

Exploring HLS Optimizations for Efficient Stereo Matching Hardware Implementation

Karim M.A. Ali¹(✉), Rabie Ben Atitallah¹, Nizar Fakhfakh²,
and Jean-Luc Dekeyser³

¹ LAMIH, University of Valenciennes, Valenciennes, France
{karim.ali,rabie.benatitallah}@univ-valenciennes.fr

² NAVYA Company, Paris, France
nizar.fakhfakh@navya-technology.com

³ CRISAL, University of Lille1, Lille, France
jean-luc.dekeyser@univ-lille1.fr

Abstract. Nowadays, FPGA technology offers a tremendous number of logic cells on a single chip. Digital design for such huge hardware resources under time-to-market constraint urged the evolution of High Level Synthesis (HLS) tools. In this work, we will explore several HLS optimization steps in order to improve the system performance. Different design choices are obtained from our exploration such that an efficient implementation is selected based on given system constraints (resource utilization, power consumption, execution time, ...). Our exploration methodology is illustrated through a case study considering a Multi-Window Sum of Absolute Difference stereo matching algorithm. We implemented our design using Xilinx Zynq ZC706 FPGA evaluation board for gray images of size 640×480 .

Keywords: FPGA · High level synthesis · Stereo matching algorithms

1 Introduction

FPGA circuits have emerged as a privileged target platforms to implement intensive signal processing applications [3]. For this reason several academic and industrial efforts have been devoted in order to increase the productivity of FPGA-based designs by means of using High Level Synthesis (HLS) tools. HLS approach in Electronic Design Automation (EDA) is a step in the design flow aiming at moving the design effort to higher abstraction levels [6]. This evolution towards HLS-based methodologies can be easily traced along the history of hardware system design [2]. Although the first generations of HLS tools failed to produce efficient hardware designs, different reasons have motivated researchers to continue improving these tools. We can mention among these reasons: 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,

etc [1]. Today several existing HLS tools have shown their efficiency for producing acceptable design performances and shortening time-to-market [6, 8].

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 important 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 them. In this research work, the design space was built by applying a set of high level synthesis optimization steps. The obtained designs have different trade-offs in terms of hardware resources (FF, LUT or BRAM), power consumption, timing and operating frequency. Our objective is to explore the possible hardware designs then choose the one that most fit with our requirements. As a case study, we focus on the development of an FPGA-based system dedicated to streaming stereo matching applications. Our application considers Multi-Window Sum of Absolute Difference (Multi-Window SAD) algorithm [4] performed on input gray images of size 640×480 with maximum disparity = 64.

As a similar work targeting stereo matching domain, authors in [9] examined five stereo matching algorithms for their HLS implementation. Five optimization steps were applied to the SW code: baseline implementation, code restructuring, bit-width reduction, pipelining and parallelization via resource duplication. Our work differs from that presented in [9] as follows: (i) Baseline implementation is considered as step zero in our work because our input code is HLS-friendly. (ii) Dividing an image into strips can be achieved in three different ways with vast difference in terms of execution time and resource utilization (Optimization #1). (iii) Parallelism was exploited in both work at different levels. In our work, data-independent loops are executed in parallel by duplicating the input data stream (Optimization #3). We also increased the number of processed disparity lines at the same time either by enlarging the size of strip (Optimization #7) or by duplicating the top-level function (Optimization #8). While authors in [9] applied parallelism only by duplicating the disparity computation pipeline. Authors in [7] purposed an optimized C-code for Sobel filter in three steps. Although the design run on Zynq platform; no details were mentioned on how the HLS-based Sobel filter was interfaced and connected to the system. In this work, we will detail this point in Sect. 4. In addition to that two more optimization steps related to Zynq platforms are presented in Sect. 3 (Optimization #5 and #6).

The rest of this paper is organized as follows: Sect. 2 describes our case study related to Multi-Window SAD stereo matching algorithm. Section 3 represents our main contribution that explores high level optimization steps for an efficient implementation for our case study. System architecture and experimental results are presented in Sect. 4.

