ViPar: High-Level Design Space Exploration for Parallel Video Processing Architectures

Karim Mohamed Abedallah Ali, Rabie Ben Atitallah, Abdessamad Ait El Cadi, Nizar Fakhfakh, Jean-Luc Dekeyser

Hindawi
International Journal of Reconfigurable Computing
Volume 2019, Article ID 4298013, 19 pages
https://doi.org/10.1155/2019/4298013

Research Article

ViPar: High-Level Design Space Exploration for Parallel Video Processing Architectures

Karim M. A. Ali,[1] **Rabie Ben Atitallah**,[2] **Abdessamad Ait El Cadi**,[1] **Nizar Fakhfakh**,[3] **and Jean-Luc Dekeyser**[4]

[1]*LAMIH, Polytechnic University of Hauts-de-France, Valenciennes, France*
[2]*Computer Science Department, University of Galatasaray, Istanbul, Turkey*
[3]*NAVYA Company, Villeurbanne, France*
[4]*CRIStAL, University of Lille1, Villeneuve-d'Ascq, France*

Correspondence should be addressed to Karim M. A. Ali; karim.ali@uphf.fr

Received 13 June 2019; Revised 22 September 2019; Accepted 3 October 2019; Published 14 November 2019

Academic Editor: John Kalomiros

Embedded video applications are now involved in sophisticated transportation systems like autonomous vehicles and driver assistance systems. As silicon capacity increases, the design productivity gap grows up for the current available design tools. Hence, high-level synthesis (HLS) tools emerged in order to reduce that gap by shifting the design efforts to higher abstraction levels. In this paper, we present ViPar as a tool for exploring different video processing architectures at higher design level. First, we proposed a parametrizable parallel architectural model dedicated for video applications. Second, targeting this architectural model, we developed ViPar tool with two main features: (1) An empirical model was introduced to estimate the power consumption based on hardware utilization and operating frequency. In addition to that, we derived the equations for estimating the hardware utilization and execution time for each design point during the space exploration process. (2) By defining the main characteristics of the parallel video architecture like parallelism level, the number of input/output ports, the pixel distribution pattern, and so on, ViPar tool can automatically generate the dedicated architecture for hardware implementation. In the experimental validation, we used ViPar tool to generate automatically an efficient hardware implementation for a Multiwindow Sum of Absolute Difference stereo matching algorithm on Xilinx Zynq ZC706 board. We succeeded to increase the design productivity by converging rapidly to the appropriate designs that fit with our system constraints in terms of power consumption, hardware utilization, and frame execution time.

1. Introduction

Nowadays, embedded video processing applications are widely spread in our daily life. Among these applications, we can mention public area video surveillance [1], crowd behaviour analysis for detecting abnormal activities [2, 3], vehicle tracking [4], intelligent transportation systems [5, 6], assisted living for elder people [7, 8], simultaneous localization and mapping (SLAM) problem [9, 10], counting passengers in vehicles [11], real-time autonomous localization [12], yawning detection [13], monitoring systems for kids safety, analyzing customer behaviour in markets, and so on.

Latest semiconductor technologies offer powerful low-cost computing hardware solutions which motivate industry and academia towards developing intelligent sensors. For video processing, smart camera systems are becoming more attractive due to several advantages. In such sensor, images are captured and processed locally to avoid remote computing or specific communication infrastructure. In addition to that, complex algorithms for data analysis and decision making could be performed to bring autonomy capability to systems like in drones and autonomous vehicles.

In this context, field-programmable gate array (FPGA) technology is a competitive solution for building embedded

video processing applications when compared to other available solutions in the market like CPU, GPU, or ASIC circuits for different reasons [14]: (i) FPGA devices are reprogrammable platforms where the hardware architecture can be redesigned to adapt with the rapid changes in the image sensor technologies or in video processing algorithms without changing the chip itself. (ii) By exploiting the inherent parallelism in video processing applications, FPGA technology enables to implement massively parallel architectures due to the huge number of programmable logics available on a single chip. (iii) FPGA-based system is considered as a solution between ASIC and processor-based system where FPGAs are characterized by their high performance per watt; thus, they are good candidate for embedded video processing applications.

Several challenges arose while designing embedded video processing applications using reconfigurable technology. First, we need to develop a flexible parallel reconfigurable architecture where huge amount of pixels is transferred from/to the computing nodes. Today, the increasing demand for more frame rate or increasing the image resolution are additional challenges for video processing applications especially when real-time constraints are considered. This challenge is augmented in the 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. Thus, we need to know how to abstract the hardware architecture to build a flexible parallel architecture where we can fix the parallelism level according to the desired performance or the available hardware resources. 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 are a time-consuming process. For that reason, designers always aim to move the design efforts to the higher abstraction levels by using high-level synthesis (HLS) tools 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 solutions where the design points 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 consisting of hundreds or thousands of design points. Accordingly, automating the exploration process is necessary to search rapidly for the solution which better fits with the given system constraints.

To address the aforementioned challenges, our contributions in this work can be summarized as follows:

(1) We proposed a generic model for parallel hardware video processing architecture. For that model, we defined the required parameters to describe the architecture at the system and processing element level. The architectural parameters are then considered as an input to automate the high-level code generation.

(2) 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 should be applied in order to obtain an efficient design. In this work, we classified the HLS optimizations either for improving the hardware implementation or for exploiting the inherent parallelism in the application. In addition to that, we showed the impact of applying each optimization step on the overall design efficiency in terms of hardware utilization, power, and performance.

(3) We developed ViPar tool to automate the design space exploration through the following steps: (i) Both resource utilization and performance were estimated for each point in the design space. (ii) An empirical power model was introduced to estimate the power consumption for each design based on the utilized resources and operating frequency. (iii) The high-level code of the parallel video processing architecture was automatically generated for experimental validation. In the frame of our industrial collaboration, during our experiments, stereo matching algorithm was chosen as a video processing application.

The rest of this paper is organized as follows: related works are overviewed in Section 2. Section 3 explains the stereo matching algorithm used in this work. Section 4 details our parameterizable parallel architectural model. Section 5 lists the applied HLS optimizations to obtain efficient hardware implementation, while Section 6 describes how ViPar tool works. The experimental results are presented in Section 7. Finally, Section 8 summarizes our presented work.

2. Related Works

In the literature, hundreds of research works present the implementation of various video processing applications over different hardware architectures ranging from CPU, GPU, FPGA, DSP, and ASIC. For FPGA, the property of being reconfigurable inspires designers to build architectures of soft-core processors like vector, VLIW, and GPGPU over FPGA. VectorBlox MXP [15] 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. 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. 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 [15, 16]. Authors in [17] proposed a customizable VLIW processor with a variable instruction set for exploiting parallelism. For experimental validation, three basic image applications were tested on the Virtex-6 FPGA board; then, authors extended their experiments by realizing a contactless palmprint extraction algorithm for biometric applications. Their VLIW implementation over FPGA showed an average speedup of

2.7x when compared to DSP-based implementation for the same application. FlexGrip [18] is a soft GPGPU processor where compiled CUDA binaries are directly executed on the FPGA-based GPGPU without hardware resynthesis. FlexGrip follows the single-instruction multiple-thread (SIMT) model where one instruction is fetched and executed simultaneously by multiple SP cores. 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. FGPU [19] is another soft GPGPU processor with single level-cache optimized for FPGA. The authors developed a compiler in order to support applications written in OpenCL [20]. FGPU has an extended MIPS assembly instruction set with additional instructions to support the OpenCL execution model. In the experimental results, the authors compared 11 applications including three image filters implemented on FGPU to other platforms.

High-level synthesis 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 the 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 [21, 22]. The second class considers the HLS tool as a black box and explores the design space of the optimization parameters offered by the tool. This class can be further subdivided into (i) HLS synthesis directives and (ii) resource sharing. For the first subclass, the HLS directives are inserted into the behavioural code as comments to affect the final synthesized microarchitecture. For example, loops can be pipelined or not with complete or partial unrolling. Arrays can also be 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 can be shared among different operations in 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 [23] 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, and then continued 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 [24] 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 [25] which exploited 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 [26]. Lin-Analyzer [27] 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 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 [28], simulated annealing [29], ant colony [23], or machine-learning techniques [30].

