial values should be obtained from a design implemented at the same operating frequency while using the same strategy.

5.3.1 Estimated utilization for LUT, FF and BRAM

From the previous observations (Figure 5.3 and Table 5.2), we can deduce a linear relation for estimating the resource utilization for LUT, FF and BRAM as follows :

$$\text{Estimated Utilization}|_{parallelism\ level\ =\ N} = Base\ cost\ +\ N * Utilization|_{parallelism\ level\ =\ 1} \quad (5.1)$$

Where *Base_cost* is the hardware cost for the basic blocks in the design and N is the level of parallelism. Table 5.3 shows that the estimation error for FF, LUT and BRAM. The estimation error ranges between 0.15 - 0.3% and 0.14 - 2.1% of the estimated value for FF and LUT respectively. This error arises because the synthesis tool could use less or more FF or LUT in order to satisfy the timing constraints. We can also conclude that the error is within the acceptable limits which reflects a correct estimation relation. For BRAM estimation, we observed an exact estimation because BRAMs represent the memory structure in the design where each processing channel in the parallel architecture should have exactly

Parallelism Level	1	2	3	4	5	6	7	8
FF								
Estimated	28903	49161	69419	89677	109935	130193	150451	170709
Actual	28903	49085	69268	89444	109624	129822	150015	170195
Error (%)	0	0.15	0.22	0.26	0.28	0.28	0.28	0.3
LUT								
Estimated	31163	55683	80203	104723	129243	153763	178283	202803
Actual	31163	55764	80493	105228	130054	155728	182108	206273
Error (%)	0	0.14	0.36	0.48	0.63	1.28	2.1	1.7
BRAM_18K								
Estimated	131	243	355	467	579	691	803	915
Actual	131	243	355	467	579	691	803	915
Error (%)	0	0	0	0	0	0	0	0

TABLE 5.3 – Estimated and actual hardware utilization for FF, LUT and BRAM_18K at different parallelism levels

the same number of arrays.

5.3.2 Estimated utilization for Slice

We have two methods to estimate the Slice utilization either by applying equation 5.1 as used before to estimate the utilization for FF, LUT and BRAM or by profiting from the fact that the Slice is composed of FFs and LUTs (for Zynq ZC706, one Slice consists of 8 FFs and 4 LUTs). Based on this fact, the estimated Slice utilization can be formulated as follows :

$$\text{Estimated Slice Utilization}|_{parallelism\ level\ =\ N} = Max\ \{\ \frac{estimated\_LUT|_{parallelism\ =\ N}}{num\_LUT\_per\_Slice}\ ,\ \frac{estimated\_FF|_{parallelism\ =\ N}}{num\_FF\_per\_Slice}\ \} \quad (5.2)$$

Where N is the level of parallelism, $estimated\_LUT$ is the estimated LUT utilization and $estimated\_FF$ is the estimated register utilization at parallelism level=N. $num\_LUT\_per\_Slice$, and $num\_FF\_per\_Slice$ is the number of LUT and FF in one Slice (for Zynq ZC706, $num\_LUT\_per\_Slice = 4$ and $num\_FF\_per\_Slice = 8$). Table 5.4 compares the estimation percentage error when the Slice utilization is calculated using equations 5.1 and 5.2 respectively. For designs from 1 to 6, we could notice that method (2) showed an estimation error ranges between 20.5 - 35.2% while method (1) showed only an estimation error between 0.12 - 7.58%. For the last two designs (# 7 and # 8), the values computed by method (1) exceed the maximum Slice number for Zynq ZC706 (max Slice = 54650). This will lead us to initially refuse these estimations and accept only the one estimated by method (2). In summary, both methods will be used as illustrated in Fig. 5.4 where method (1) is used

Parallelism level	Actual value	Estimation method (1) (equation 5.1)		Estimation method (2) equation (5.2)	
		Estimated value	Error (%)	Estimated value	Error (%)
1	10534	10534	0	7791	35.2
2	18422	18444	0.12	13921	24.4
3	26548	26354	0.73	20051	32.4
4	33640	34264	1.82	26181	28.5
5	40326	42174	4.38	32311	24.8
6	46337	50084	7.48	38441	20.5
7	51022	57994	12.02	44571	14.5
8	54470	65904	17.35	50701	7.5

TABLE 5.4 – Estimated Slice utilization using two different methods : estimation method (1) and estimation method (2) by applying equation 5.1 and 5.2 respectively

till the estimated Slice value hits the maximum Slice boundary where we will switch to use method (2).

5.4 Power Estimation Model

There are three types contribute to the power consumption in FPGA : (1) **Static power**. It is the power consumed when there is no signal transition occurred at the logic gates. As gate technology scales down, static power becomes a more dominant factor in the total chip power consumption. (2) **Short-circuit power**. This power is dissipated when a signal transition occurs at a gate output where both the pull-up and pull-down transistors can be conducted simultaneously for a short period. (3) **Dynamic power**. This power is resulted from the design activity and varies over time with the design activity.

Dynamic power could be further broken down into power consumed by clocks, interconnect wires and hardware resources. Equation 5.3 [96] is used to formulate the dynamic power consumption where n is the total number of nodes, f is the clock frequency, V_{dd} is the supply voltage, C_i is the load capacitance for node n_i, and S_i is the switching activity for node n_i.

$$\text{Power}_{dynamic} = \frac{1}{2} f V_{\text{dd}}{}^2 \sum_{i=1}^{n} C_{\text{i}} S_{\text{i}} \tag{5.3}$$

For correct power estimation, a detailed placement and routing design is required to estimate the power consumption at each single node n_i in the design. In literature, three different approaches are used to estimate FPGA dynamic power consumption : characterization through board measurements [107] [45] [90], by using statistical model [56] [78] [89] [52] and by using simulation model [55] [37] [97]. In fact, obtaining a detailed

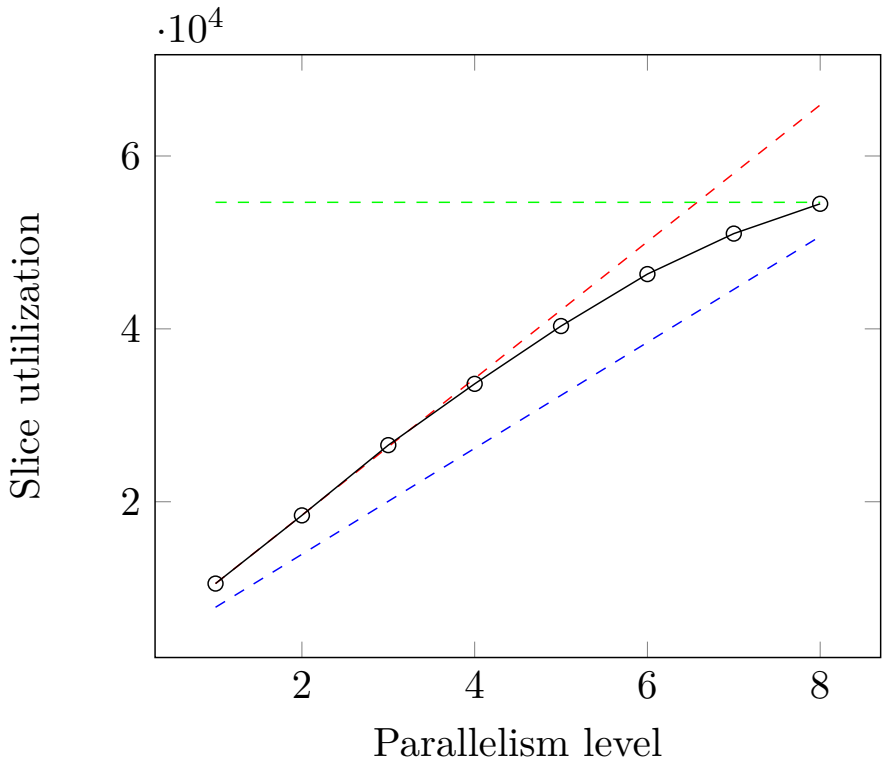

FIGURE 5.4 – Slice utilization estimated by method (1) --- and method (2) --- alongside the actual Slice values —o— for maximum Slice boundary --- at Slice = 54650

placement and routing design takes a considerable time which ranges between 30 - 90 min or even more for larger designs. During design space exploration, it is not practical to build a detailed design for every single point to estimate the power consumption. However, instead of that, quick power estimations are accepted at that early design stage to compare the power consumption of different design points. In this section, we will present an empirical power model based on hardware resources and operating frequency characterization.

5.4.1 Power Measurement

During our experiments, the power was measured through UCD90120A power controller mounted on Zynq-ZC706 board using TI Fusion Digital Power Designer software [98]. The power consumed by the FPGA chip was measured by monitoring rail 1 (VCCINT) of the power controller with a sample rate of 5 samples/s. Figure 5.5 shows how the sampled power varies over the time ; however by calculating the moving average, we noticed that the average measured power stabilized after 600 samples (i.e. after 120 seconds). For correct average power values, the power was sampled for at least 10 minutes on average.

The total power consumption is affected by how much hardware resources are used and at which frequency the design is operating. In order to formulate this relation, two basic

hardware blocks were designed where one uses only Slices while the other uses only BRAMs. Each hardware block was implemented in a separate design at different parallelism level operating at frequencies of values 50, 100, 150 and 200 MHz. We kept increasing the level of parallelism until full hardware utilization. For each design, the power was measured practically then it was plotted versus hardware utilization as depicted in Fig. 5.6. In that figure, we have two plots where the measured power is plotted versus BRAM and Slice respectively at four different frequencies 50 MHz —●—, 100 MHz —●—, 150 MHz —●— and 200 MHz —●—. We can observe from the plots that there is a correlation between the measured power and hardware utilization (Slice and BRAM). The plotted lines are not parallel to each other which reflect the interaction of frequency in the power equation. This correlation between the measured power, BRAM, Slice and frequency can be formulated by using regression analysis to obtain the power estimation model.

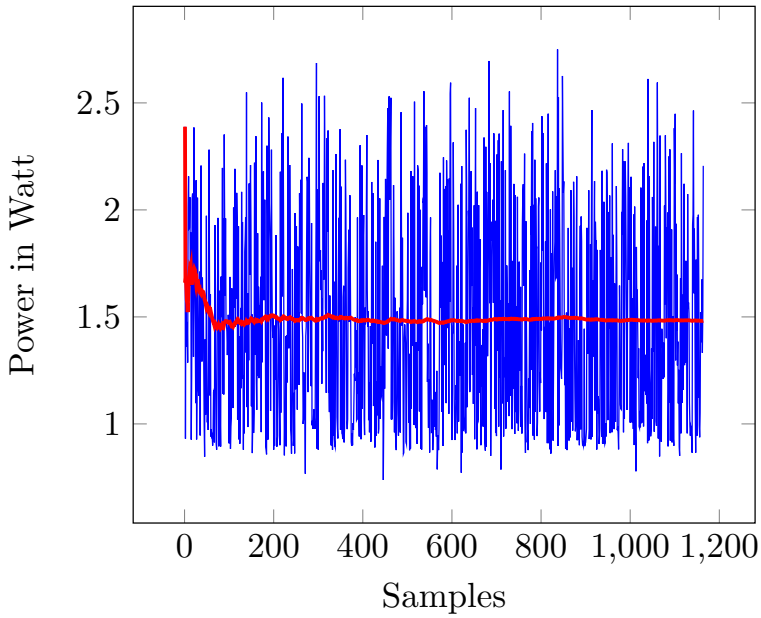

FIGURE 5.5 – Power —— is sampled every 200 ms while its moving average —— stabilizes after 600 samples

5.4.2 Power Regression Model

Table 5.5 lists the set of experiments conducted for building the power estimation model. For each experiment, we measured between 2700 - 3000 samples where all samples are arranged in one spreadsheet as an input for regression analysis to estimate the relationship between power, frequency, Slice and BRAM. Equation 5.4 describes the regression model between the dependent variable (power) and the independent variables (Slice, BRAM, frequency) where ξ