2 Multi-window SAD Matching Algorithm

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 1a shows how the depth of objects is calculated in 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). Therefore; the depth of pixel (P) from the two camera can be calculated from the following equation:

$$\text{depth} = \frac{\text{baseline} * \text{focal length}}{\text{disparity}} = \frac{b * f}{(X_R - X_L)} \tag{1}$$

Several methods in the literature were proposed to find the best matching [10]. In Multi-Window SAD [4], the absolute difference between pixels from the right and left images are aggregated within a window. 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 non-errored parts are considered in calculations. Figure 1b shows 5-window SAD configuration: pixel (P) lies in the middle of window (E) while it is surrounded by another four windows named (A, B, C and D). The four windows are partially overlapped at the border pixel (P). The score of any window is equal to the aggregation of its pixels. In 5-window SAD, the correlation score at pixel (P) is equal to the score value of window (E) in addition to the best minimum two score values of the other four windows (A, B, C and D). The minimum score among the candidates is considered as the best matching. Occluded objects are common to happen in stereo matching problem where sometimes the objects are only captured by

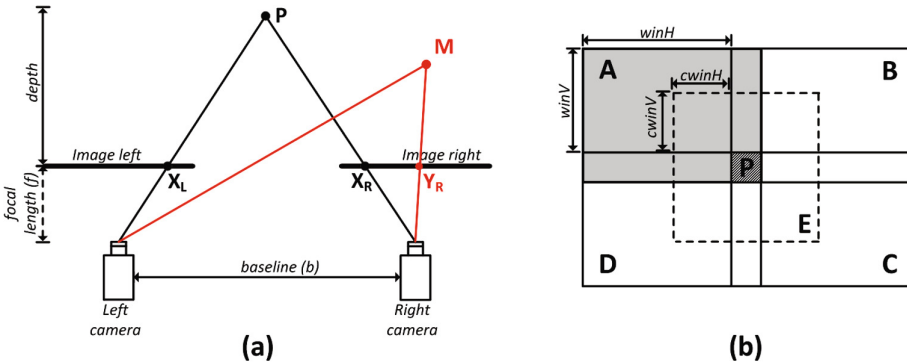


Fig. 1. (a) Calculating the depth of an object in stereo matching problem (b) 5-window SAD configuration

one camera. For example, pixel (M) in Fig. 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.

3 High-level Synthesis Optimizations

In this section, we are going to explore the possible optimization steps that could be done in order to achieve an efficient hardware implementation. The C code was written in HLS-friendly syntax with neither file read/write, nor dynamic memory allocation nor system calls. The optimization steps are incrementally applied to the design as listed in Table 1. From our point of view, a fair comparison between designs is valid only for adjacent rows in order to observe the impact of adding this optimization to the overall design performance. The SW code was synthesized by Vivado HLS to obtain the first synthesizable design (Design #1). Table 1 shows that Design #1 had an overuse for BRAM (BRAM_18K=7392) while using Xilinx Zynq ZC706 platform of maximum BRAM_18K=1090. This will lead to the first optimization step which is dividing an image into strips during processing in order to reduce the required memory usage.

Optimization #1: Dividing an image into strips. In strip processing, the code will be repetitively executed until one frame is completely processed. The pixels can be summed in three different ways: (i) Design #2 aggregates the pixels in the horizontal direction along the scanline then in the vertical one. (ii) While aggregation is done vertically along the column length then horizontally along the scanline for Design #3. (iii) However in Design #4, the pixels are aggregated within one window then box-filtering technique [5] is applied to get the summation of other windows along the horizontal direction. 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 execution time (it was improved by 73% of that reported for Design #2). Therefore, we will consider Design #4 as a base for the next optimization steps.

Optimization #2: Using arbitrary precision data types. Vivado HLS supports arbitrary precision data types to define variables with smaller bit width. Using this optimization will produce systems of the same accuracy but with less area utilization. A complete analysis was done to know exactly the required number of bits for each variable. In Table 1, Design #5 showed around 31% reduction for LUT and 40% reduction for FF after applying optimization #2.