In this paper, we proposed a parameterizable generic architecture for video processing where the processing elements are dedicated for a certain video application to have area and power customized. The processing element was optimized by applying HLS optimizations; then, the architecture was explored at higher design level by tuning different parameters like parallelism level, operating frequency, and so on. Comparing to the existed tools, ViPar tackles the exploration challenge at more abstracted level making profit from the application properties (image size, sliding window size, parallelism level, etc.) and the system requirements (frame rate, power consumption, etc.). Our tool compares between different design points by estimating power consumption, hardware utilization, and performance, and then it generates automatically the parallel architecture for the best candidate designs.

3. Sum of Absolute Difference 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. The difference between the two points on the image plane is defined as *disparity*, as depicted in Figures 1(a) and 1(b).

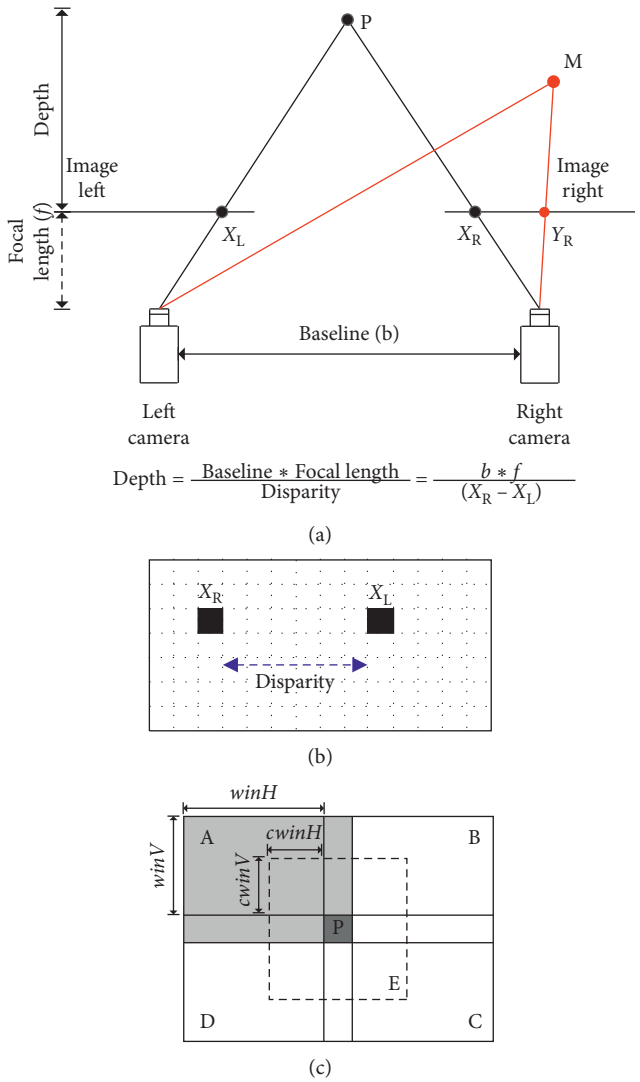

Figure 1: (a) Calculating the depth of an object in stereo matching problem. (b) Disparity is defined as the distance in pixels between (X_R) and (X_L). (c) 5-window SAD configuration.

Several algorithms were proposed in the literature to find the best matching [31]. One of these algorithms is multiwindow sum of absolute difference algorithm (multiwindow SAD) [32], where the 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 appears at the regions of depth discontinuity, the correlation window can be divided into smaller windows, and only nonerror parts are considered in the calculations. Figure 1(c) shows 5-window SAD configuration, where pixel (P) lies in the middle of window (E) of height = $2 * cwinV + 1$ and width = $2 * cwinH + 1$ and surrounded by another four windows named A, B, C, and D of height = $winV + 1$ and width = $winH + 1$. We defined *window score* as the aggregation of the absolute difference of the pixels within that window.

The algorithm is described in Listing 1, where it scans all the image pixels at every disparity value ranging from zero to the maximum value (DISP_MAX). The calculation of window (E) is usually performed independently from the others because its size is different from others. The for-loop described between Lines 2–27 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 $N \times N$ with an aggregation window of size $M \times M$, then $N^2(M^2 - 1)$ addition operations are needed for window aggregation at every single pixel. However, by applying box-filtering [33] in both horizontal and vertical directions, the number of additions will be reduced to $4N^2$. Box-filtering is applied in both horizontal and vertical directions (Lines 5–12) 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 scores of the other four windows (Lines 15–21). 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 22–24). 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 the reference, while vice versa happens in the second one (Lines 22–27). 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 28–31).

4. Parallel Video Processing Architecture Model

Our proposed parameterizable architectural model is depicted in Figure 2. It is an array-based video processing architecture, where N processing elements are running in parallel. Each processing 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). In our previous work, we implemented a generic architecture for pixel distributor/collector that depends on the properties of the image processing algorithm (like macroblock size, sliding window stride, etc.) [34].

In this model, we defined a set of parameters to describe the properties of the parallel video architecture at the processing element, the system level, and for the top-level function. At the processing element level, we defined the port properties for the input ports (X_0, X_1, ..., X_i) and output ports (Y_0, Y_1, ..., Y_j). These port properties include name, data type, the source of the input pixel stream (*src*), and the range of image scanlines which are mapped to that

```
Algorithm: Multi_Window_SAD
  for every disparity(d) where d = 0 ⟶ DISP_MAX:
    for every element(x, y) where x = 0 ⟶ imgW, y = 0 ⟶ imgH:
      Abs[y][x] = abs(IMG_R[y][x] − IMG_L[y][x + d]);
    for every element(x, y) where x = winH ⟶ imgW−winH, y = 0 ⟶ imgH:
        row[y][x] = row[y][x − 1] + Abs[y][x] − Abs[y][x − winH];
      rowE[y][x] = rowE[y][x − 1] + Abs[y][x] − Abs[y][x − 2 ∗ cwinH];
    for every element(x, y) where x = 0 ⟶ imgW, y = winV ⟶ imgH−winV:
        scr[y][x] = scr[y − 1][x] + row[y][x] − row[y − winV][x];
      scrE[y][x] = scrE[y − 1][x] + rowE[y][x] − rowE[y − 2 ∗ cwinV][x];
    for every element(x, y) where x = winH ⟶ imgW−winH, y = winV ⟶ imgH−winV:
      scoreA = scr[y][x];
      scoreB = scr[y][x + winH];
      scoreC = scr[y + winV][x + winH];
      scoreD = scr[y + winV][x];
      scoreE = scrE[y + winV − cwinV][x + winH − cwinH];
      min_score = scoreE + MIN_2_values{scoreA, scoreB, scoreC, scoreD};
      if min_score < bestscoreR[y + winV][x + winH]:
        bestscoreR[y + winV][x + winH] = min_score;
        DISP_IMG_R[y + winV][x + winH] = d;
      if min_score < bestscoreL[y + winV][x + winH + d]:
        bestscoreL[y + winV][x + winH + d] = min_score;
        DISP_IMG_L[y + winV][x + winH + d] = d;
  for every element(x, y) where x = 0 ⟶ imgW, y = 0 ⟶ imgH:
    dispVal = DISP_IMG_R[y][x];
    if abs(DISP_IMG_L[y][x + dispVal] −>dispVal) > 1:
      DISP_IMG_R[y][x] = 0;
```

Algorithm 1: Pseudocode for multiwindow SAD algorithm.

port during execution (*store_scanlines_from*, *store_scanlines_to*). For example, for the 5-window SAD algorithm, we defined six input ports for AB, CD, and E windows, where each window has two inputs for the left and right images. If the window size for AB and CD is 23 x 8, then (*store_scanlines_from* and *store_scanlines_to*) will be 0 and 8 for window AB, while it will be 8 and 15 for window CD.

At the top-level function level, we defined the properties of the input/output stream ports. These port properties include their 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. While the system properties define the general system properties such as the size of the input/output images and what level of parallelism is realized. For example, in autonomous vehicles, our input image size was 640 x 480 for the multiwindow SAD stereo matching algorithm with a global constraint of minimum 15 frames/s. With such high-level system description, hardware implementation details are hidden; hence, it facilitates the system design for video processing developers.

When the depicted architecture is described at 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. The same image scanline could be mapped to several processing elements to guarantee data-level parallelism execution. Indeed, 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. This challenge is duplicated when a large number of parallel architectures need to be examined in order to find an efficient implementation that fulfils the design requirements. Consequently, coding these architectures manually is a time-consuming process and an error-prone process. In order to address the above challenge, we developed ViPar tool which explores the design space, and then automatically generates the high-level codes for the best candidate parallel architectures. The generated code is then compiled by the HLS tool to obtain the corresponding RTL design. In the next section, we will expose the high-level optimizations introduced in order to obtain an efficient processing element before exploring the possible parallel architectures by using ViPar tool.