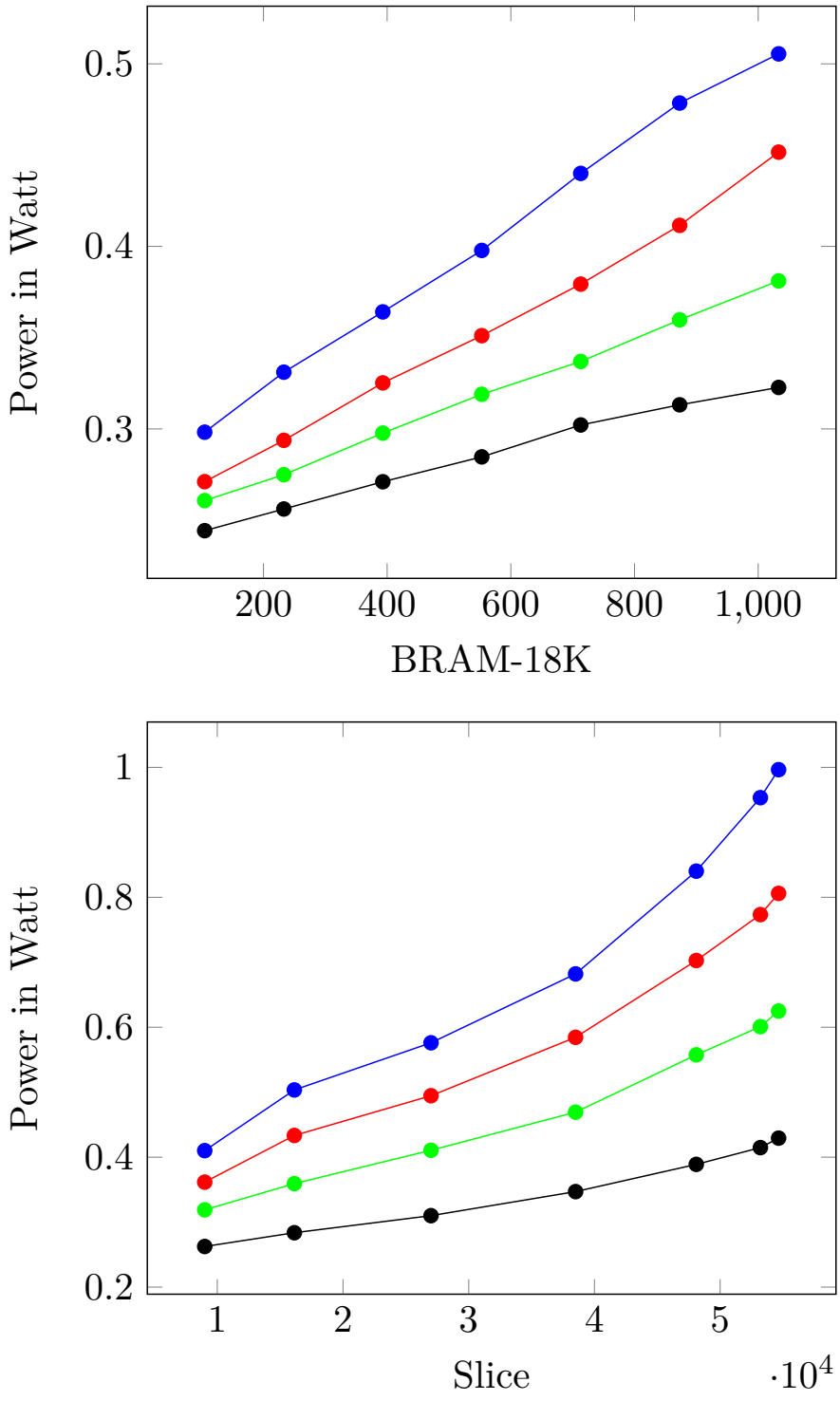

FIGURE 5.6 – Measured power with BRAM and Slice variation at frequencies 50 MHz —●—, 100 MHz —●—, 150 MHz —●— and 200 MHz —●—

Design	Frequency (MHz)	BRAM 18K	Slice	FF	LUT	Power in Watt (moving average)
1	50	96	8907	24737	23484	0.279
2	50	96	15503	41257	41006	0.310
3	50	156	25122	66959	68155	0.346
4	50	246	34083	93117	95716	0.383
5	50	366	46956	128006	132704	0.427
6	50	486	50216	146369	152893	0.442
7	50	606	54499	181253	189816	0.465
8	50	846	54124	176681	184439	0.504
9	50	1026	54506	187707	195683	0.515
10	100	96	8845	24737	23680	0.333
11	100	96	15219	41257	41204	0.373
12	100	156	24918	66959	68349	0.447
13	100	246	33739	93117	95908	0.522
14	100	366	46825	128006	132718	0.622
15	100	486	50300	146369	152882	0.657
16	100	606	54488	181253	189793	0.701
17	100	846	54097	176681	184455	0.720
18	100	1026	54476	187707	195709	0.744
19	150	96	8992	24737	23728	0.368
20	150	96	14989	41257	41248	0.431
21	150	156	24821	66959	68391	0.552
22	150	246	33948	93117	95946	0.651
23	150	366	46702	128006	132754	0.791
24	150	486	49885	146369	152907	0.847
25	150	606	54507	181253	189828	0.909
26	150	846	54038	176681	184490	0.937
27	150	1026	54453	187707	195780	0.971
28	200	96	8743	24737	23926	0.417
29	200	96	14797	41257	41459	0.508
30	200	156	23674	66959	68635	0.644
31	200	246	32905	93117	96246	0.788
32	200	366	44567	128006	133074	0.958
33	200	486	49834	146369	153273	1.048
34	200	606	54089	181253	190225	1.127
35	200	846	54037	176681	184909	1.186
36	200	1026	54492	187707	196224	1.208

TABLE 5.5 – Set of designs for building the power estimation model

represents the noise while equation 5.5 formulates the estimated power ($\hat{power}$) :

$$Power = f(Slice, BRAM, Frequency) + \xi \tag{5.4}$$

$$\hat{Power} = f(Slice, BRAM, Frequency) \tag{5.5}$$

There are various kinds of regression models to predict the power. In our study, we will compare three models (linear, pure-quadratic and full-quadratic) to choose the one which better fits. As previously shown in Fig. 5.6, the plotted power lines are almost linear with BRAM and Slice, but they do not have the same slope which indicates the interaction of frequency with power. The following equations are the equations for the three regression models (linear, pure-quadratic and full-quadratic) where β_i is the unknown regression coefficient :

- Linear regression model :

$$\hat{Power} = \beta_0 + \beta_1 * Slice + \beta_2 * BRAM + \beta_3 * Frequency \tag{5.6}$$

- Pure-quadratic regression model :

$$\begin{aligned} \hat{Power} = {} & \beta_0 + \beta_4 * Slice * BRAM + \beta_5 * Slice * Frequency \\ & + \beta_6 * BRAM * Frequency + \beta_7 * Slice^2 + \beta_8 * BRAM^2 \\ & + \beta_9 * Frequency^2 \end{aligned} \tag{5.7}$$

- Full-quadratic regression model :

$$\begin{aligned} \hat{Power} = {} & \beta_0 + \beta_1 * Slice + \beta_2 * BRAM + \beta_3 * Frequency + \beta_4 * Slice \\ & * BRAM + \beta_5 * Slice * Frequency + \beta_6 * BRAM * Frequency \\ & + \beta_7 * Slice^2 + \beta_8 * BRAM^2 + \beta_9 * Frequency^2 \end{aligned} \tag{5.8}$$

Before going into the details, it is preferable to explain some statistical definitions that will be used in the analysis for comparing the models.

- **Residual value**. It is the vertical distance between a data point and the regression line. They are positive if they are above the regression line and negative if they are below it, but if the regression line passes through the points, then the residual will be zero.
- **Residual Sum of Squares (Residual SS)**. It tells if the statistical model is a good fit for the data or not by calculating the overall difference between the data and their predicted values where $\Sigma\ error^2 = \Sigma\ (Power_{\text{actual}} - Power_{\text{predicted}})^2$. Smaller Residual SS means that the model better fits the data while greater values means the poorer

it fits them while a zero value means that the model is a perfect fit. Residual sum of Squares is used to calculate the coefficient of determination (R^2).

- **The coefficient of determination (R-squared)**. R^2 tells how many points fall on the regression line; in other words, R^2 represents the percentage of variability in the model. R-squared is explained by the fitted regression model by a value ranging between 0 to 1. For example, $R^2 = 0.8$ means that 80% of the variation of power-values fit with the regression model.

$$R^2 \quad = \quad 1 - \frac{Residual\ sum\ of\ Squares}{Total\ sum\ of\ Squares} \tag{5.9}$$

- **The correlation coefficient (Multiple-R)**. Multiple-R explains how strong the relationship between the dependent and independent variables is. For example, a value of 1 means a perfect positive relationship while a value of zero means no relationship at all. It is the square root of R-squared.
- **Adjusted R-squared**. It is adjusted for the number of coefficients in the model. This value is often used to compare models of different numbers of coefficients.
- **Null hypothesis**. It is the commonly accepted fact that the researchers work to nullify, reject or disprove it. It is named "null" because we try to nullify it, but it does not mean that the statement is null itself. For example, the null hypothesis for the coefficient β in the regression model equation is that $\beta = 0$.
- **Significance level (Alpha level α)**. It is the probability of making the wrong decision when the null hypothesis is true while the confidence level is defined as $(1 - \alpha)$.
- **P-value**. It is used in hypothesis test for either supports or rejects the null hypothesis. P-value is an evidence against the null hypothesis where the smaller p-value is, the stronger evidence to reject the null hypothesis. For confidence level = 98% (i.e. $\alpha = 0.02$), if P ≤ 0.02 then the null hypothesis is rejected; otherwise it will be accepted.
- **Significance-F**. It is the probability that the regression equation does not explain the variation in the dependent variable (power). If the Significance-F is not less than the confidence level then there is no meaningful correlation.

Table 5.6 lists the regression analysis for the three models (linear, pure-quadratic and full-quadratic). For the three models, Table 5.6 showed zero value for the Significance-F; therefore, the three models are valid (i.e. our results are statistically significant, and they likely did not happen by chance). Adjusted R-squared can be checked to tell us how many points fall on the regression line. It was 0.912166 for linear, 0.994009 for pure-quadratic and 0.99413 for full-quadratic. Apparently, the difference in Adjusted R^2 between pure-quadratic and full quadratic was not that big difference. For that reason, we can either choose the pure-quadratic to have fewer model parameters or to choose the full-quadratic to have the highest R^2; in our case, we chose the full-quadratic model. Finally, by applying a confidence level of 99% (i.e. $\alpha = 0.01$), we checked the corresponding p-values for the coefficients β of

	Linear model	**Pure-Quadratic model**	**Full-Quadratic model**
Regression SS	6384.396	6957.231	6958.076
Residual SS	614.766	41.93071	41.0856
Total SS	6999.162	6999.162	6999.162
Significance-F	0	0	0
Multiple R	0.955074	0.997	0.997061
R-square (R^2)	0.912166	0.994009	0.99413
Adjusted R^2	0.912163	0.994009	0.994129
Observations	100152	100152	100152
Intercept			
Coefficient β_0	-0.10964	0.251097	0.222776
P-value	0	0	0
Slice			
Coefficient β_1	7.88 e-06	XX	1.29 e-06
P-value	0	XX	3 e-161
BRAM			
Coefficient β_2	0.000151	XX	9.39 e-05
P-value	0	XX	8.81 e-31
Frequency			
Coefficient β_3	0.003124	XX	0.000116
P-value	0	XX	2.28 e-60
Slice * BRAM			
Coefficient β_4	XX	1.31 e-09	2.38 e-09
P-value	XX	5.24 e-59	1.19 e-54
Slice * Frequency			
Coefficient β_5	XX	7.3 e-08	7.03 e-08
P-value	XX	0	0
BRAM * Frequency			
Coefficient β_6	XX	4.29 e-07	5.51 e-07
P-value	XX	0	0
Slice ^{2}			
Coefficient β_7	XX	-1.2 e-11	-4.6 e-11
P-value	XX	6.73 e-91	0
BRAM ^{2}			
Coefficient β_8	XX	6.83 e-09	-9.5 e-08
P-value	XX	0.015541	1.3 e-147
Frequency ^{2}			
Coefficient β_9	XX	7.63 e-07	4.91 e-07
P-value	XX	0	9.91 e-81

TABLE 5.6 – Regression analysis for linear, pure-quadratic and full-quadratic power estimation model

the full-quadratic model ($\beta_0 \rightarrow \beta_9$) ; we could conclude that the null hypothesis is rejected for all coefficients ($\beta_0 \rightarrow \beta_9$) where p < 0.01. The full-quadratic power estimation model is described in equation 5.10 as follows :

$$\begin{aligned} \text{Power}|_{\text{Estimated}} \;=\; & 0.222776 \;+\; 1.29 \times 10^{-6} \times Slice \;+\; 9.39 \times 10^{-5} \times BRAM \;+\; 11.6 \times 10^{-5} \\ & \times Frequency \;+\; 2.38 \times 10^{-9} \times Slice \times BRAM \;+\; 7.03 \times 10^{-8} \times Slice \\ & \times Frequency \;+\; 5.51 \times 10^{-7} \times BRAM \times Frequency \;-\; 4.6 \times 10^{-11} \\ & \times Slice^2 \;-\; 9.5 \times 10^{-8} \times BRAM^2 \;+\; 4.91 \times 10^{-7} \times Frequency^2 \end{aligned} \tag{5.10}$$

We need to analyse the residual values to prove that the hypothesis behind the full-quadratic regression model holds. In the residual plot, the residuals are plotted on the vertical axis while the independent variable (power) is on the horizontal axis. If the points in the plot are randomly dispersed around the horizontal axis, then the regression model is appropriate for the data. In addition to that, both the sum and the mean average of the residual values should equal to zero. Figure 5.7 shows the residual plot for the full-quadratic regression model. It is obvious from the plot that the residual points are normally distributed around the horizontal axis. In addition to that, the sum of residuals and their mean average were 2.76245 e-09 and 2.75826 e-14 which were almost equal to zero. From this analysis, we can conclude that our full-quadratic model described in equation 5.10 is a valid regression model.