Optimization #3: Executing data-independent loops in parallel. Along the same scanline, the score for window (B) is used after ($winH+1$) pixel shift as a score for a new window (A). It is also the same case for windows (C) and (D). Therefore, only three score calculation loops are needed for windows (A/B, C/D and E). By duplicating the input data stream, the three loops can run in parallel. In Table 1, Design #6 reported the effect of optimization #3.

Optimization #4: Using HLS optimization directives. Using HLS optimization directives to tune the design performance is one of the fundamental

Table 1. Synthesis results reported by Vivado HLS for each optimization step

| Design | Slice | FF | LUT | BRAM 18K | SRL | Freq. (MHz) | Exec. time (ms) | % change in Perf. |
|------------|---|-------|-------|----------|-----|-------------|-----------------|-------------------|
| SW version | 380 ms on core i7@ 2.7 GHz and 16 GB of RAM | | | | | | | |
| #1 | X | 2637 | 5918 | 7392 | 0 | 100 | X | X |
| #2 | 898 | 1743 | 2735 | 155 | 0 | 100 | 30080 | 0 |
| #3 | 859 | 1758 | 2659 | 113 | 0 | 100 | 22410 | 25.4 |
| #4 | 1400 | 2552 | 3738 | 75 | 0 | 100 | 8163 | 72.8 |
| #5 | 983 | 1525 | 2567 | 47 | 0 | 100 | 5786 | 29.1 |
| #6 | 985 | 1713 | 2768 | 65 | 0 | 100 | 2679 | 53.7 |
| #7 | 2695 | 6088 | 7611 | 57 | 0 | 100 | 328 | 87.7 |
| #8 | 2688 | 6134 | 7661 | 59 | 0 | 100 | 331 | -9.1 |
| #9 | 2822 | 6365 | 8116 | 59 | 0 | 100 | 307 | 7.2 |
| #10 | 7989 | 20256 | 24433 | 112 | 0 | 100 | 76 | 75.2 |
| #11 | 7995 | 18765 | 24945 | 112 | 39 | 150 | 51 | 32.8 |
| #12 | 8038 | 21250 | 26483 | 112 | 121 | 200 | 38 | 25.5 |

optimization steps. We examined three types of directives that gave a crucial improvement in performance: (i) Arrays were partitioned either into smaller arrays (partial) or as individual registers (complete) in order to boost the system throughput by increasing the number of available read/write ports. (ii) Loops were unrolled by factor = 2 to make profit from the existed physical dual-port for arrays implemented as BRAMs. (iii) Loops were pipelined with Initiation Interval (II) = 1 to enhance the system performance. In Table 1, Design #7 listed the estimated hardware resources and execution time after using optimization #4.

Optimization #5: Choosing I/O interface protocol for the top-level function. *HLS_SAD* core is synthesized for Zynq platform where pixels flow through DMA-based connections as shown in Fig. 2. We chose AXI-Stream for I/O ports while AXI-Lite was chosen for controlling the hardware core. AXI-Stream defined by Vivado HLS comes only with the fundamental signals (*TDATA*, *TREADY*, *TVALID*) but for DMA communications, *TLAST* signal is also needed. Therefore, Design #8 was modified such that the output port also includes a *TLAST* signal with 9% decrease in performance as listed in Table 1.

Optimization #6: Grouping pixels at I/O ports for DMA-based communication. Zynq platform has four High Performance bus (HP bus) between Processing System (PS) and Programmable Logic (PL) of 64-bit data width. The designer can benefit from this data width by merging pixels at I/O ports. In our design, the input pixel is 32-bit width while the output disparity pixel 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. This data merging requires an additional attention from the designer while separating the pixels at the input or merging them at the output. Design #9 showed 7% improvement in the execution time.