5. High-Level Synthesis Optimizations

Using high-level synthesis (HLS) tools in electronic design automation (EDA) aims at moving the design efforts to higher abstraction levels. Today, it becomes more easier to design, implement, and verify complex video processing algorithms on reconfigurable technology by using high-level synthesis tools. In this section, we will classify HLS optimizations either for improving the hardware implementation or for exploiting the inherent parallelism in the video

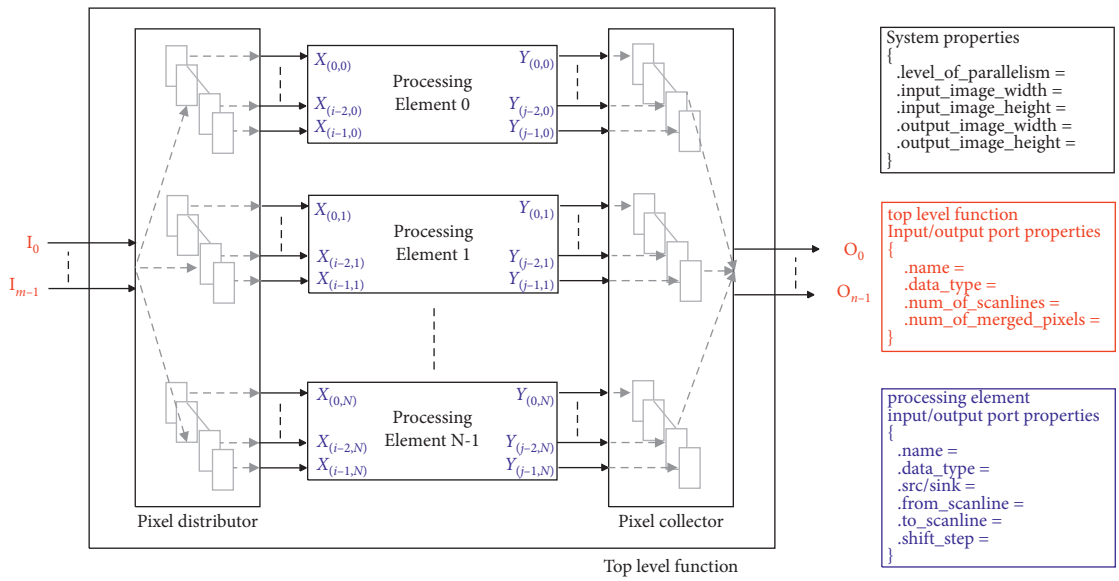

Figure 2: Array-based video processing architecture.

processing application. In order to obtain an efficient hardware implementation, the high-level code is subjected to a set of HLS optimization steps [35]. We will show the impact of these optimizations in terms of hardware utilization and performance on the multiwindow SAD algorithm.

5.1. Optimizations for Better Hardware Implementation. This type of optimization modifies the software code to fit for hardware implementation.

5.1.1. Dividing the 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). Design #1 showed an overuse for BRAM, where the FPGA platform has maximum BRAM_18K = 1090, as listed in Table 1. In strip processing, loop boundaries and array dimensions are updated to reflect a strip size processing area instead of full image size, and the code will be repetitively executed until all the strips are processed. For example, the image height is updated from *imgH* to *stripH*. In the multiwindow SAD algorithm, the pixels can be summed in three different ways: (i) Design #2 aggregates the pixels in the horizontal direction then the result is aggregated in the vertical one. (ii) While in Design #3, the aggregation is done vertically along the column length then horizontally along the scanline. (iii) However, in Design #4, the pixels are aggregated within the window boundary in both directions. Table 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.

5.1.2. Using Arbitrary Precision Data Types. HLS tools support arbitrary precision data types by defining variables with smaller bitwidth. Instead of using the native C-based data types of width 8, 16, 32, and 64 bit, we can define our variables with adjustable bitwidth to produce systems of the same accuracy but with less area utilization. By doing bitwidth analysis, we can know the lower and upper limits for each variable, and then the required number of bits can be exactly assigned. In Table 1, Design #5 showed around 31% reduction for LUT and 40% reduction for FF.

5.1.3. Choosing I/O Interface Protocols. The generated hardware block is connected to other blocks in the design through various types of I/O protocols. Different communication protocols are available 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 classes of ports can be defined as follows: (1) Clock and Reset ports. (2) Block-level interface protocol is used to control and check the current state of the HLS block (start, ready, busy, or done state). (3) Port-level interface protocol is created for each argument in the top-level function with various configurations like memory interface, two-way handshaking with valid and acknowledge signals, or as AXI4 interfaces (AXI4-Stream, AXI4-Lite, and AXI4 master). In our design, AXI-Stream was chosen for the port-level interface. While AXI-Lite was selected for the block-level interface protocol to control the operation of the hardware block. In Table 1, Design #6 listed the hardware cost after adding the I/O interface protocol.

5.1.4. Grouping Input/Output Pixels. If the I/O pixels are of size less than the bitwidth of the communication bus, 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

Table 1: Synthesis results for each optimization step.

	Optimization	Slice	FF	LUT	BRAM (18K)	Time (ms)
SW version running at 380 ms on core i7@ 2.7 GHz and 16 GB of RAM						
#1	First synthesizable design	X	2637	5918	7392	X
#2	Hor. then ver. aggregation	898	1743	2735	155	30080
#3	Ver. then hor. aggregation	859	1758	2659	113	22410
#4	In both directions	1400	2552	3738	75	8163
#5	Arbit. prec. data types	983	1525	2567	47	5786
#6	I/O interface protocols	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 disp. lines	2790	4578	7796	102	815
#10	Calculating 8 disp. lines	5012	8502	14027	204	432
#11	Calculating 12 disp. lines	6594	12563	18476	252	339
#12	Loop pipelining	1161	2004	3546	67	1174
#13	False dependency	1115	2030	3433	67	1002
#14	Data-level parallelism	2771	6365	8155	59	313

from the designer while separating the pixels at the input ports or merging them at the output ports. In our design, the communication bus is 64-bit, the input pixel is 32-bit, 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. Design #7 showed 7% improvement in the execution time as listed in Table 1.

5.2. Optimizations for Exploiting the Inherent Parallelism. This type of optimization modifies the software code to exploit the parallelism in the application at different levels (pipeline-level, task-level, or data-level parallelism). We will exploit the inherent parallelism in the multiwindow SAD algorithm to improve the execution time.

5.2.1. Task-Level Parallelism. In task-level parallelism, independent data tasks can be executed concurrently. For 5-window SAD algorithm, as shown in Figure 1, 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 is applied for windows (C) 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 and (ii) to rewrite them in separated functions to allow the HLS tool to schedule them in parallel. The common image lines between windows are duplicated to allow data-independent window calculations. For example, the common image lines between windows A/B and E are duplicated to the local arrays of both. In Table 1, Design #8 reports the effect of applying task-level parallelism where the execution time is improved by around 50%.

5.2.2. 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 3 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 the 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 1. The other way of performing pipeline-level parallelism is by adding HLS directives like LOOP_PIPELINE directive to the 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 1 reported the hardware cost and the execution time after applying the pipeline directive on Design #8. 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 interloop-dependency, II = 1 cannot 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. In Table 1, Design #13 showed 15% gain in execution time than Design #12 when false interloop-dependency is removed.

5.2.3. 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 sets of data in parallel. Data-level parallelism is applied in two different ways: (i) by applying ARRAY_PARTITION and LOOP_UNROLL directives and (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. LOOP_UNROLL directive duplicates the computation process to operate on a different set of data by creating multiple copies of the loop body. Loops 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.

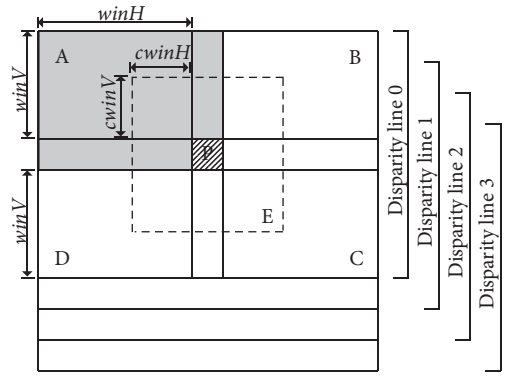

Figure 3: Enlarging the strip height to calculate 4 disparity lines.