5.5 Performance Estimation

Two factors affect the execution time for video processing application : number of parallel processing channels and the operating frequency. As discussed in the previous chapter, it is common to divide the image into strips till one frame is completely processed. The execution time for strip processing is formulated in equation 5.11 which is the summation of clock cycles required to transfer the pixels from/to the processing element plus the clock cycles required for algorithm processing.

$$\begin{aligned} & Strip\ Processing\ (in\ cycles) \;= \\ & \qquad Cycles|_{\text{writing input pixels}} + Cycles|_{\text{PE Processing}} + Cycles|_{\text{reading output pixels}} \end{aligned} \tag{5.11}$$

Where Cycles$|_{\text{writing input pixels}}$ is the number of clock cycles required to transfer the pixels from the memory to the processing element through DMA communication, Cycles$|_{\text{PE Processing}}$ is the number of clock cycles required by the processing element for executing the application and Cycles$|_{\text{reading output pixels}}$ is the number of clock cycles required to transfer the processed pixels back to the memory. By using the following equation, we could know how many strips

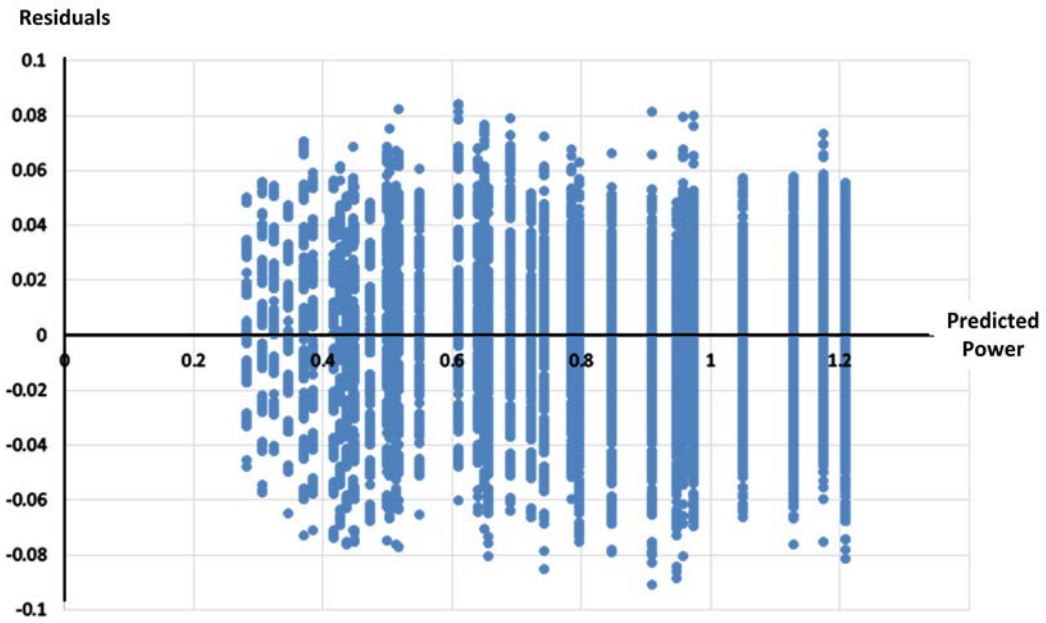

FIGURE 5.7 – High-level code generation design flow

are in one image frame :

$$Num\_of\_strips = \frac{Num\_image\_lines}{Num\_output\_lines|_{\text{parallelism level = 1}} * parallelism\ level} \quad (5.12)$$

Where Num_image_lines is the number of scanlines in one image and $Num\_output\_lines|_{\text{parallelism level = 1}}$ is the number of image lines produced by a single processing channel from one image strip processing. From the previous equations, the frame execution time can be calculated as follows :

$$Frame\ Execution\ Time\ (in\ seconds) = \frac{Num\_of\_strips * Strip\ Processing\ (in\ clk\ cycles)}{Frequency\ (in\ MHz)} \quad (5.13)$$

5.6 Automatic High-level Code Generation

5.6.1 Design Flow

Figure 5.8 shows that the high-level code generation tool has two input files which are : (1) **Processing Element C++ File** which implements the functionality of the video processing algorithm. (2) **System Specification File** which represents the system architecture constructed in Fig. 5.1. The *Specification File* has four main sections :

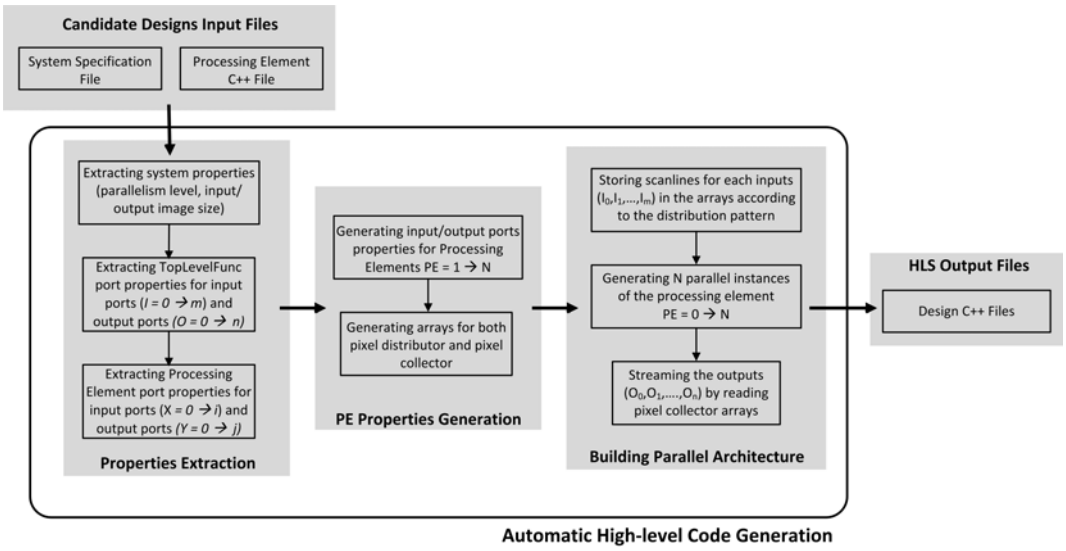

FIGURE 5.8 – High-level code generation design flow

- **Header section**. It states all header files and constants which are defined in the application.
- **System Properties section**. It defines the general system properties such as the size of the input/output images and what level of parallelism is realized.
- **Top Level Function section**. It defines the port properties for both system input ports (I_0, I_1, ..., I_m) and system output ports (O_0, O_1, ..., O_n). Port properties include its name, data type, how many scanlines are transferred from/to the system during execution (*num_of_scanlines*) and either if pixels are grouped during transfer to optimize bus communication or not (*num_of_merging_elements*). For example, if an 8-bit pixel is transferred over 64-bit bus width then 8 pixels can be merged and sent at once. For system interface, the tool implements AXI-Stream protocol for the input/output ports.
- **Processing Element section**. It defines the port properties of its input ports (X_0, X_1, ..., X_i) and output ports (Y_0, Y_1, ..., Y_j). These properties are its name, data type, the source of the input pixel stream (*src*) and the range of image scanlines which are mapped to that port during execution (*store_scanlines_from*, *store_scanlines_to*). In this section, we define only the parameters for the first processing element, then subsequently the tool can generate the port parameters automatically for the other processing elements by using the *shift_step* property. For example, if the first image scanline is mapped to the first processing element while the fifth one is mapped to the second processing core then the value of the *shift_step* property for that port is 4.

Figure 5.8 illustrates the three main phases to generate the high-level code for the parallel video processing architecture.

- **Properties extraction phase.** From *System Specification File*, the tool can extract *level_of_parallelism* and other input/output ports properties for both top-level and processing element blocks.
- **PE properties generation.** Based on the extracted properties, the tool can derive the properties of the other processing cores in the architecture (from PE = 1 to *N-1*) automatically. For both *pixel distributor* and *pixel collector*, arrays are created such that each input/output port ($X_{(i,PE)}$ or $Y_{(j,PE)}$) is mapped to a single array structure.
- **Building the parallel architecture.** Finally, the tool builds the parallel architecture by generating C++ code for : (1) Pixel distributor subroutine to store image scanlines in arrays according to the distribution pattern. (2) Instantiating a number of parallel processing instances equal to the level of parallelism. (3) Pixel collector subroutine to stream out the processed image scanlines. In addition to that, the tool manages how the pixels are separated before distribution or merged at the output ports in order to reduce bus communication time. After applying *HLS optimizations/User constraints*, the generated C++ design files are compiled by the High-level Synthesis tool to give the corresponding RTL design.

5.6.2 Code Generation

Listing 5.1 shows an example of the specification file for video downscaler (4 :1) for an input VGA image size. As described before, the specification file is subdivided into four main sections : *Header* section includes all header files and definitions (lines 1-12). *System Properties* section (lines 14-20) defines the size of the input/output image in addition to the level of parallelism implemented by the generated architecture (line 19). *Top-level* section (lines 22-35) defines the name of the top-level function *VideoDownScaler_parallel32* (line 23) and the port properties for the system input/output ports. In this application, there is one single input port *data_img* (lines 24-28) and one single output port *img_result* (lines 29-33). *Processing Element* section is the last section in the file (lines 37-53) where the number and the properties of the input/output ports for the processing element are defined.

Table 5.7 lists the number of Lines Of Code (LOC) generated by the tool for different applications at different levels of parallelism. LOC are calculated after excluding both blank lines and comments. To move from one parallelism level to another, we need only to change the value of *level_of_parallelism* parameter in *#System_Properties#* section. Consequently, a significant design time is saved by automating that step. For example, the size of system specification file for 5-window SAD algorithm is 98 lines where LOC ratio between the generated code to specification file is 3.2 (314 :98) for one processing element architecture, and it reaches to 74 (7244 :98) for an architecture containing 64 processing element. While

```
1  ## Header ##
2  #include "ap_int.h"
3  #define IMG_WIDTH               640
4  #define IMG_HEIGHT              480
5  #define IMG_SIZE             307200
6  #define IMG_WIDTH_2             320
7  #define IMG_HEIGHT_2            240
8  #define IMG_SIZE_4             76800
9  #define WIN_HEIGHT              2
10 #define STRIP_SIZE_PARA32_8        5120
11 #define IMG_WIDTH_2_PARA32_8     1280
12 ## ENDOF_Header ##
13
14 ## System_Properties ##
15 input_image.width   = 640
16 input_image.height  = 480
17 output_image.width  = 320
18 output_image.height = 240
19 Parallelism_Level    = 32
20 ## ENDOF_System_Properties ##
21
22 ## Top_Level_Function ##
23 Name = VideoDownScaler_parallel32
24 Num_of_inputs = 1
25 Input_0.name = data_img[STRIP_SIZE_PARA32_8]
26 Input_0.type = unsigned long long int
27 Input_0.num_of_scanlines = 64
28 Input_0.num_of_merging_elements = 8
29 Num_of_outputs = 1
30 Output_0.name = img_result[IMG_WIDTH_2_PARA32_8]
31 Output_0.type = unsigned long long int
32 Output_0.num_of_scanlines = 32
33 Output_0.num_of_merging_elements = 8
34 Interface = AXI-Stream
35 ## ENDOF_Top_Level_Function ##
36
37 ## Processing_Element ##
38 Name = VideoDownScaler
39 Num_of_inputs = 1
40 Input_0.name = image[IMG_WIDTH][WIN_HEIGHT]
41 Input_0.type = unsigned char
42 Input_0.src = data_img[STRIP_SIZE_PARA32_8]
43 Input_0.store_scanlines_from = 0
44 Input_0.store_scanlines_to = 1
45 Input_0.shift_step = 2
46 Num_of_outputs = 1
47 Output_0.name = image_result[IMG_WIDTH_2]
48 Output_0.type = unsigned char
49 Output_0.sink = img_result[IMG_WIDTH_2_PARA32_8]
50 Output_0.store_scanlines_from = 0
51 Output_0.store_scanlines_to = 0
52 Output_0.shift_step = 1
53 ## ENDOF_Processing_Element ##
```

Listing 5.1 – Specification file for video downscaler (4 :1) for input VGA image

level of Parallelism	Spec. File	1	4	8	16	32	64
5-win SAD	98	314	644	1084	1964	3724	7244
1-win SAD	74	249	552	904	1608	3016	5832
Video scaler	54	87	195	339	627	1203	2355
Conv. filter	52	69	195	363	634	1098	2026

TABLE 5.7 – Number of code lines generated for different applications at different parallelism level

for convolution filter, this ratio is 1.3 (69 :52) for one processing element and increases to 39 (2026 :52) for 64 elements architecture.