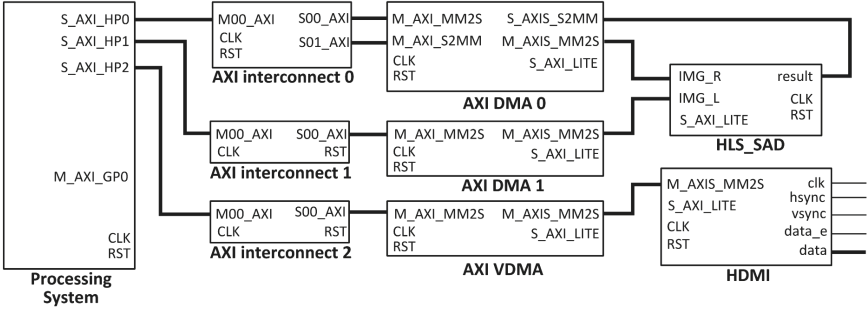


Fig. 2. System architecture block diagram

Optimization #7: Enlarging the size of strip. During strip processing, there is only one scanline difference between two strips when processing two adjacent disparity lines. From Fig. 1b, one disparity line needs a strip of size $= 2 * win_V + 1$ while four adjacent disparity lines need a strip of size $= 2 * win_V + 4$. In Design #10, four disparity lines are calculated using the same pipeline such that the execution time is reduced to the quarter (Table 1).

Optimization #8: Duplicating the top-level function. In this optimization step, we run multiple instances of Design #10 in parallel. Simply, we defined a new top-level function that contains multiple instances of the function defined in Design #10. In the experimental results, we will explore designs of 5, 6, 7 or 8 instances running in parallel at frequencies of 100, 150 or 200 MHz.

4 Experimental Results

The generated *HLS_SAD* IP was tested experimentally to validate both its proper functioning and the estimated results. During our experiments, we used Vivado 2015.2 design suite to implement our system over Zynq ZC706 FPGA evaluation board (XC7Z045-FFG900) with input grey images of size 640×480 . The system was configured for 5-window SAD with the following parameters: $winH = 23$, $winV = 7$, $cwinH = 7$, $cwinV = 3$ and maximum disparity = 64.

Figure 2 illustrates the connection of *HLS_SAD* core to the other cores in the system. Pixels were transferred between the processing system (*PS*) and *HLS_SAD* block through two *AXI DMA* cores. *AXI VDMA* and *HDMI* cores were used to display the obtained disparity image on the output screen.

We obtained different design choices by exploring the effect of optimization #8 at different operating frequencies of 100, 150 or 200 MHz as listed in Table 2. During the experiments, we increased the level of parallelism up to 8 instances operating at the same time. We stopped at that level due to the limited LUT resources (design #23 consumed 95.37% of LUT). Default synthesis and implementation strategies were used by default for all designs. For design #18, *Flow_Perf_Optimized_High* and *Performance_Explore* were used as synthesis and

Table 2. Synthesis results for designs at different levels of parallelism

| Design | Level of parallelism | Slice (54650) | FF (437200) | LUT (218600) | BRAM 18K (1090) | Freq. (MHz) | Frame exec. time (ms) | Power (W) | Energy (mJ) |
|--------|----------------------|--|-------------|--------------|-----------------|-------------|-----------------------|-----------|-------------|
| #10 | 1 | 10534 | 28903 | 31163 | 131 | 100 | 83.91 | 0.852 | 71.49 |
| #11 | 1 | 10111 | 27410 | 31140 | 131 | 150 | 57.74 | 0.949 | 54.80 |
| #12 | 1 | 9642 | 29895 | 31184 | 131 | 200 | 45.02 | 1.043 | 46.96 |
| #13 | 5 | 40326 | 109624 | 130262 | 579 | 100 | 20.56 | 1.512 | 31.09 |
| #14 | 5 | 41208 | 102174 | 130571 | 579 | 150 | 14.37 | 1.795 | 25.79 |
| #15 | 5 | 38980 | 114617 | 130917 | 579 | 200 | 11.44 | 2.109 | 24.13 |
| #16 | 6 | 46337 | 129822 | 155728 | 691 | 100 | 17.83 | 1.612 | 28.74 |
| #17 | 6 | 48752 | 120873 | 155822 | 691 | 150 | 12.5 | 1.943 | 24.29 |
| #18 | 6 | 52670 | 141335 | 195408 | 691 | 200 | 9.98 | 2.519 | 25.14 |
| #19 | 7 | 51022 | 150015 | 182108 | 803 | 100 | 15.75 | 1.667 | 26.26 |
| #20 | 7 | 50592 | 139557 | 184409 | 803 | 150 | 11.08 | 2.047 | 22.68 |
| #21 | 7 | Timing constraints are not met @ 200 MHz | | | | | | | |
| #22 | 8 | 54470 | 170195 | 206273 | 915 | 100 | 14.42 | 1.794 | 25.87 |
| #23 | 8 | 54636 | 158259 | 208993 | 915 | 150 | 10.17 | 2.115 | 21.51 |
| #24 | 8 | Timing constraints are not met @ 200 MHz | | | | | | | |