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 1. 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 section, we will use the model described in Section 4 to define the required parameters to implement a parallel hardware architecture consisted of N processing elements by the help of ViPar tool.

6. ViPar Tool

In this section, we will describe the design flow of ViPar tool and how the performance metrics like area, power, and execution time are estimated by our tool at a higher design level. In addition to that, we will show how the high-level code is generated by ViPar after defining the main properties of the parallel architecture.

6.1. ViPar Tool Design Flow. Figure 4 depicts the design flow for ViPar tool where the initial input is the performance metrics of the design at parallelism level = 1. These metrics include area utilization, power consumption, and execution time. During the area estimation phase, 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 which FPGA chip is selected during the exploration process. After that, the produced set of design alternatives is estimated for power consumption and execution time as well. According to the system constraints, the design space is pruned to the set of the candidate designs. Then, high-level code generation is done to generate automatically the C++ files for each design candidate. The generated code constrained by HLS optimizations/user constraints are considered as the input to the high-level synthesis tool to obtain the RTL design. Later, the RTL design is implemented to generate the design bit stream. Experimentally, we can measure the design metrics to verify how far are the estimations from the real values and to make sure that the system constraints are fulfilled.

6.2. Area Estimation. Two factors affect the resource utilization of the implemented design: (i) *Synthesis/Implementation strategy*. The synthesis tool offers a set of predefined strategies to obtain the hardware implementation. It is a multiobjective problem where each strategy has a certain objective to optimize like power, area, performance, and so on. (ii) *Operating frequency*. The synthesis tool tries to achieve timing closure alongside satisfying the objective of the applied strategy. At higher operating frequencies, the tool allocates more hardware resources to satisfy the timing constraints.

Figure 5 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 from the figure: (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 for the same design implemented by two different synthesis strategies is small at lower parallelism level, while it becomes more significant for high levels of parallelism.

In highlight of the previous observations and based on the depicted parallel video processing architecture in Figure 2, the hardware cost in terms of slice, FF, LUT, and BRAM can be divided into (i) *Base cost* which represents the required resources for implementing the basic blocks existing in every single design. These basic blocks include AXI-DMA blocks for pixel transfer, AXI-interconnect blocks, AXI-VDMA blocks for video target peripherals, and so on. (ii) *Parallel cost* which represents the required hardware resources for implementing the processing elements of the parallel architecture. We can deduce a linear relation for estimating slice, LUT, FF, and BRAM as follows:

$$\begin{aligned}&\text{estimated utilization}|_{\text{parallelism level}=N}\\&\quad=\text{base cost}+N*\text{utilization}|_{\text{parallelism level}=1}\end{aligned}\tag{1}$$

where *Base_cost* is the hardware cost for the basic blocks in the design and N is the level of parallelism. For slice estimation, we have another equation based on the fact that the slice is composed of FFs and LUTs (for example, in Zynq ZC706, one slice consists of 8 FFs and 4 LUTs). Therefore, the estimated slice utilization can be formulated as follows:

$$\begin{aligned}&\text{estimated slice utilization}|_{\text{parallelism level}=N}\\&\quad=\max\left\{\frac{estimated\_LUT|_{\text{parallelism}=N}}{num\_LUT\_per\_Slice},\frac{estimated\_FF|_{\text{parallelism}=N}}{num\_FF\_per\_Slice}\right\},\end{aligned}\tag{2}$$

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.

6.3. Power Estimation Model. There are three types contributing to the power consumption in FPGA: static power, short-circuit power, and dynamic power. Dynamic power

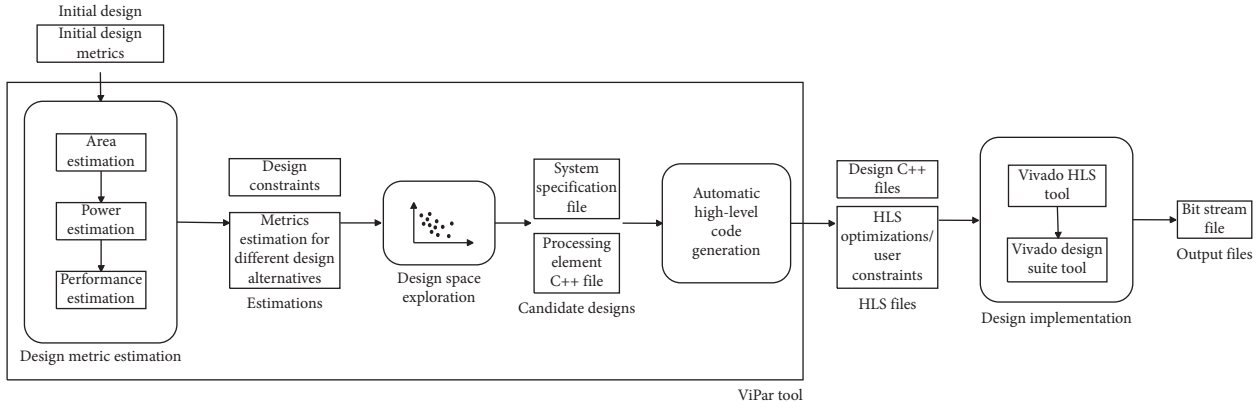

Figure 4: Design flow with ViPar tool.

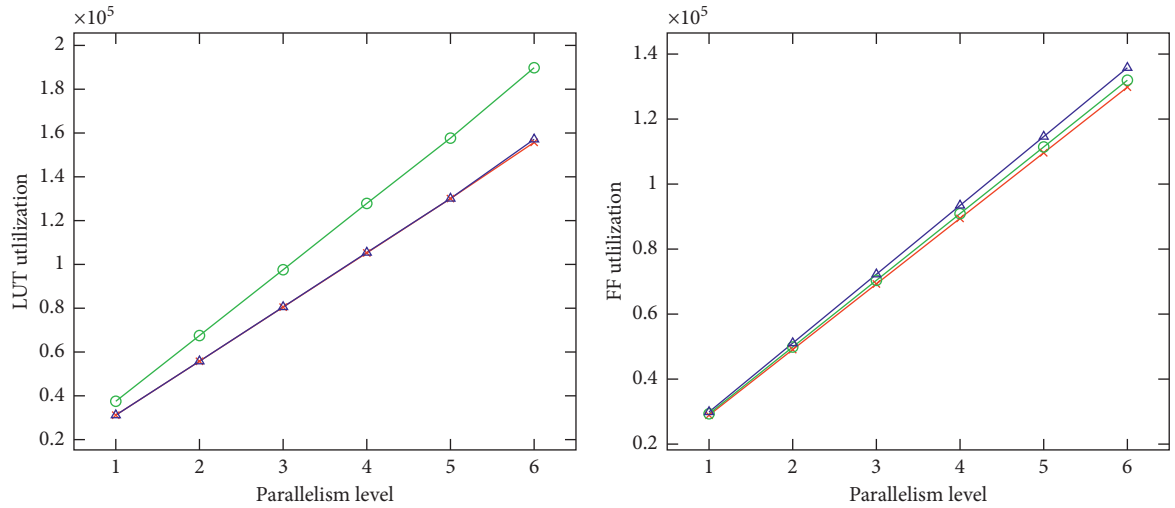

Figure 5: LUT/FF utilization at different parallelism levels for *Default* strategy at 100 MHz, *Default* strategy at 200 MHz, and *Performance_Explore* strategy at 100 MHz.

could be further broken down into power consumed by clocks, interconnect wires, and hardware resources. Equation (3) [36] 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.

$$\text{Power}_{\text{dynamic}} = \frac{1}{2} f V_{dd}^2 \sum_{i=1}^{n} C_i S_i. \quad (3)$$

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 fact, obtaining a detailed placement and routing design takes a considerable time which could range between 30 and – 90 min or even more for larger designs. During exploration, it is not practical to run a detailed implementation to estimate the power consumption for every single point. Instead of that, quick power estimations are accepted at that early design stage to compare the power consumption of different design points. In this work, we will present an empirical power model based on hardware resources and operating frequency characterization.