5.7 Experimental Results

By exploiting pipeline-level parallelism for 5-window SAD algorithm, three different designs were implemented. These designs are named *pipe4*, *pipe8* and *pipe12* where 4, 8, and 12 disparity lines are processed by the same processing channel. In this section, these initial three designs will be explored by varying the level of parallelism at different operating frequencies. By the help of ViPar tool, we will estimate the hardware utilization, power consumption and performance for each alternative in the design space as explained in the previous sections. According to the system constraints, only the candidate designs will be selected for synthesizing. The high-level codes for the candidate designs will be generated automatically then synthesized by the HLS tool to give the corresponding RTL design. The RTL design is then implemented and experimented to verify the estimated design metrics.

5.7.1 Area, Power and Performance Estimations

Resource utilization, power and frame execution time were estimated by means of the derived equations in the previous sections for the different designs listed in Table 5.8 and 5.9. The designs for *pipe4*, *pipe8* and *pipe12* at parallelism level = 1 operating at 100, 150 and 200 MHz are considered as the initial points for our estimation process (designs #1, #9, #17,#25, #29, #33, #37, #40 and #43). For area estimation, we keep increasing the level of parallelism till one of the resources either Slice or BRAM is completely utilized. The upper boundary for resources could differ from one case to another according to the used FPGA chip during the exploration process. For example in this exploration, Zynq ZC706 was used with maximum hardware resources of Slice = 54650, FF = 437200, LUT = 218600 and BRAM_18K = 1090.

Experimental measurements for power and performance were conducted in order to evaluate how far the estimations are correct from the real values. By default, *Default* synthesis/implementation strategies are used but if they failed to satisfy the timing

#	Freq. in MHz	level of Parall-elism	Slice (54650)	FF (437200)	LUT (218600)	BRAM (1090)	Power in mW	Frame Exec. Time in ms
1	100	1	10534	28903	31163	131	342.91	84.875
2		2	18444	49161	55683	243	418.29	44.85
3		3	26354	69419	80203	355	489.81	30.965
4		4	34264	89677	104723	467	557.46	24.838
5		5	42174	109935	129243	579	621.25	20.656
6		6	50084	130193	153763	691	681.17	17.854
7		7	44571	150451	178283	803	662.85	15.7
8		8	50701	170709	202803	915	710.87	14.32
9	150	1	10111	27410	31140	131	390.71	56.733
10		2	17754	46175	55636	243	494.28	29.975
11		3	25397	64940	80132	355	594.22	20.693
12		4	33040	83705	104623	467	690.54	16.596
13		5	40683	102470	129124	579	783.22	13.8
14		6	48326	121235	153620	691	872.28	11.927
15		7	44529	140000	178116	803	853.05	10.488
16		8	50653	158765	202612	915	925.64	9.565
17	200	1	9642	29895	31184	131	437.3	42.736
18		2	16893	51145	55723	243	565.37	22.574
19		3	24144	72395	80262	355	690.15	15.581
20		4	31395	93645	104801	467	811.62	12.494
21		5	38646	114895	129340	579	929.78	10.387
22		6	45897	136145	153879	691	1044.64	8.976
23		7	53148	157395	178418	803	1156.2	7.891
24		8	50740	178645	202957	915	1144.27	7.196

TABLE 5.8 – Estimations for utilization, power and frame execution time for *pipe4* designs

constraints, then they are replaced by *Performance Explore* strategies (*Performance Explore* strategies were used for designs #22, #35 and #44). However, some designs could not be synthesized even after changing the strategy due to the unsatisfied timing constraints or due to the lack of the hardware resources required to apply that new strategy (non-synthesized designs are #23, #24, #36 and #45).

Figure 5.9 depicts the percentage of estimation error for Slice, LUT and FF such that positive values mean overestimated values and negative values mean underestimated ones while the points of discontinuity in the plot are for the non-synthesized designs #23, #24, #36 and #45. The percentage estimation error ranges between -21% to 0.4%, -3.7% to 0.3% and -14.6% to 8% for LUT, FF and Slice respectively. The maximum estimation error for LUT occurred for design #22 by -21% and for design #44 by -15% due to the change of the implementation strategy (*Performance Explore* was used instead of the default strategies). For BRAM, the estimation error was not plotted since both the estimated and measured values

#		Freq. in MHz	level of Para-llelism	Slice (54650)	FF (437200)	LUT (218600)	BRAM (1090)	Power in mW	Frame Exec. Time in ms
25	Pipe8	100	1	15608	47992	48170	203	391.73	44.85
26			2	28745	87339	89700	387	510.4	24.838
27			3	41882	126686	131230	571	618.42	17.854
28			4	43190	166033	172760	755	649.25	14.32
29		150	1	16625	44339	48591	203	729.75	12.167
30			2	30357	80033	90533	387	650.15	16.596
31			3	44089	115727	132475	571	813.63	11.927
32			4	43605	151421	174417	755	837.69	9.565
33		200	1	16278	48644	53711	203	548.49	22.574
34			2	30001	88643	100770	387	777.02	12.494
35			3	43724	128642	147829	571	993.97	8.976
36			4	48722	168641	194888	755	1092.14	7.196
37	Pipe12	100	1	26073	67071	66804	275	475.8	30.965
38			2	49292	125497	126967	531	652.76	17.854
39			3	46783	183923	187130	787	674.48	12.774
40		150	1	24133	61240	69833	275	567.12	20.693
41			2	45392	113835	133017	531	818.27	11.927
42			3	49051	166430	196201	787	893.56	8.531
43		200	1	22127	67339	68978	275	646	15.581
44			2	41461	126033	131286	531	957.05	8.976
45			3	48399	184727	193594	787	1093.59	6.418

TABLE 5.9 – Estimations for utilization, power and frame execution time for *pipe8* and *pipe12* designs

were identical. Figure 5.10 shows the percentage error in the estimated frame execution time where it ranges between -10.4% to 4.3% for different designs. This error arose due to the time consumed to set the DMA communication between the Processing System (PS) and the Programmable Logic (PL).

For fast power estimations at high-level design, only information about frequency and resource utilization are available. Fig. 5.11 shows that the power consumption was underestimated by values range between 34% to 62.3% of the real measured values. It is reasonable to see that difference because some factors which contribute to the power consumption like switching activity, clock tree and the interconnect wires are not considered in the model equation. For further analysis, the estimated and measured power were plotted as depicted in Fig. 5.12 ; it is clear that the two curves behave in the same manner. In other words, the derived power model can be used for relative power comparison between alternative designs during the design space exploration process. However, it can not be used to estimate a value near from the real measurements for a single design due to the lack of full implementation design details.

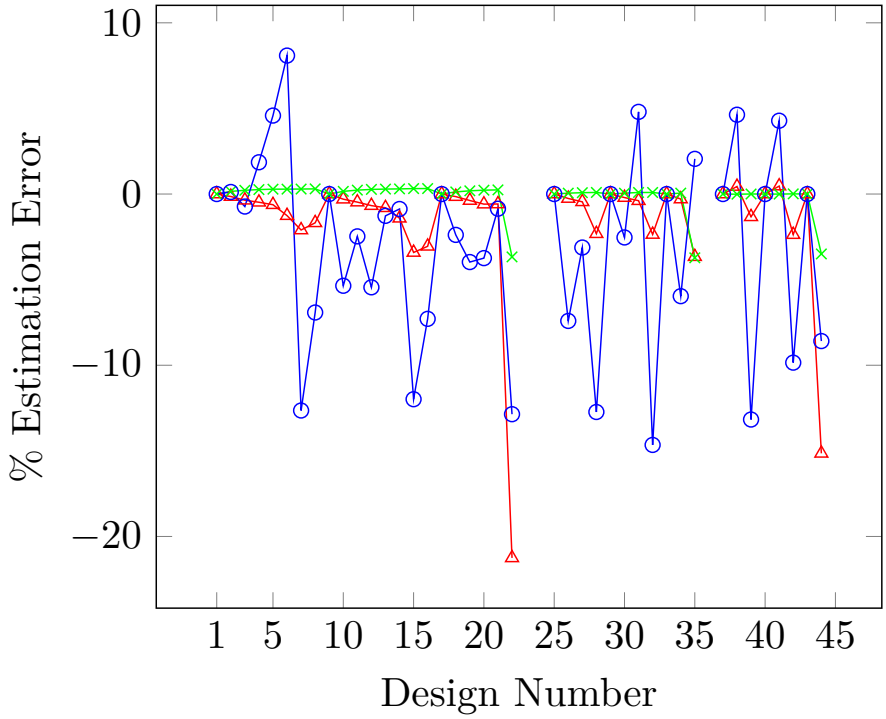

FIGURE 5.9 – The estimation percentage error for Slice —○—, LUT —△— and FF —×— when compared to the measured values for different designs

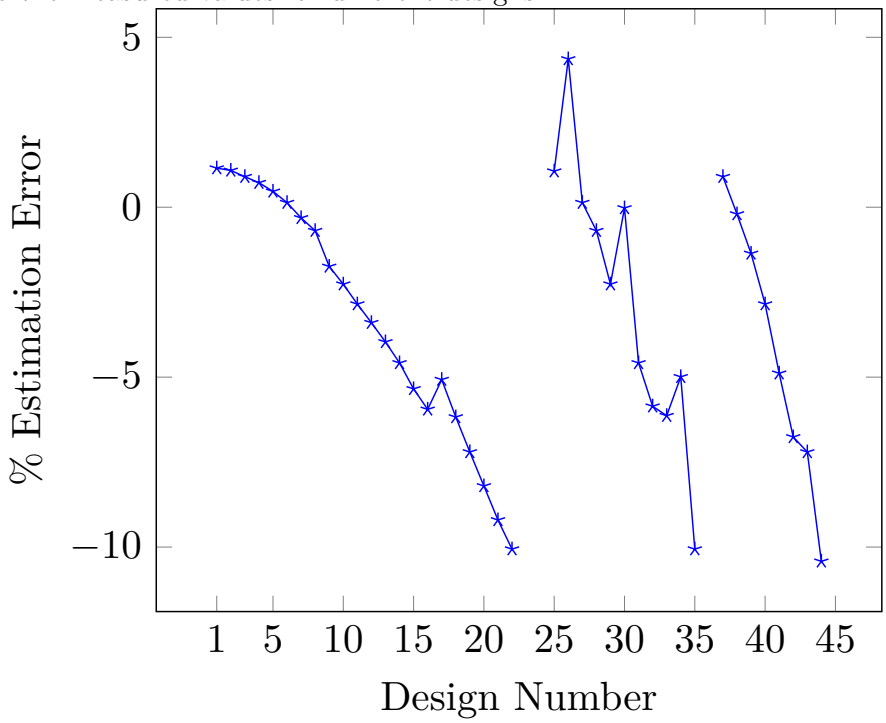

FIGURE 5.10 – The estimation percentage error for frame execution time when compared to the measured values

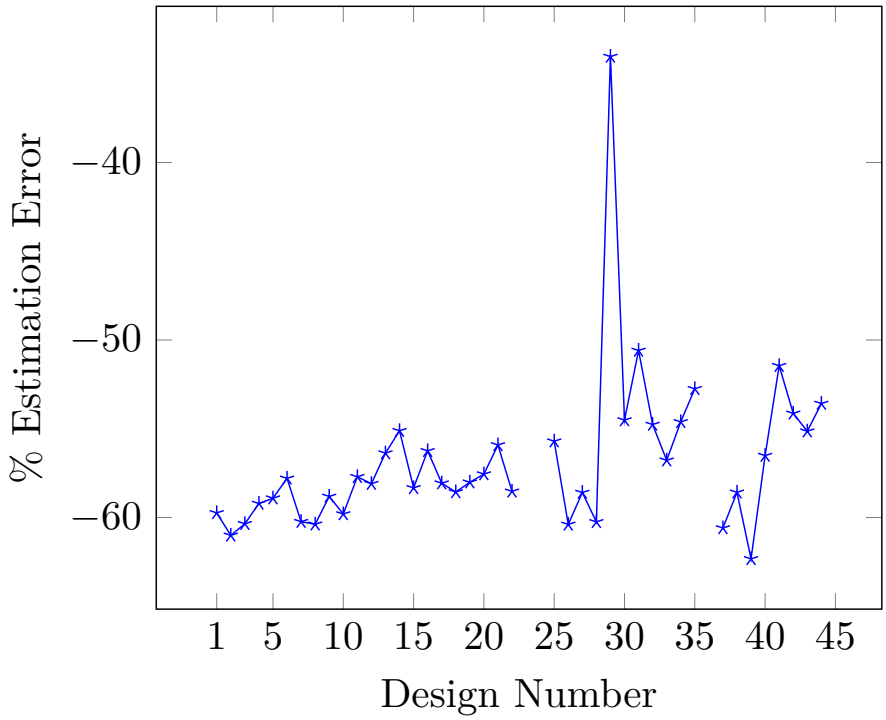

FIGURE 5.11 – The estimation percentage error for power consumption when compared to the measured values