6.4. Power Measurement. The power was measured through an UCD90120A power controller mounted on the Zynq-ZC706 board using TI Fusion Digital Power Designer software. 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. 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 while operating at frequencies of

values 50, 100, 150, and 200 MHz. We kept increasing the level of parallelism till full hardware utilization. For each design, the power was measured practically, and then it was plotted versus hardware utilization, as depicted in Figure 6.

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.

6.5. Power Regression Model. The power estimation model was built by varying the independent variables (slice, BRAM, and frequency) as follows: slice at 9K, 15K, 25K, 35K, 45K, 50K, and 54K, BRAM at 96, 156, 246, 366, 486, 606, 846, and 1026, and frequency at 50, 100, 150, and 200 MHz. For each experiment, we measured between 2700 and −3000 samples where all the measured samples are then arranged in one spreadsheet as an input for regression analysis to estimate the relationship between power, frequency, slice, and BRAM. There are various kinds of regression models for power prediction. In our study, we will compare three models (linear, pure-quadratic, and full-quadratic) to choose the one which better fits.

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

(i) *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.

(ii) *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 $\sum \text{error}^2 = \sum (\text{power}_{\text{actual}} - \text{power}_{\text{predicted}})^2$.

(iii) *The Coefficient of Determination (R-squared)*. R^2 tells how many points fall on the regression line. R-squared is explained by a value ranging between 0 and 1.

$$R^2 = 1 - \frac{\text{Residual sum of squares}}{\text{Total sum of squares}}. \quad (4)$$

(iv) *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.

(v) *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 $(1 - \alpha)$.

(vi) *P-Value*. It is used in the hypothesis test for either supports or rejects the null hypothesis. *P*-value is an evidence against the null hypothesis where the smaller the *P*-value, the stronger evidence to reject the null hypothesis.

(vii) *Significance-F*. It is the probability that the regression equation does not explain the variation in the dependent variable (power).

Table 2 lists the regression analysis for the three models (linear, pure-quadratic, and full-quadratic). For the three models, Table 2 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 the linear model, 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 pure-quadratic to have fewer model parameters or to choose 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 the full-quadratic model ($\beta_0 \longrightarrow \beta_9$); we could conclude that the null hypothesis is rejected for all coefficients ($\beta_0 \longrightarrow \beta_9$), where $P < 0.01$. The full-quadratic power estimation model is described in the following equation:

$$\text{power}|_{\text{estimated}} = \beta_0 + \beta_1 * S + \beta_2 * B + \beta_3 * F + \beta_4 * S * B + \beta_5 * S * F + \beta_6 * B * F + \beta_7 * S^2 + \beta_8 * B^2 + \beta_9 * F^2, \quad (5)$$

where S is slice, B is BRAM, F is Frequency, $\beta_0 = 0.2228$, $\beta_1 = 1.29 \times 10^{-6}$, $\beta_2 = 9.39 \times 10^{-5}$, $\beta_3 = 11.6 \times 10^{-5}$, $\beta_4 = 2.38 \times 10^{-9}$, $\beta_5 = 7.03 \times 10^{-8}$, $\beta_6 = 5.51 \times 10^{-7}$, $\beta_7 = 4.6 \times 10^{-11}$, $\beta_8 = 9.5 \times 10^{-8}$, and $\beta_9 = 4.91 \times 10^{-7}$.

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 that data. In addition to that, both the sum and the mean average of the residual values should equal to zero. Figure 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 almost equal to zero (2.76245 e-09 and 2.75826 e-14, respectively). From this analysis, we can conclude that our full-quadratic model described in equation (5) is valid.

6.6. Estimating Execution Time. The execution time for video processing application is affected by the number of parallel processing channels and the applied operating frequency. It is common to divide one image into strips during frame processing. The execution time for strip processing is formulated in equation (8) which is equal to the summation of

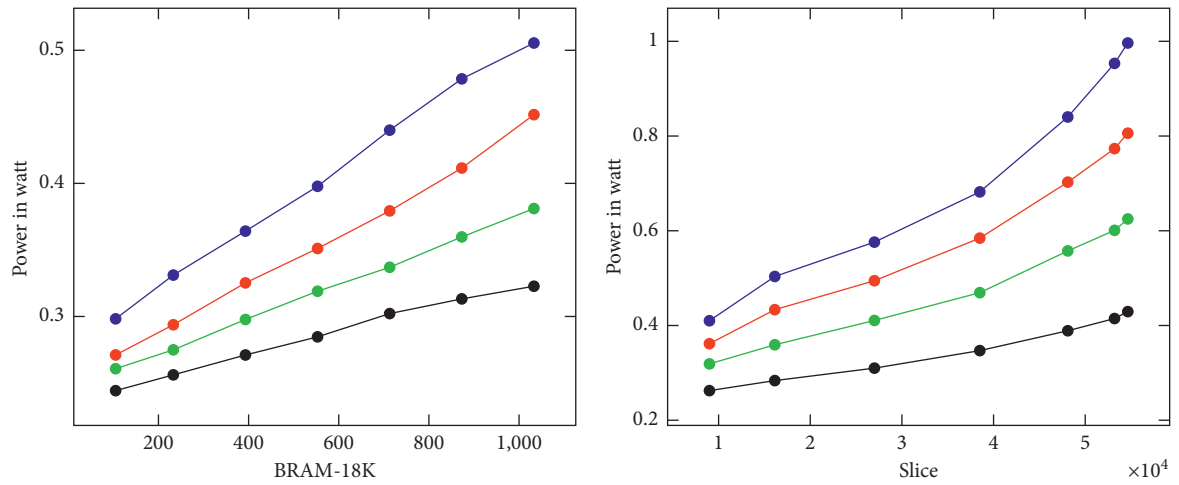

Figure 6: Measured power with BRAM and slice variation at frequencies 50 MHz, 100 MHz, 150 MHz, and 200 MHz.

Table 2: Regression analysis for linear, pure-quadratic, and full-quadratic power estimation model.

	Linear mode	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

clock cycles required to transfer the pixels from/to the processing element plus the clock cycles required for algorithm processing.

$$\begin{aligned}&\text{Strip processing (in cycles)}\\&= \text{cycles}|_{\text{writing input pixels}} + \text{cycles}|_{\text{PE processing}}\\&\quad + \text{cycles}|_{\text{reading output pixels}}.\end{aligned} \tag{6}$$

where $\text{Cycles}|_{\text{writing input pixels}}$ is the number of clock cycles required to transfer the pixels from the memory to the processing element through DMA communication, $\text{Cycles}|_{\text{PE Processing}}$ is the number of clock cycles required by the processing element for executing the application, and $\text{Cycles}|_{\text{reading output pixels}}$ 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 are there in one image frame:

$$\begin{aligned}&Num\_of\_strips\\&= \frac{Num\_image\_lines}{Num\_output\_lines|_{\text{parallelism level}=1} * \text{parallelism level}}.\end{aligned} \tag{7}$$

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

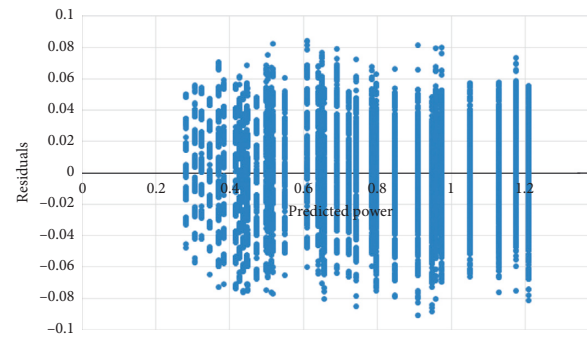

Figure 7: Residual plot for the full-quadratic regression model.

one image strip processing. From the previous equations, the frame execution time can be calculated as follows:

$$\begin{aligned}&\text{frame execution time (in seconds)}\\&= \frac{Num\_of\_strips * \text{strip processing (in clk cycles)}}{\text{frequency (in MHz)}}.\end{aligned} \tag{8}$$

6.7. Automatic High-Level Code Generation. Figure 8 shows that the high-level code generation tool has two input files which are (1) *Processing Element C++ File* which

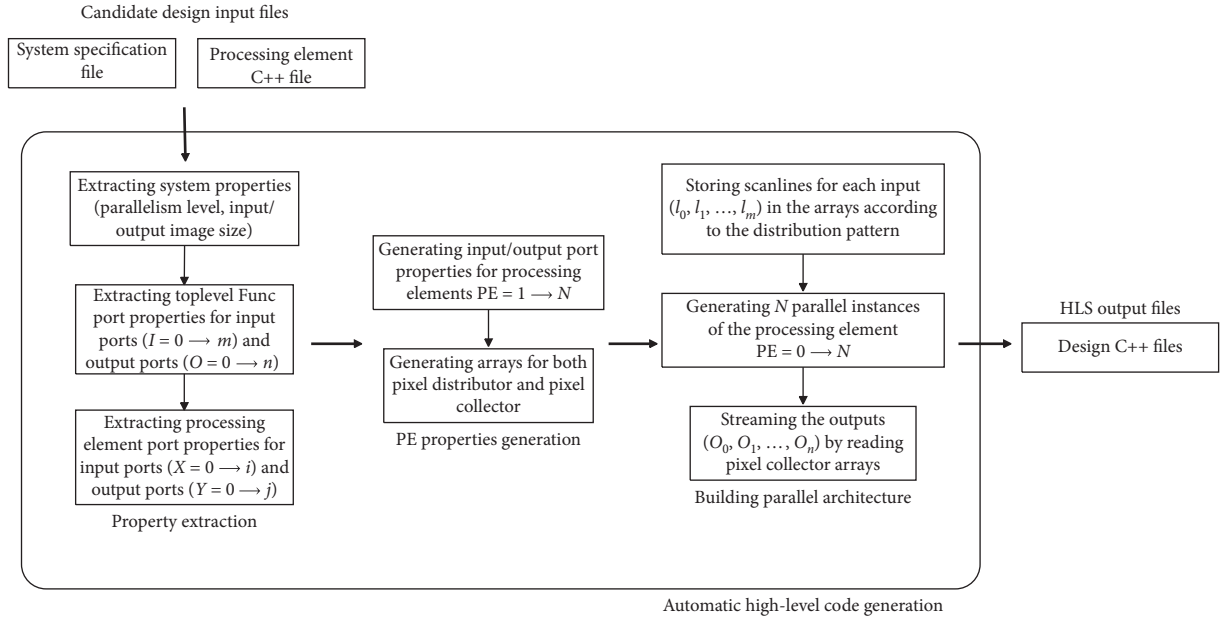

Figure 8: High-level code generation design flow.

implements the functionality of the video processing algorithm and (2) *System Specification File* which represents the properties of the system architecture constructed in Figure 2. The *Specification File* has four main sections: (1) header section, (2) system property section, (3) top-level function section, and (4) processing element section. In the *Processing Element* section, we define only the parameters for the first processing element, and then subsequently the tool can automatically generate the port parameters 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 8 illustrates the three main phases to generate the high-level code for parallel video processing architecture.

(i) *Property Extraction Phase*. From the *System Specification File*, the tool can extract *level_of_parallelism* and other input/output port properties for both top-level and processing element blocks.

(ii) *PE Property 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.

(iii) *Building the Parallel Architecture*. Finally, the tool builds the parallel architecture by generating the 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, and (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.

Listing 2 shows an example of the specification file for video downscaler (4:1) for an input VGA image size (640×480). As described before, the specification file is subdivided into four main sections: *Header* section includes all header files and definitions (lines 1–12), *System Property* 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), and *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 3 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 to change only 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 the system specification file for the 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 elements, while for the convolution filter, this ratio is 1.3 (69:52) for one processing element and increases to 39 (2026:52) for 64 element architecture.

7. Experimental Results

In this section, our industrial case study 5-window SAD algorithm will be explored by means of our developed ViPar tool. As a first step for exploring the design space, the processing element was implemented efficiently in terms of hardware utilization and frame rate by adding high-level synthesis optimizations. As discussed in Section 5, we obtained three different implementations by exploiting pipeline-level parallelism. 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 at different operating frequencies (100, 150, and 200 MHz) by varying the level of parallelism till full hardware utilization. Our experiments were based on Zynq XC7Z045-FFG900 Xilinx evaluation board and 5-window SAD configured for $winH = 23$, $winV = 7$, $cwinH = 7$, $cwinV = 3$, and maximum disparity = 64. It is worth to mention that changing the 5-window SAD configuration or the hardware platform will result in a new design space that needs to be explored.

By the help of ViPar tool, we will estimate the hardware utilization, power consumption, and execution time for each alternative in the design space. According to the system constraints, only the candidate designs will be selected for synthesis. We highlight that the high-level codes of the candidate designs will be generated automatically, and then synthesized to give the corresponding RTL design. The RTL design is then implemented and experimented to verify the estimated design metrics. Resource utilization, power, and frame execution time were estimated by means of the derived equations detailed in the previous section. Tables 4 and 5 list the points in the design space. The number of disparity lines processed by the same processing channel classifies the points into three groups (*pipe4*, *pipe8*, and *pipe12*). For each group, the same design was implemented at three different operating frequencies (100, 150, and 200 MHz); then, we can exploit data-level parallelism by increasing the parallelism level till full hardware utilization.

7.1. Area and Execution Time Estimations. The designs for *pipe4*, *pipe8*, and *pipe12* at parallelism level = 1 running 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.

The 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 constraints, then they are replaced by *Performance Explore* strategies (*Performance Explore* was 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 (nonsynthesized designs are #23, #24, #36, and #45).

Figure 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 nonsynthesized designs (#23, #24, #36, and #45). The percentage estimation error ranges between −21% and 0.4%, −3.7% and 0.3%, and −14.6% and 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 for synthesis while estimations were based on the default strategies). For BRAM, the estimation error was not plotted since both the estimated and measured values were identical. Figure 10 shows the percentage error in the estimated frame execution time where it ranges between −10.4% and 4.3% for different designs. This error arose due to the additional time consumed to set the DMA communication between the processing system (PS) and the programmable logic (PL).

7.2. Power Estimation. For fast power estimations at high-level design, only information about frequency and resource utilization are available. Figure 11 shows that the power consumption was underestimated by values ranging between 34% and 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 in Figure 12. From the figure, we can deduce that the two curves behave in the same manner. In conclusion, the derived power model can be used for relative power comparison between alternative designs during the design space exploration process. However, it cannot be used to estimate a value near from the real measurements for a single design due to the lack of full design implementation details.

7.3. Design Space Exploration. All design variations listed in Tables 4 and 5 could be accepted as a solution but the applied system constraints will direct our final decision to choose one design among the others. Figure 13 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 area represents the system constraints defined by the designer which are frame

```
(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 ##
```

Algorithm 2: Specification file for video downscaler (4:1) for input VGA image.

execution time ≤ 15 ms, LUT ≤ 150000, FF ≤ 120000, BRAM ≤ 700, and frequency ≤ 150 MHz. From Figure 13, 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, and BRAM = 275) and acceptable execution time (15.6 ms) compared 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 of design #43 or to stuck to them.

Table 3: Number of code lines generated for different applications at different parallelism levels.

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 4: Estimations for utilization, power, and frame execution time for *pipe4* designs.

#	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: Estimations for utilization, power, and frame execution time for *pipe8* and *pipe12* designs.

#		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

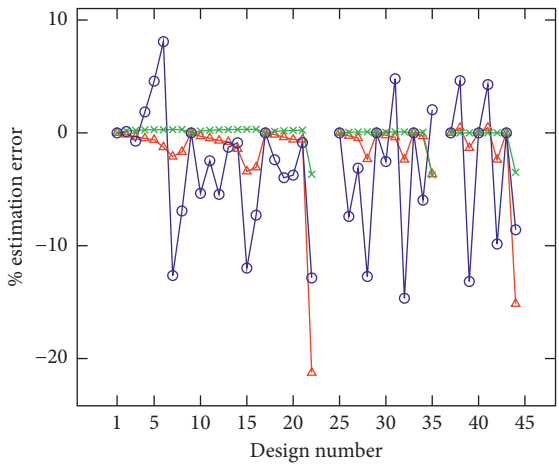

Figure 9: The estimation percentage error for slice, LUT, and FF when compared to the measured values.