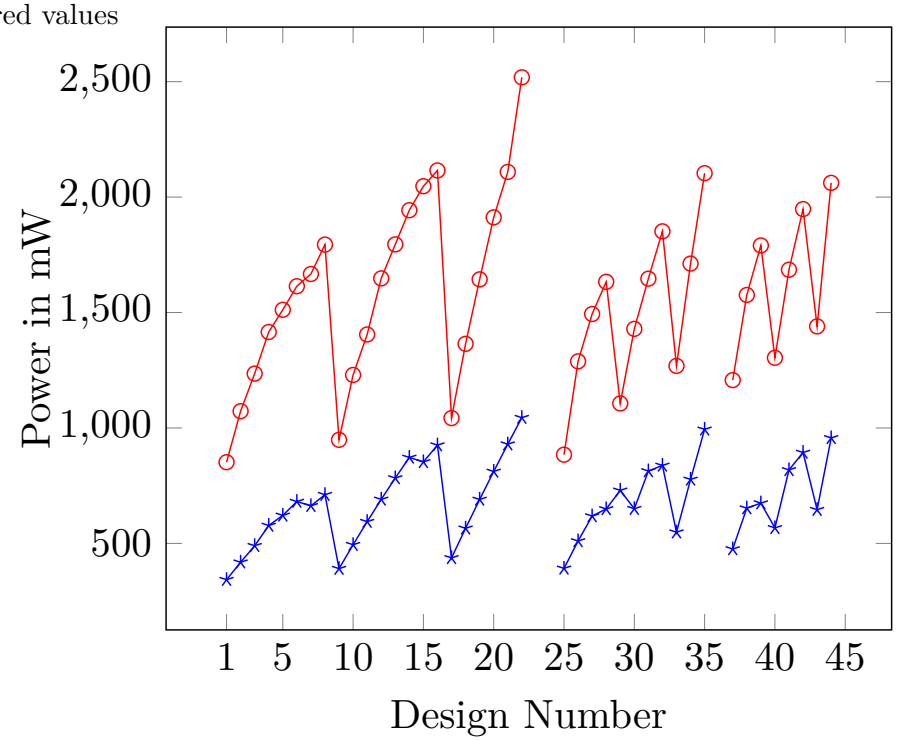

FIGURE 5.12 – Estimated —⋆— and measured —○— power for different designs

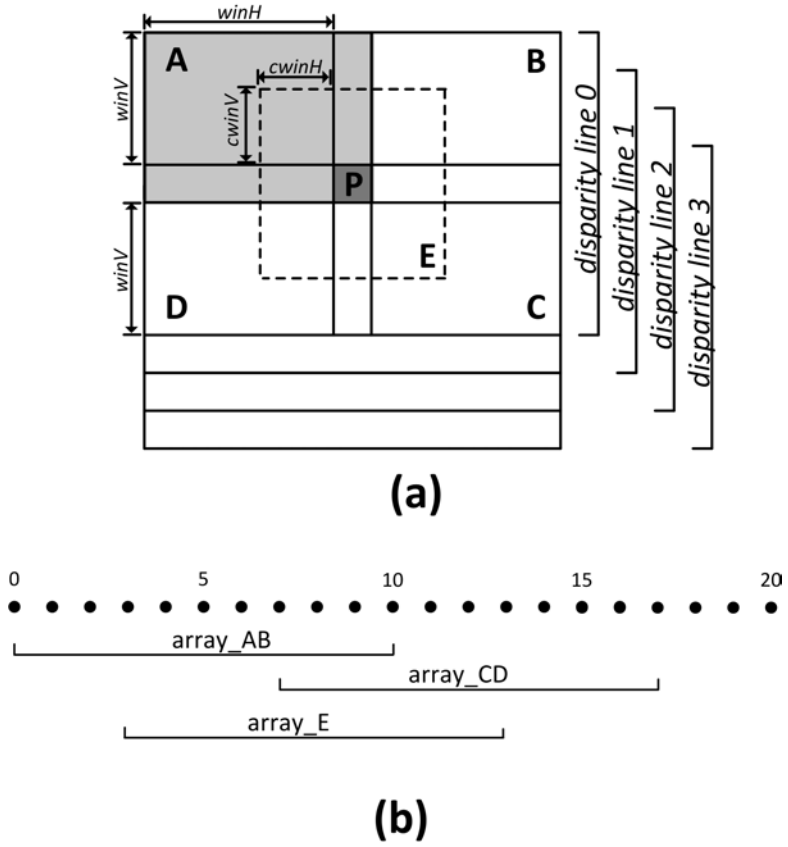

FIGURE 5.13 – (a) Enlarging the strip size to calculate 4 disparity lines (b) Image scanlines distribution pattern

5.7.2 High-level Code Generation

Figure 5.13a shows how four disparity lines can be calculated by enlarging the strip size (Remember that 5-win SAD configuration has the following parameters : winH =23, winV =7, cwinH =7, cwinV =3 and maximum disparity=64). Each strip consists of 18 image lines where image lines from 0 to 10, from 7 to 17 and from 4 to 13 are used to calculate the sum of windows A/B, C/D and E respectively as depicted in Fig. 5.13b.

Listing 5.2 lists the *Processing_Element* section of the specification file where each processing channel has 6 input ports such that each port is linked to a separate array structure (*data_R_AB*, *data_L_AB*, *data_R_CD*, *data_L_CD*, *data_R_E* and *data_L_E*) (Lines 4, 10, 16, 22, 28 and 34). We defined 6 properties for each input port : *name*, *type* and *src* to define its name, type and the top-level input source linked to that port. *store_scanlines_from* and *store_scanlines_to* are used to define the distribution pattern of the image lines. For example, Fig. 5.13b depicts that calculating the summation of windows C/D requires to store the image lines from 7 to 17 in arrays *data_R_CD* and *data_L_CD* (Lines 19-20 and 25-26).

```
## Processing_Element ##
Name = MultiWinSAD_pipe4
Num_of_inputs = 6
Input_0.name = data_R_AB[LINE_SIZE][STRIP_HEIGHT_ABCD_PIPE4]
Input_0.type = ap_int<18>
Input_0.src = data_img_R[STRIP_SIZE_PIPE4]
Input_0.store_scanlines_from = 0
Input_0.store_scanlines_to = 10
Input_0.shift_step = 4
Input_1.name = data_L_AB[LINE_SIZE][STRIP_HEIGHT_ABCD_PIPE4]
Input_1.type = ap_int<18>
Input_1.src = data_img_L[STRIP_SIZE_PIPE4]
Input_1.store_scanlines_from = 0
Input_1.store_scanlines_to = 10
Input_1.shift_step = 4
Input_2.name = data_R_CD[LINE_SIZE][STRIP_HEIGHT_ABCD_PIPE4]
Input_2.type = ap_int<18>
Input_2.src = data_img_R[STRIP_SIZE_PIPE4]
Input_2.store_scanlines_from = 7
Input_2.store_scanlines_to = 17
Input_2.shift_step = 4
Input_3.name = data_L_CD[LINE_SIZE][STRIP_HEIGHT_ABCD_PIPE4]
Input_3.type = ap_int<18>
Input_3.src = data_img_L[STRIP_SIZE_PIPE4]
Input_3.store_scanlines_from = 7
Input_3.store_scanlines_to = 17
Input_3.shift_step = 4
Input_4.name = data_R_E[LINE_SIZE][STRIP_HEIGHT_E_PIPE4]
Input_4.type = ap_int<18>
Input_4.src = data_img_R[STRIP_SIZE_PIPE4]
Input_4.store_scanlines_from = 4
Input_4.store_scanlines_to = 13
Input_4.shift_step = 4
Input_5.name = data_L_E[LINE_SIZE][STRIP_HEIGHT_E_PIPE4]
Input_5.type = ap_int<18>
Input_5.src = data_img_L[STRIP_SIZE_PIPE4]
Input_5.store_scanlines_from = 4
Input_5.store_scanlines_to = 13
Input_5.shift_step = 4
Num_of_outputs = 4
Output_0.name = best_disparity_0[LINE_SIZE]
Output_0.type = unsigned char
Output_0.sink = disp_img_result[LINE_SIZEx4]
Output_0.store_scanlines_from = 0
Output_0.store_scanlines_to = 0
Output_0.shift_step = 4
Output_1.name = best_disparity_1[LINE_SIZE]
Output_1.type = unsigned char
Output_1.sink = disp_img_result[LINE_SIZEx4]
Output_1.store_scanlines_from = 1
Output_1.store_scanlines_to = 1
Output_1.shift_step = 4
Output_2.name = best_disparity_2[LINE_SIZE]
Output_2.type = unsigned char
Output_2.sink = disp_img_result[LINE_SIZEx4]
Output_2.store_scanlines_from = 2
Output_2.store_scanlines_to = 2
Output_2.shift_step = 4
Output_3.name = best_disparity_3[LINE_SIZE]
Output_3.type = unsigned char
Output_3.sink = disp_img_result[LINE_SIZEx4]
Output_3.store_scanlines_from = 3
Output_3.store_scanlines_to = 3
Output_3.shift_step = 4
## ENDOF_Processing_Element ##
```

Listing 5.2 – Specification file for calculating 4 disparity lines for 5-win SAD algorithm (Processing_Element section)

shift_step is used to determine from which image line will start the next strip processing for the other processing channels (Line 9). To generate the high-level code for another level of parallelism ; simply, we do the following modifications : (1) Updating the defined constants for the size of the input/output pixels (Listing 5.3, Lines 3 and 4). (2) Updating the value of the level of parallelism (Line 12). (3) Updating the number of image lines write/read to/from the architecture (Lines 20, 24 and 29). While Listing 5.2 for the processing element is kept unchanged. In the same way, we will generate the high-level code for other design choices *pipe8* and *pipe12*.

```
1  ## Header ##
2  #include "functions.h"
3  #define STRIP_SIZE_PIPE4      5760
4  #define LINE_SIZEx4           320
5  ## ENDOF_Header ##
6
7  ## System_Properties ##
8  input_image.width   = 640
9  input_image.height  = 480
10 output_image.width  = 640
11 output_image.height = 480
12 Parallelism_Level = 1
13 ## ENDOF_System_Properties ##
14
15 ## Top_Level_Function ##
16 Name = D04_pipe4_parallel1
17 Num_of_inputs = 2
18 Input_0.name = data_img_R[STRIP_SIZE_PIPE4]
19 Input_0.type = unsigned long long int
20 Input_0.num_of_scanlines = 18
21 Input_0.num_of_merging_elements = 2
22 Input_1.name = data_img_L[STRIP_SIZE_PIPE4]
23 Input_1.type = unsigned long long int
24 Input_1.num_of_scanlines = 18
25 Input_1.num_of_merging_elements = 2
26 Num_of_outputs = 1
27 Output_0.name = disp_img_result[LINE_SIZEx4]
28 Output_0.type = unsigned long long int
29 Output_0.num_of_scanlines = 4
30 Output_0.num_of_merging_elements = 8
31 Interface = AXI-Stream
32 ## ENDOF_Top_Level_Function ##
```

Listing 5.3 – Specification file for calculating 4 disparity lines for 5-win SAD algorithm

5.7.3 Design Space Exploration

All design variations listed in Tables 5.8 and 5.9 could be accepted as a solution but the applied system constraints will direct our final decision to choose one design among the others. Fig. 5.14 depicts some of the candidate designs (#7, #31, #42 and #43) along with the system constraints to guide the designer towards an efficient solution. The orange shaded