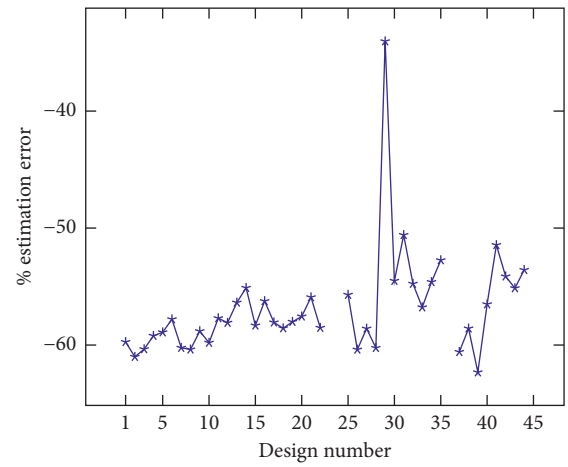

Figure 11: The estimation percentage error for power consumption when compared to the measured values.

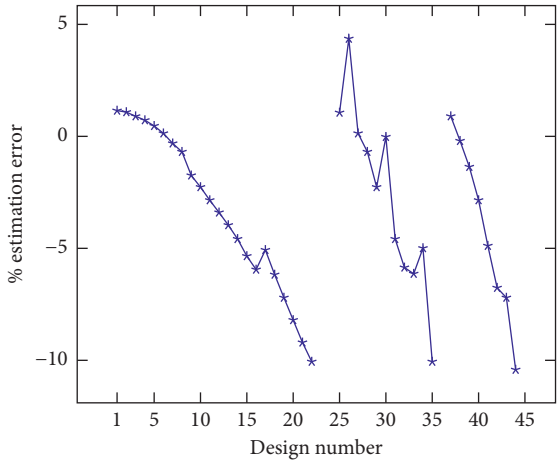

Figure 10: The estimation percentage error for frame execution time when compared to the measured values.

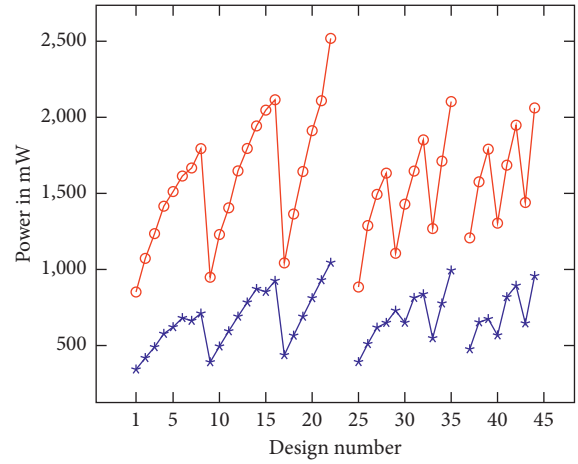

Figure 12: Estimated and measured power for different designs.

7.4. Comparison. Table 6 compares between our tool ViPar and other tools in the literature that are used for high-level exploration for video processing applications. In very high-level synthesis tool (VHLS) [37], algorithms are described in Matlab or OpenCL. The synthesis of Matlab-to-RTL is explored to change the vector-oriented source code into the scalar-oriented program. While the intermediate code is represented by using control and data-flow graphs (CDFG). Finally, the generated code is classically synthesized via control and data-flow extraction and RTL generation processes. Two applications were tested: Kubelka–Munk genetic algorithm (KMGA) [39] for the multispectral image-based skin lesion assessments and level set method (LSM) [40] for very high-resolution satellite image segmentation. The experimental results showed that the design complexity for VHLS version is 50% less than its equivalent in C code. Algorithms in Lin-Analyzer [27] are written in C/C++. It explores the design space of the application when different high-level optimizations are added like loop unrolling, loop pipelining, array partitioning, and so on. For each design point, the FPGA performance metrics are estimated. The

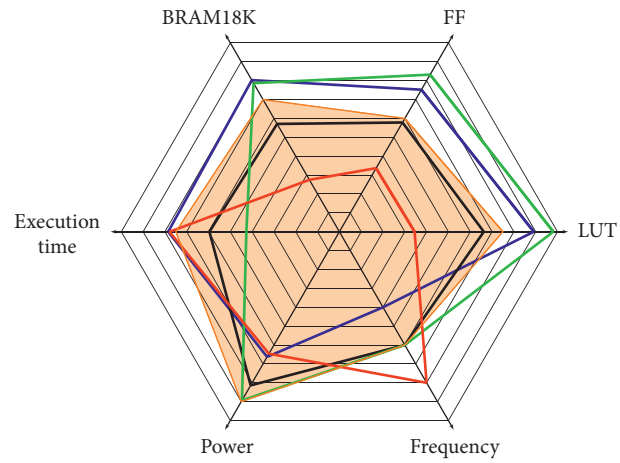

Figure 13: Radar chart for designs #7, #31, #42, #43, and system constraints.

required time for exploration ranges from seconds to minutes. In [38], algorithms are represented as dataflow graphs written in RVC-CAL language [41]. The graph nodes

Table 6: Comparing between different high-level tools for video processing application.

	VHLS [37]	Lin-Analyzer [27]	RVC [38]	ViPar
Programming language	Matlab or OpenCL	C/C++	RVC-CAL language	C/C++
Exploration level	Exploring Matlab-to-RTL synthesization	Exploring the design space for different optimization directives like loop unrolling, pipelining, ...	Exploring data-flow algorithms to C-based synthesization	Exploring the design space for parallelism level, operating frequency, power consumption, hardware cost, ...
Applications	KMGA LSM	Convolution 3D	HEVC decoder	Multiwindow SAD

are represented by the computational units that communicate concurrently, while the arcs represent the data flowing as tokens along unbounded FIFO channels. The dataflow-based video processing algorithm is then compiled by Open RVC-CAL compiler to generate a C-based code, which is then fed to a high-level synthesis (HLS) tool for generating a synthesizable hardware implementation. HEVC decoder was considered as the case study in [38].

Table 7 compares between the HLS implementation generated by ViPar tool versus handwritten VHDL implementation for two applications: video downscaler (16:1) and convolution filter of kernel size $= 3 \times 3$. For video downscaler, 6 parallel processing elements were implemented, where 2 PEs were located for each colour channel. While for the convolution filter, 6 PEs were dedicated for each colour channel with a total number of 18 PEs. From the synthesis results, the hardware cost is almost similar for both HLS and handwritten VHDL implementations with difference range between 3 and50% for different hardware resources.

Table 7: Comparing HLS and hand-written VHDL implementations for video downscaler (16:1) and convolution filter.

	FF	LUT	BRAM_18K
HLS_video_downscaler	1362	2022	15
VHDL_video_downscaler	2052	1753	12
HLS_convolution_filter	4968	5958	9
VHDL_convolution_filter	5136	3925	9

8. Conclusion and Future Works

In this paper, we presented ViPar as a tool for high-level design space exploration dedicated for video applications. First, we introduced a parameterizable model for describing parallel video processing architectures. Second, high-level optimizations were used to obtain an efficient hardware processing element in terms of hardware utilization and frame rate. Third, ViPar tool was used to explore rapidly the design space at high design level before going into the step of detailed implementation. In order to compare between different design points in ViPar tool, we derived the required equations for estimating the power consumption, hardware utilization, and frame execution time. At the last step, by describing the parallel hardware architecture of the candidate designs in the specification file, ViPar tool was able to generate automatically the corresponding parallel architecture for synthesis and experimental evaluation. In the experimental results, 5-window SAD stereo matching was explored as our industrial case study. The design space consisted of different points varying in parallelism level, operating frequency (100, 150, or 200 MHz), and the processing pipeline (*pipe4*, *pipe8*, *or pipe12*). Throughout this example, we demonstrated how ViPar tool could explore the design space for the candidate designs which meet our system constraints. ViPar tool estimates the performance parameters at high design level as well as generating automatically their corresponding parallel architectures for hardware implementation.

As future works, we will extend ViPar tool to consider the case of multiapplication design space exploration. In this situation, we have multiapplication multiobjective 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. Self-adaptivity for video applications like in autonomous vehicles is another case where some image filters could replace each other to adapt to the environmental changes. ViPar can be used to explore the possible hardware architectures to satisfy these scenarios of filter replacements.

Data Availability

The data used to support the findings of this study are available from the corresponding author upon request.

Disclosure

An earlier version of this work has been presented as a thesis in Université de Valenciennes et du Hainaut-Cambresis at the following link: https://tel.archives-ouvertes.fr/tel-01791649/document.

Conflicts of Interest

The authors declare that there are no conflicts of interest regarding the publication of this paper.

References

[1] M. Zablocki, K. Gosciewska, D. Frejlichowski, and R. Hofman, "Intelligent video surveillance systems for public spaces—a survey," *Journal of Theoretical and Applied Computer Science*, vol. 8, no. 4, pp. 13–27, 2014.

[2] 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, vol. 8, no. 10, pp. 1590–1599, 2013.

[3] S. Vishwakarma and A. Agrawal, "A survey on activity recognition and behavior understanding in video surveillance," *The Visual Computer*, vol. 29, no. 10, pp. 983–1009, 2013.

[4] W. Jiang, C. Xiao, H. Jin, S. Zhu, and Z. Lu, "Vehicle tracking with non-overlapping views for multi-camera surveillance system," in *Proceedings of the 2013 IEEE 10th International Conference on High Performance Computing and Communications 2013 IEEE International Conference on Embedded and Ubiquitous Computing*, pp. 1213–1220, IEEE, Zhangjiajie, China, November 2013.

[5] 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*, vol. 16, no. 3, pp. 1107–1119, 2015.

[6] 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*, vol. 12, no. 4, pp. 1624–1639, 2011.

[7] 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*, vol. 33, no. 2, pp. 36–44, 2016.

[8] S. Fleck and W. Straber, "Smart camera based monitoring system and its application to assisted living," *Proceedings of the IEEE*, vol. 96, no. 10, pp. 1698–1714, 2008.

[9] L. Bampis and A. Gasteratos, "Revisiting the bag-of-visual-words model: a hierarchical localization architecture for mobile systems," *Robotics and Autonomous Systems*, vol. 113, pp. 104–119, 2019.

[10] G. Grisetti, R. Kummerle, C. Stachniss, and W. Burgard, "A tutorial on graph-based slam," *IEEE Intelligent Transportation Systems Magazine*, vol. 2, no. 4, pp. 31–43, 2010.

[11] A. Amanatiadis, E. Karakasis, L. Bampis, S. Ploumpis, and A. Gasteratos, "Viped: On-road vehicle passenger detection for autonomous vehicles," *Robotics and Autonomous Systems*, vol. 112, pp. 282–290, 2019.

[12] C. Patruno, R. Marani, M. Nitti, T. D'Orazio, and E. Stella, "An embedded vision system for real-time autonomous localization using laser profilometry," *IEEE Transactions on Intelligent Transportation Systems*, vol. 16, no. 6, pp. 3482–3495, 2015.

[13] M. Omidyeganeh, S. Shirmohammadi, S. Abtahi et al., "Yawning detection using embedded smart cameras," *IEEE Transactions on Instrumentation and Measurement*, vol. 65, no. 3, pp. 570–582, 2016.

[14] P. Cooke, J. Fowers, G. Brown, and G. Stitt, "A tradeoff analysis of FPGAs, GPUs, and multicores for sliding-window applications," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 8, no. 1, pp. 1–24, 2015.

[15] A. Severance and G. G. F. Lemieux, "Embedded supercomputing in FPGAs with the VectorBlox MXP matrix processor," in *Proceedings of the 2013 International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp. 1–10, IEEE, Montreal, QC, Canada, September 2013.

[16] G. Hegde and N. Kapre, "Energy-efficient acceleration of openCV saliency computation using soft vector processors," in *Proceedings of the 2015 IEEE 23rd Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 76–83, IEEE, Vancouver, BC, Canada, May 2015.

[17] 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*, vol. 9, no. 1, pp. 47–59, 2014.

[18] K. Andryc, M. Merchant, and R. Tessier, "FlexGrip: a soft GPGPU for FPGAs," in *Proceedings of the 2013 International Conference on Field-Programmable Technology (FPT)*, pp. 230–237, Kyoto, Japan, December 2013.

[19] 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*, pp. 254–263, ACM, New York, NY, USA, 2016.

[20] M. A. Kadi and M. Huebner, "Integer computations with soft GPGPU on FPGAs," in *Proceedings of the 2016 International Conference on Field-Programmable Technology (FPT)*, pp. 28–35, IEEE, Xi'an, China, December 2016.

[21] 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*, vol. 60, no. 1, pp. 79–93, 2014.

[22] A. Sengupta and R. Sedaghat, "Integrated scheduling, allocation and binding in high level synthesis using multi structure genetic algorithm based design space exploration," in *Proceedings of the 2011 12th International Symposium on Quality Electronic Design*, pp. 1–9, IEEE, Santa Clara, CA, USA, March 2011.

[23] B. C. Schafer, "Probabilistic multiknob high-level synthesis design space exploration acceleration," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35, no. 3, pp. 394–406, 2016.

[24] 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*, vol. 36, no. 1, pp. 97–105, 2017.

[25] 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 *Proceedings of the 2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pp. 157–162, IEEE, Grenoble, France, March 2015.

[26] C. Lo and P. Chow, "Model-based optimization of high level synthesis directives," in *Proceedings of the 2016 26th International Conference on Field Programmable Logic and Applications (FPL)*, pp. 1–10, IEEE, Lausanne, Switzerland, August 2016.

[27] G. Zhong, A. Prakash, Y. Liang, T. Mitra, and S. Niar, "Lin-Analyzer: a high-level performance analysis tool for FPGA-based accelerators," in *Proceedings of the 2016 53rd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pp. 1–6, IEEE, Austin, TX, USA, June 2016.

[28] X. Gao and T. Yoshimura, "Genetic algorithm based pipeline scheduling in high-level synthesis," in *Proceedings of the 2013 IEEE 10th International Conference on ASIC*, pp. 1–4, IEEE, Shenzhen, China, October 2013.

[29] 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)*, pp. 1–6, IEEE, San Francisco, CA, USA, May 2014.

[30] H.-Y. Liu and L. P. Carloni, "On learning-based methods for design-space exploration with high-level synthesis," in *Proceedings of the 2013 50th ACM/EDAC/IEEE Design*

Automation Conference (DAC), pp. 1–7, Austin, TX, USA, May 2013.

[31] D. Scharstein and R. Szeliski, "A taxonomy and evaluation of dense two-frame stereo correspondence algorithms," *International Journal of Computer Vision*, vol. 47, no. 1–3, pp. 7–42, 2002.

[32] H. Hirschmuller, "Improvements in real-time correlation-based stereo vision," in *Proceedings of the IEEE Workshop on Stereo and Multi-Baseline Vision (SMBV 2001)*, pp. 141–148, IEEE, Kauai, HI, USA, December 2001.

[33] M. J. McDonnell, "Box-filtering techniques," *Computer Graphics and Image Processing*, vol. 17, no. 1, pp. 65–70, 1981.

[34] K. M. A. Ali, R. Ben Atitallah, S. Hanafi, and J.-L. Dekeyser, "A generic pixel distribution architecture for parallel video processing," in *Proceedings of the 2014 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–8, Cancun, Mexico, December 2014.

[35] K. M. Ali, R. B. Atitallah, N. Fakhfakh, and J.-L. Dekeyser, "Exploring hls optimizations for efficient stereo matching hardware implementation," in *Proceedings of the International Symposium on Applied Reconfigurable Computing*, pp. 168–176, Springer, Delft, The Netherlands, April 2017.

[36] F. Sun, H. Wang, F. Fu, and X. Li, "Survey of FPGA low power design," in *Proceedings of the 2010 International Conference on Intelligent Control and Information Processing*, pp. 547–550, IEEE, Dalian, China, August 2010.

[37] 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*, pp. 160–165, ACM, New York, NY, USA, 2016.

[38] M. Abid, K. Jerbi, M. Raulet, O. Déforges, and M. Abid, "Efficient system-level hardware synthesis of dataflow programs using shared memory based fifo," *Journal of Signal Processing Systems*, vol. 90, no. 1, pp. 127–144, 2018.

[39] R. Jolivot, Y. Benezeth, and F. Marzani, "Skin parameter map retrieval from a dedicated multispectral imaging system applied to dermatology/cosmetology," *Journal of Biomedical Imaging*, vol. 2013, Article ID 978289, 15 pages, 2013.

[40] S. Balla-Arabe, X. Gao, B. Wang, F. Yang, and V. Brost, "Multi-kernel implicit curve evolution for selected texture region segmentation in vhr satellite images," *IEEE Transactions on Geoscience and Remote Sensing*, vol. 52, no. 8, pp. 5183–5192, 2014.

[41] J. Eker and J. W. Janneck, *Janneck J: Cal Language Report: Specification of the Cal Actor Language*, University of California, Berkeley, CA, USA, 2003.