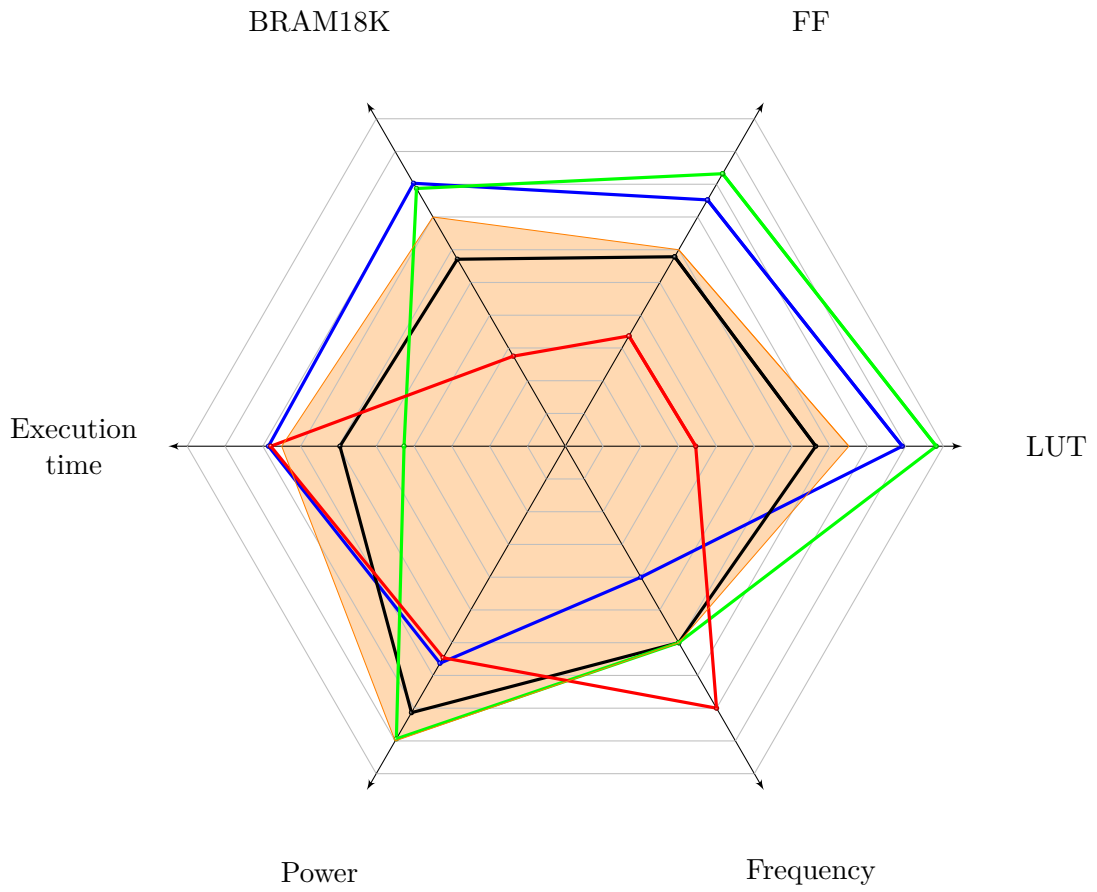

FIGURE 5.14 – Radar chart for designs #7 , #31 , #42 , #43 and system constraints

area represents the system constraints defined by the designer which are : frame execution time $\leq$ 15 ms, LUT $\leq$ 150000, FF $\leq$ 120000, BRAM $\leq$ 700 and frequency $\leq$ 150 MHz. From Fig. 5.14, we could deduce that design #31 succeeded to satisfy all the system constraints (for design #31, LUT = 132475 , FF = 115727 , BRAM = 571 , frequency = 150 MHz and frame execution time = 12 ms). Design #43 had relatively less hardware utilization (LUT = 68978 , FF = 67339 , BRAM = 275) and acceptable execution time (15.6 ms) in compare with design #31 ; however, it failed to meet the frequency constraint. In such case, the designer can think either to change the system constraints to profit from the less hardware utilization or to be stuck with them.

5.8 Conclusion

In this chapter, we presented ViPar tool which explores the design space for the best candidate parallel architectures. To compare different design alternatives in the design space,

we derived equations to estimate each of area utilization, performance and power consumption. The experiments showed that the percentage estimation error for area utilization was between -3.5% to 0.4% for LUT, -0.1% to 0.3% for FF, -14.6% to 8% for Slice and zero percentage for BRAM. For frame execution time, the percentage estimation error was between -10.4% to 4.3%. While for power consumption, the percentage estimation error was significantly high ranging between 34% to 62% of the real measured values. The reason behind that error is due to some key affecting factors in the power consumption like switching activity, the number of interconnecting wires and clock tree were not taken into consideration in the model equation. We showed that the estimation power model fitted for relative power comparison between alternative designs rather than using it to estimate the power consumption of a single design. Finally, we demonstrated an example of design space exploration problem where the estimation equations guided the designer towards the efficient solution under certain system constraints.

Chapter 6

Conclusion and Perspectives

6.1 Conclusions

In this chapter, we will summarize the main contributions achieved in this work, and we will put our work in perspective with suggestions for future works. This thesis emerged from a real industrial problem in the domain of autonomous vehicle. Multiple image sensors are usually installed in autonomous vehicles for static/dynamic obstacles detection, tracking and classification. Although there are other sensors like lidar and radar which can be used for achieving the same task ; but each solution has its pros and cons. Accordingly, it is preferable to integrate those different sensors (camera, lidar and radar) in the same system to assure the highest operation safety conditions. For image sensors, there is a continuous demand for increasing the frame rate and for enhancing the image resolution for better obstacle detection. Moreover, the images should be processed under real-time constraints. Different challenges arose while solving that problem. We can mention them in the following points : First, image/video applications are good candidates for parallel processing. Accordingly, parallel architectures should be considered while trying to find the answers for the questions like how parallelism level can be calculated ? how the input pixel stream can be distributed over the parallel processing elements to guarantee the maximum performance. Second, the process of building parallel video architectures should be automated to increase the design productivity and to decrease the production costs while respecting the constraints for time-to-market.

In this thesis work, we chose FPGA technology as a platform for our proposed solution for the following main reasons : First, FPGAs are reconfigurable hardware that can be reprogrammed to build massively parallel architectures for video processing applications. Second, compared to other technologies, FPGA offer an excellent tradeoff between computing rate and power consumption at reasonable production costs, which suits for battery-based systems. In the light of the aforementioned challenges, our contributions were presented as follows :

In Chapter 3, we presented a generic model for pixel distribution/collection for parallel streaming video architectures. In this model, first, we defined the required parameters to allow any size processing window to slide freely in the horizontal and vertical direction. Second, the hardware architecture for the pixel distributor/collector was generated automatically to minimize the design efforts. In the experimental results, we showed the hardware implementation of parallel architectures used to process two different video streaming applications : video downscaler (16 :1) and convolution filter of kernel=3x3. In addition to that, we used hardware parallelism to reduce the power consumption by scaling down the operating frequency on the parallel processing elements. We derived the equations used to calculate the depth of FIFOs for two different cases either when all processing elements were activated or when they were not yet activated. We derived as well the equation for calculating the required level of parallelism to maintain the same processing rate. The variation in the parallelism level, depth of FIFO and frequency, formed a set of different design alternatives

that could be considered as a solution. The designer could select one design rather than another according to the design constraints in terms of power consumption and hardware utilization. In the experimental results, we showed the implementation for two applications : video downscaler (16 :1) and AES-encrypted HD image with maximum power reduction of 19.6% and 5.4% respectively. This variation in power reduction in both applications was due to the different power breakdown (for video downscaler, 53% of power was consumed by BRAM while for AES encryption, 52% of power was consumed by logic and signals).

In Chapter 4, Multi-window Sum of Absolute Difference stereo matching (Multi-window SAD) was proposed by our industrial partner NAVYA for hardware implementation. We used high-level synthesis (HLS) tools for implementation mainly to decrease the developing time and to easily evaluate different architectural configurations. We used a set of HLS optimization techniques to efficiently implement the corresponding hardware architecture. For each optimization step, we showed its impact on the overall design quality concerning hardware cost and execution time. In the experimental results, we explained how to build the hardware part of our application as well as the software part. Different architectures could be evaluated by varying the parallelism level, the operating frequency (100, 150 or 200 MHz) and the processing pipeline (*pipe_4*, *pipe_8* or *pipe_12*) to form a design space of different alternatives in terms of hardware utilization, frame rate and power consumption.

In Chapter 5, we developed ViPar as a tool for design space exploration. In ViPar, we introduced the equations needed for estimating the hardware utilization and frame execution time for different alternatives in the design space. Experimental power measurements were done to build an empirical regression model for power estimation based on three independent variables (Slice, BRAM and frequency). We compared statistically three regression models (linear, pure-quadratic and full-quadratic) to select the model which better described the consumed power. For Multi-window SAD stereo application, we explored its design space for the best candidates by comparing different designs regarding hardware resources (LUT, FF and BRAM), frame rate, frequency and power consumption. Finally, for the candidate designs, we described their parallel hardware architecture in the specification file then ViPar was used to generate the corresponding architecture automatically for synthesis and experimental evaluation. In the experimental results, the estimated values for area utilization, execution time and power consumption for Multi-window SAD application were compared to the experimentally measured values for evaluation. The percentage error in area estimation was in the range of -14% to 8% for Slice, -3.7% to 0.3% for FF and -3.6% to 0.4% for LUT where underestimations were denoted by negative values and overestimations were indicated by positive values. While for BRAM, estimated and measured values were identical. For frame rate estimations, the percentage error was between -10.4% and 4.3% for different design points. The power was underestimated by our empirical model by a percentage error reached to 62%. This power underestimation was due to considering only the hardware resources

and frequency for building the power estimation model. While other factors which could participate in power consumption like interconnection wires, switching activity or clock tree are neglected because they are calculated from full synthesis process. In that case, running full implementation would oppose our goal of doing fast power estimations at high design levels. Therefore, our estimation power model is not targeting single design power estimation, but it fits for inter-designs power comparison which is our case while exploring the design space.

6.2 Perspectives

Multi-application design space exploration. In this thesis, we considered the case where a single application task was mapped and parallelized on a single FPGA. In autonomous vehicle domain, it is normal to have multiple tasks running concurrently to form a long processing channel. For example, in stereo matching applications, camera calibration and image filter algorithms can be executed to enhance the quality of the preprocessed image. While other algorithms for object classification and tracking can be processed after knowing the depth of objects by the stereo matching application. In this situation, we will have multi-application multi-objective design space exploration problem where we will search for a feasible solution that satisfies the global system constraints in terms of performance, area utilization and power consumption.

The first steps towards a feasible solution are by profiling those applications to figure out the bottleneck computation tasks to be firstly considered for hardware acceleration as well as to avoid accelerating functions which could be executed at acceptable performance on General Purpose Processors. The communication protocol between application tasks can be in different forms like point-to-point AXI Stream, NoC-based communication, etc. Each communication method will have its impact on the performance and on how the data can flow from one node to another. While searching for an efficient solution, our design space will grow tremendously when the aforementioned factors are included ; for that reason, searching algorithms (exact or heuristic) are advised for solving that problem.

Previously, our research team worked to solve an application mapping problem on a heterogeneous platform (CPU + FPGA) using Mixed Integer Programming (MIP) method [9]. In our case, multi-application design space exploration will be done in two steps. First, we explore how the tasks are mapped either to CPU or FPGA. Second, we explore the design parameters concerning parallelism level, operating frequency, etc, to show how the FPGA-mapped tasks are implemented to satisfy the system constraints. Heuristic searching algorithms are preferred in that case rather than using exact methods to find the feasible task mapping within a short time in term of seconds or minutes.

Self-adaptivity. It is the ability of the system to adapt itself due to external changes. Self-adaptivity is done by monitoring the environment then adapt its behaviour in order to

preserve or improve the operation of the system according to some previously defined criteria. The quality of image processing results can be affected by the environmental conditions (day/night, foggy/clear/raining, illumination intensity, etc). For autonomous vehicles, the input image can be preprocessed by an additional filter to guarantee the same quality of results. In that case, there is no need to have all the image filters mapped and running at the same time but it is recommended to apply the correct filter in the correct corresponding environmental situation. According to that, the system will not be only autonomous but also smart by adapting to the external influences. FPGAs are promoted to implement self-adaptive systems because they are dynamic and partial reconfigurable architectures [16]. We can profit from self-adaptivity to avoid system failures and consequently, increase the safety conditions in the autonomous vehicles. For example, if one sensor failed during the operation time then the system could use self-adaptivity to take some actions, for instance : (i) to reconfigure its architecture to isolate the failed sensor. (ii) to adapt the decision-making behaviour by neglecting the data coming from the failed sensor.

Three-step process should be followed to have a self-adaptive system : monitoring, analyzing and decision making. At runtime different parameters can be monitored like performance, power, etc, for behaviour analysis then decisions can be taken upon that. Task mapping decisions in a self-adaptive system can have two different scenarios : In the first scenario, we consider only the predefined mapping scenarios which are tested and validated for use in critical safety application like autonomous vehicles. While in the second scenario, task mapping is done at runtime by using heuristic algorithms for fast decision making. Industrial cases will help us to build real scenarios for self-adaptivity. In those scenarios, we have to define if those algorithms are running simultaneously or mutually exclusive then we have to figure out how they are mapped to the reconfigurable platform without violating the main system constraints in terms of performance, power consumption and area utilization.

System scalability. In autonomous vehicles, several functions are integrated for detecting traffic light signs, lanes, pedestrians, etc. Accelerating all these algorithms by using a single FPGA may be infeasible and accordingly, migrating to a multiple-FPGA platform is advisable. Several vendors introduce multi-FPGA boards to address the system requirements for the most demanding high-performance applications. We can list some examples for multi-FPGA boards which exist today in the market like HTG-847 and HTG-747 from HiTechGlobal [3], Merrick FPGA series from Enterpoint [2], Prodigy multi-FPGA from S2C [7], quad Virtex UltraScale and quad Virtex 7 multi-FPGA boards from PRO DESIGN Electronic [6]. Some of these platforms are customizable according to the system requirements like PicoComputing multi-FPGA board [4] in terms of the number of FPGA nodes and the type of each FPGA device. The problem of flexible scalable multi-FPGA was previously tackled in our team [103]. In that work, all the communication lanes between FPGA modules go via PCIe switch to keep the wire overhead per FPGA module constant. During multi-FPGA design space exploration,

first, we have to explore which tasks are mapped to which FPGA then we have to explore the different alternatives to implement this task on the selected FPGA concerning parallelism level, frequency and resource utilization. We will also search for efficient solutions for some other questions like : can the same application be mapped to two different FPGAs ? In case of failure, how can an application be migrated from one FPGA to another ?

Global processing model for autonomous vehicles. Different types of sensors are integrated into autonomous shuttles. Smart sensors are usually used where most of the processing is done locally at the sensor node, and only the results are communicated to the central unit for decision making. Defining a global processing model for a system integrating several of sensors is a challenging task. In such kind of systems, we have to define how decisions are made because sometimes the captured data by different sensors are redundant and sometimes they are complementary. For that reason, multi-sensor data fusion is required for correct decision making [70]. Using deep learning [54] in video/image processing applications for obstacle detection, tracking and classification, is very promising in the domain of autonomous vehicles [17]. The processing model will search for answers to questions like : How data can be fused from different sensors to take certain decisions like stopping in front of obstacles ? How will the system behave when one sensor is failed ? And how the system recovery will be done to maintain the same safety conditions ? How will the sensors act when the central unit failed ? Will they exchange data in a non-centralized way ? or will they elect a new node as the central unit ? And based on what conditions will be this election ?

References

[1] Xilinx Ultrascale+ Zynq MPSoC platform. https://www.xilinx.com/products/silicon-devices/soc/zynq-ultrascale-mpsoc.html.

[2] Enterpoint Ltd. https://www.enterpoint.co.uk/products/spartan-6-development-boards/merrick-6/.

[3] HiTech Global. http://www.hitechglobal.com/boards/allboards.htm.

[4] Micron Technology. https://www.micron.com.

[5] NAVYA company. http://navya.tech/en/.

[6] PRO DESIGN Electronic GmbH. http://www.prodesign-europe.com/proFPGA_Products.html.

[7] S2C Inc. http://www.s2cinc.com/design-classification/multi-fpga-prototyping.

[8] N. Abel, S. Manz, F. Grull, and U. Kebschull. Increasing Design Changeability Using Dynamic Partial Reconfiguration. *IEEE Transactions on Nuclear Science*, 57(2) :602–609, April 2010.

[9] A. Ait El Cadi, O. Souissi, R. Ben Atitallah, N. Belanger, and A. Artiba. Mathematical programming models for scheduling in a CPU/FPGA architecture with heterogeneous communication delays. *Journal of Intelligent Manufacturing*, Apr 2015.

[10] M. Al Kadi, B. Janssen, and M. Huebner. FGPU : An SIMT-Architecture for FPGAs. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 254–263, New York, NY, USA, 2016. ACM.

[11] G. Amato, F. Carrara, F. Falchi, C. Gennaro, and C. Vairo. Car parking occupancy detection using smart camera networks and Deep Learning. In *2016 IEEE Symposium on Computers and Communication (ISCC)*, pages 1212–1217, June 2016.

[12] M. Amiri, F. M. Siddiqui, C. Kelly, R. Woods, K. Rafferty, and B. Bardak. FPGA-Based Soft-Core Processors for Image Processing Applications. *Journal of Signal Processing Systems*, 87(1) :139–156, Apr 2017.

[13] K. Andryc, M. Merchant, and R. Tessier. FlexGrip : A soft GPGPU for FPGAs. In *2013 International Conference on Field-Programmable Technology (FPT)*, pages 230–237, Dec 2013.

[14] O. Arcas-Abella, G. Ndu, N. Sonmez, M. Ghasempour, A. Armejach, J. Navaridas, W. Song, J. Mawer, A. Cristal, and M. Luján. An empirical evaluation of high-level synthesis languages and tools for database acceleration. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[15] Avent. *FMC-IMAGEON EDK Reference Design Tutorial.*

[16] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka. Dynamic and Partial FPGA Exploitation. *Proceedings of the IEEE*, 95(2) :438–452, Feb 2007.

[17] K. Behrendt and J. Witt. Deep learning lane marker segmentation from automatically generated labels. In *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 777–782, Sept 2017.

[18] Y. Bi, C. Li, and F. Yang. Very High Level Synthesis for Image Processing Applications. In *Proceedings of the 10th International Conference on Distributed Smart Camera*, ICDSC '16, pages 160–165, New York, NY, USA, 2016. ACM.

[19] M. Bramberger, A. Doblander, A. Maier, B. Rinner, and H. Schwabach. Distributed Embedded Smart Cameras for Surveillance Applications. *Computer*, 39(2) :68–75, Feb 2006.

[20] V. Brost, F. Yang, and C. Meunier. Flexible VLIW processor based on FPGA for efficient embedded real-time image processing. *Journal of Real-Time Image Processing*, 9(1) :47–59, Mar 2014.

[21] N. Calagar, S. D. Brown, and J. H. Anderson. Source-level debugging for FPGA high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–8, Sept 2014.

[22] A. Canis, J. Choi, M. Aldham, V. Zhang, A. Kammoona, T. Czajkowski, S. D. Brown, and J. H. Anderson. Legup : An open-source high-level synthesis tool for fpga-based processor/accelerator systems. *ACM Trans. Embed. Comput. Syst.*, 13(2) :24 :1–24 :27, Sept. 2013.

[23] J. Chiang and S. Zammattio. *Five Ways to Build Flexibility into Industrial Applications with FPGAs.* Intel FPGA.

[24] A. Cilardo and L. Gallo. Interplay of loop unrolling and multidimensional memory partitioning in hls. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 163–168, March 2015.

[25] J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang. High-Level Synthesis for FPGAs : From Prototyping to Deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 30(4) :473–491, April 2011.

[26] Y. Cong, J. Yuan, and Y. Tang. Video Anomaly Search in Crowded Scenes via Spatio-Temporal Motion Context. *IEEE Transactions on Information Forensics and Security*, 8(10) :1590–1599, Oct 2013.

[27] P. Cooke, J. Fowers, G. Brown, and G. Stitt. A Tradeoff Analysis of FPGAs, GPUs, and Multicores for Sliding-Window Applications. *ACM Trans. Reconfigurable Technol. Syst.*, 8(1) :2 :1–2 :24, Mar. 2015.

[28] P. Coussy, D. D. Gajski, M. Meredith, and A. Takach. An introduction to high-level synthesis. *IEEE Design Test of Computers*, 26(4) :8–17, July 2009.

[29] K. C. Dey, A. Mishra, and M. Chowdhury. Potential of Intelligent Transportation Systems in Mitigating Adverse Weather Impacts on Road Mobility : A Review. *IEEE Transactions on Intelligent Transportation Systems*, 16(3) :1107–1119, June 2015.

[30] M. J. Dworkin. SP 800-38A 2001 Edition. Recommendation for Block Cipher Modes of Operation : Methods and Techniques. Technical report, Gaithersburg, MD, United States, 2001.

[31] F. Erden, S. Velipasalar, A. Z. Alkar, and A. E. Cetin. Sensors in Assisted Living : A survey of signal and image processing methods. *IEEE Signal Processing Magazine*, 33(2) :36–44, March 2016.

[32] S. Fleck and W. Straßer. Smart Camera Based Monitoring System and Its Application to Assisted Living. *Proceedings of the IEEE*, 96(10) :1698–1714, Oct 2008.

[33] E. Fossum. CMOS Image Sensors : electronic camera on a chip. In *Electron Devices Meeting, 1995. IEDM '95., International*, pages 17–25, Dec 1995.

[34] L. Gallo, A. Cilardo, D. Thomas, S. Bayliss, and G. A. Constantinides. Area implications of memory partitioning for high-level synthesis on FPGAs. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2014.

[35] X. Gao and T. Yoshimura. Genetic algorithm based pipeline scheduling in high-level synthesis. In *2013 IEEE 10th International Conference on ASIC*, pages 1–4, Oct 2013.

[36] J. Goeders and S. J. E. Wilton. Signal-Tracing Techniques for In-System FPGA Debugging of High-Level Synthesis Circuits. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1) :83–96, Jan 2017.

[37] J. B. Goeders and S. J. E. Wilton. VersaPower : Power estimation for diverse FPGA architectures. In *2012 International Conference on Field-Programmable Technology*, pages 229–234, Dec 2012.

[38] R. Gonzalez and R. Woods. *Digital Image Processing*. Pearson Education, 2011.

[39] Y. Hara, H. Tomiyama, S. Honda, and H. Takada. Proposal and Quantitative Analysis of the CHStone Benchmark Program Suite for Practical C-based High-level Synthesis. *Journal of Information Processing*, 17 :242–254, 2009.

[40] G. Hegde and N. Kapre. Energy-Efficient Acceleration of OpenCV Saliency Computation Using Soft Vector Processors. In *2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 76–83, May 2015.

[41] M. Hemmati, M. Biglari-Abhari, S. Niar, and S. Berber. Real-time multi-scale pedestrian detection for driver assistance systems. In *2017 54th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2017.

[42] K. Heyse, T. Davidson, E. Vansteenkiste, K. Bruneel, and D. Stroobandt. Efficient implementation of Virtual Coarse Grained Reconfigurable Arrays on FPGAS. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–8, Sept 2013.

[43] H. Hirschmuller. Improvements in Real-Time Correlation-Based Stereo Vision. In *Stereo and Multi-Baseline Vision, 2001. (SMBV 2001). Proceedings. IEEE Workshop on*, pages 141–148, 2001.

[44] E. Homsirikamol and K. Gaj. Can high-level synthesis compete against a hand-written code in the cryptographic domain ? a case study. In *2014 International Conference on ReConFigurable Computing and FPGAs (ReConFig14)*, pages 1–8, Dec 2014.

[45] R. Jevtic and C. Carreras. Power Measurement Methodology for FPGA Devices. *IEEE Transactions on Instrumentation and Measurement*, 60(1) :237–247, Jan 2011.

[46] W. Jiang, C. Xiao, H. Jin, S. Zhu, and Z. Lu. Vehicle Tracking with Non-overlapping Views for Multi-camera Surveillance System. In *2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pages 1213–1220, Nov 2013.

[47] D. H. Jones, A. Powell, C. S. Bouganis, and P. Y. K. Cheung. GPU Versus FPGA for High Productivity Computing. In *2010 International Conference on Field Programmable Logic and Applications*, pages 119–124, Aug 2010.

[48] M. A. Kadi and M. Huebner. Integer computations with soft GPGPU on FPGAs. In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 28–35, Dec 2016.

[49] R. Kalarot and J. Morris. Comparison of FPGA and GPU implementations of real-time stereo vision. In *2010 IEEE Computer Society Conference on Computer Vision and Pattern Recognition - Workshops*, pages 9–15, June 2010.

[50] B. Kienhuis, E. F. Deprettere, P. v. d. Wolf, and K. A. Vissers. A Methodology to Design Programmable Embedded Systems - The Y-Chart Approach. In *Embedded Processor Design Challenges : Systems, Architectures, Modeling, and Simulation - SAMOS*, pages 18–37, London, UK, UK, 2002. Springer-Verlag.

[51] A. Kulkarni, D. Stroobandt, A. Werner, F. Fricke, and M. Hübner. Pixie : A heterogeneous virtual coarse-grained reconfigurable array for high performance image processing applications. *CoRR*, abs/1705.01738, 2017.

[52] A. Lakshminarayana, S. Ahuja, and S. Shukla. High Level Power Estimation Models for FPGAs. In *2011 IEEE Computer Society Annual Symposium on VLSI*, pages 7–12, July 2011.

[53] J. Lapalme, B. Theelen, N. Stoimenov, J. Voeten, L. Thiele, and E. Aboulhamid. Y-chart based system design : a discussion on approaches. 2009.

[54] Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature Magazine*, 521 :436–444, 2015.

[55] F. Li, D. Chen, L. He, and J. Cong. Architecture Evaluation for Power-efficient FPGAs. In *Proceedings of the 2003 ACM/SIGDA Eleventh International Symposium on Field Programmable Gate Arrays*, FPGA '03, pages 175–184, New York, NY, USA, 2003. ACM.

[56] H. Li, S. Katkoori, and W.-K. Mak. Power Minimization Algorithms for LUT-based FPGA Technology Mapping. *ACM Trans. Des. Autom. Electron. Syst.*, 9(1) :33–51, Jan. 2004.

[57] P. Li, Y. Wang, P. Zhang, G. Luo, T. Wang, and J. Cong. Memory partitioning and scheduling co-optimization in behavioral synthesis. In *2012 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 488–495, Nov 2012.

[58] G. Liu, M. Tan, S. Dai, R. Zhao, and Z. Zhang. Architecture and synthesis for area-efficient pipelining of irregular loop nests. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(11) :1817–1830, Nov 2017.

[59] H.-Y. Liu and L. P. Carloni. On learning-based methods for design-space exploration with high-level synthesis. In *2013 50th ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–7, May 2013.

[60] J. Liu, J. Wickerson, and G. A. Constantinides. Loop splitting for efficient pipelining in high-level synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 72–79, May 2016.

[61] C. Lo and P. Chow. Model-based optimization of high level synthesis directives. In *2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–10, Aug 2016.

[62] A. Mahapatra and B. C. Schafer. Machine-learning based simulated annealer method for high level synthesis design space exploration. In *Proceedings of the 2014 Electronic System Level Synthesis Conference (ESLsyn)*, pages 1–6, May 2014.

[63] H. Makhijani and S. Meier. A high level design solution for FPGA's. In *WESCON/94. Idea/Microelectronics. Conference Record*, pages 596–603, Sep 1994.

[64] G. Martin and G. Smith. High-level synthesis : Past, present, and future. *IEEE Design Test of Computers*, 26(4) :18–25, July 2009.

[65] MathWorks. *HDL Coder User's Guide*, September 2016.

[66] M. Mattavelli, I. Amer, and M. Raulet. The reconfigurable video coding standard [standards in a nutshell]. *IEEE Signal Processing Magazine*, 27(3) :159–167, May 2010.

[67] M. McDonnell. Box-Filtering Techniques. *Computer Graphics and Image Processing*, 17(1) :65 – 70, 1981.

[68] W. Meeus, K. Van Beeck, T. Goedemé, J. Meel, and D. Stroobandt. An overview of today's high-level synthesis tools. *Design Automation for Embedded Systems*, 16(3) :31–51, Sep 2012.

[69] D. Meidanis, K. Georgopoulos, and I. Papaefstathiou. FPGA power consumption measurements and estimations under different implementation parameters. In *Field-Programmable Technology (FPT), 2011 International Conference on*, pages 1–6, Dec 2011.

[70] M. Munz, M. Mahlisch, and K. Dietmayer. Generic centralized multi sensor data fusion based on probabilistic sensor and environment models for driver assistance systems. *IEEE Intelligent Transportation Systems Magazine*, 2(1) :6–17, Spring 2010.

[71] R. Nane, V. M. Sima, B. Olivier, R. Meeuws, Y. Yankova, and K. Bertels. DWARV 2.0 : A CoSy-based C-to-VHDL hardware compiler. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 619–622, Aug 2012.

[72] D. Navarro, . Lucı´a, L. A. Barragán, I. Urriza, and . Jiménez. High-level synthesis for accelerating the fpga implementation of computationally demanding control algorithms for power converters. *IEEE Transactions on Industrial Informatics*, 9(3) :1371–1379, Aug 2013.

[73] ON semiconductor. *VITA 2000 2.3 Megapixel 92 FPS Global Shutter CMOS Image Sensor.*

[74] D. J. Pagliari, M. R. Casu, and L. P. Carloni. Accelerators for breast cancer detection. *ACM Trans. Embed. Comput. Syst.*, 16(3) :80 :1–80 :25, Mar. 2017.

[75] M. Pelcat, C. Bourrasset, L. Maggiani, and F. Berry. Design productivity of a high level synthesis compiler versus HDL. In *2016 International Conference on Embedded Computer Systems : Architectures, Modeling and Simulation (SAMOS)*, pages 140–147, July 2016.

[76] N. K. Pham, A. K. Singh, A. Kumar, and M. M. A. Khin. Exploiting loop-array dependencies to accelerate the design space exploration with high level synthesis. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 157–162, March 2015.

[77] C. Pilato and F. Ferrandi. Bambu : A modular framework for the high level synthesis of memory-intensive applications. In *2013 23rd International Conference on Field programmable Logic and Applications*, pages 1–4, Sept 2013.

[78] K. K. W. Poon, S. J. E. Wilton, and A. Yan. A Detailed Power Model for Field-programmable Gate Arrays. *ACM Trans. Des. Autom. Electron. Syst.*, 10(2) :279–302, Apr. 2005.

[79] A. Prost-Boucle, O. Muller, and F. Rousseau. Fast and standalone design space exploration for high-level synthesis under resource constraints. *Journal of Systems Architecture*, 60(1) :79 – 93, 2014.

[80] B. Rinner and W. Wolf. An Introduction to Distributed Smart Cameras. *Proceedings of the IEEE*, 96(10) :1565–1575, Oct 2008.

[81] M. Ruiz, G. Sutter, S. Lopez-Buedo, J. Ramos, J. E. L. de Vergara, and J. Aracil. Leveraging open source platforms and high-level synthesis for the design of FPGA-based 10 GbE active network probes. In *2015 International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, pages 1–6, Dec 2015.

[82] B. C. Schafer. Probabilistic multiknob high-level synthesis design space exploration acceleration. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3) :394–406, March 2016.

[83] B. C. Schafer. Enabling high-level synthesis resource sharing design space exploration in fpgas through automatic internal bitwidth adjustments. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 36(1) :97–105, Jan 2017.

[84] D. Scharstein and R. Szeliski. A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vision*, 47(1-3) :7–42, Apr. 2002.

[85] M. Schmid, N. Apelt, F. Hannig, and J. Teich. An image processing library for C-based high-level synthesis. In *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, pages 1–4, Sept 2014.

[86] M. Schmid, O. Reiche, F. Hannig, and J. Teich. Loop coarsening in C-based High-Level Synthesis. In *2015 IEEE 26th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 166–173, July 2015.

[87] A. Sengupta and R. Sedaghat. Integrated scheduling, allocation and binding in high level synthesis using multi structure genetic algorithm based design space exploration. In *2011 12th International Symposium on Quality Electronic Design*, pages 1–9, March 2011.

[88] A. Severance and G. G. F. Lemieux. Embedded supercomputing in FPGAs with the VectorBlox MXP Matrix Processor. In *2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pages 1–10, Sept 2013.

[89] L. Shang, A. S. Kaviani, and K. Bathala. Dynamic Power Consumption in Virtex™-II FPGA Family. In *Proceedings of the 2002 ACM/SIGDA Tenth International Symposium on Field-programmable Gate Arrays*, FPGA '02, pages 157–164, New York, NY, USA, 2002. ACM.

[90] D. Sharma, V. Dumitriu, and L. Kirischian. *Architecture Reconfiguration as a Mechanism for Sustainable Performance of Embedded Systems in case of Variations in Available Power*, pages 177–186. Springer International Publishing, Cham, 2017.

[91] F. M. Siddiqui, M. Russell, B. Bardak, R. Woods, and K. Rafferty. IPPro : FPGA based image processing processor. In *2014 IEEE Workshop on Signal Processing Systems (SiPS)*, pages 1–6, Oct 2014.

[92] H. Sim, A. Rahman, and J. Lee. Efficient high-level synthesis for nested loops of nonrectangular iteration spaces. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 24(8) :2799–2802, Aug 2016.

[93] K. C. Smith, A. Wang, and L. C. Fujino. Through the Looking Glass : Trend Tracking for ISSCC 2012. *IEEE Solid-State Circuits Magazine*, 4(1) :4–20, winter 2012.

[94] R. Solomon, P. A. Sandborn, and M. G. Pecht. Electronic part life cycle concepts and obsolescence forecasting. *IEEE Transactions on Components and Packaging Technologies*, 23(4) :707–717, Dec 2000.

[95] C. L. Sotiropoulou, S. Gkaitatzis, A. Annovi, M. Beretta, P. Giannetti, K. Kordas, P. Luciano, S. Nikolaidis, C. Petridou, and G. Volpi. A Multi-Core FPGA-Based 2D-Clustering Implementation for Real-Time Image Processing. *IEEE Transactions on Nuclear Science*, 61(6) :3599–3606, Dec 2014.

[96] F. Sun, H. Wang, F. Fu, and X. Li. Survey of FPGA low power design. In *2010 International Conference on Intelligent Control and Information Processing*, pages 547–550, Aug 2010.

[97] X. Tang, P. E. Gaillardon, and G. D. Micheli. FPGA-SPICE : A simulation-based power estimation framework for FPGAs. In *2015 33rd IEEE International Conference on Computer Design (ICCD)*, pages 696–703, Oct 2015.

[98] Texas Instruments. *Fusion Digital Power Designer GUI for Isolated Power Applications*, June 2014.

[99] D. B. Thomas. Synthesisable recursion for C++ HLS tools. In *2016 IEEE 27th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*, pages 91–98, July 2016.

[100] T. Tziortzios and S. Dokouzyannis. High throughput and energy efficient two-dimensional inverse discrete cosine transform architecture. *IET Image Processing*, 7(5) :533–541, July 2013.

[101] J. Villarreal, A. Park, W. Najjar, and R. Halstead. Designing Modular Hardware Accelerators in C with ROCCC 2.0. In *2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, pages 127–134, May 2010.

[102] S. Vishwakarma and A. Agrawal. A survey on activity recognition and behavior understanding in video surveillance. *The Visual Computer*, 29(10) :983–1009, Oct 2013.

[103] V. Viswanathan. A Scalable Flexible and Dynamic Reconfigurable Architecture for High Performance Embedded Computing, PhD dissertation, University of Valenciennes, October, 2015.

[104] Y. Wang, P. Li, and J. Cong. Theory and Algorithm for Generalized Memory Partitioning in High-level Synthesis. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 199–208, New York, NY, USA, 2014. ACM.

[105] C. D. Ward and C. W. Sohns. Electronic component obsolescence. *IEEE Instrumentation Measurement Magazine*, 14(6) :8–12, December 2011.

[106] S. Windh, X. Ma, R. J. Halstead, P. Budhkar, Z. Luna, O. Hussaini, and W. A. Najjar. High-level language tools for reconfigurable computing. *Proceedings of the IEEE*, 103(3) :390–408, March 2015.

[107] Xilinx. *A Simple Method of Estimating Power in XC4000XL/EX/E FPGAs.*

[108] Xilinx. *ZC706 Evaluation Board for the Zynq-7000 XC7Z045 All Programmable SoC User Guide.*

[109] Xilinx. *LogiCORE IP Color Filter Array Interpolation v3.0*, December 2010.

[110] Xilinx. *Power Methodology Guide*, April 2013.

[111] Xilinx. *UG902 Vivado Design Suite User Guide High-Level Synthesis*, June 2015.

[112] Z. Xue and D. B. Thomas. SynADT : Dynamic Data Structures in High Level Synthesis. In *2016 IEEE 24th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 64–71, May 2016.

[113] J. Yan, J. Yuan, P. H. W. Leong, W. Luk, and L. Wang. Lossless compression decoders for bitstreams and software binaries based on high-level synthesis. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 25(10) :2842–2855, Oct 2017.

[114] P. Yiannacouras, J. G. Steffan, and J. Rose. Portable, flexible, and scalable soft vector processors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 20(8) :1429–1442, Aug 2012.

[115] J. Yudi, C. H. Llanos, and M. Huebner. System-level design space identification for Many-Core Vision Processors. *Microprocessors and Microsystems*, 52(Supplement C) :2 – 22, 2017.

[116] M. Zabłocki, K. Gościewska, D. Frejlichowski, and R. Hofman. Intelligent video surveillance systems for public spaces – a survey. *Journal of Theoretical and Applied Computer Science*, 8 :13–27, 2014.

[117] J. Zhang, F. Y. Wang, K. Wang, W. H. Lin, X. Xu, and C. Chen. Data-Driven Intelligent Transportation Systems : A Survey. *IEEE Transactions on Intelligent Transportation Systems*, 12(4) :1624–1639, Dec 2011.

[118] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar. Lin-Analyzer : A high-level performance analysis tool for FPGA-based accelerators. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6, June 2016.