implementation strategies respectively to meet the time constraints. For designs #21 and #24, although we tried several strategies, the tool failed to meet the time constraints for an operating frequency of 200 MHz.

The frame execution time was firstly estimated by Vivado HLS as shown in Table 1 then it was measured experimentally as listed in Table 2. For all designs, we could notice that Vivado HLS underestimated the frame execution time with an error range between 10–30%. The reason for this underestimation is that Vivado HLS did not consider the time spent for DMA communication while pixels are transferred from/to *HLS_SAD* core. Table 2 lists the required hardware resources to realize the system architecture depicted in Fig. 2. We could notice that at the same level of parallelism, changing the operating frequency led to different numbers for FF and LUT in order to satisfy the timing constraints. For example, in comparison with design #16, FF decreased by 6.9% and increased by 8.9% for designs #17 and #18 respectively while LUT was almost unchanged in design #17 and increased by 25% for design #18.

The power consumption was measured experimentally through UCD90120A power controller mounted on Zynq ZC706 FPGA board. Two factors mainly contribute to the power consumption: the used hardware resources and the operating frequency. Design #18 showed the maximum power consumption of 2.52 W at 200 MHz. Although design #23 utilized more hardware resources, it showed 16% less in power consumption (2.12 W) since it operates at 150 MHz. Calculating energy consumption showed that some design points were more energy efficient

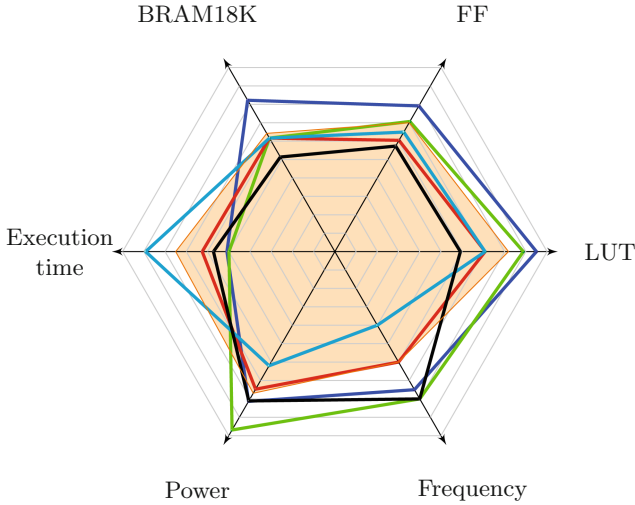


Fig. 3. Radar chart for designs #15 —, #16 —, #17 —, #18 —, #23 — and system constraints —. (Color figure online)

than others even if they consumed more power. For example, design #23 had the lowest energy consumption of 21.51 mJ although it recorded one of the highest power consumption (2.12 W).

All design variations listed in Table 2 could be accepted as a solution but the applied system constraints will direct our final decision to choose one design among the others. Figure 3 depicts some of the candidate designs (#15, #16, #17, #18 and #23) along with the system constraints to guide the designer towards the efficient solution. The orange shaded area represents the system constraints defined by the designer which are: power consumption ≤ 2 W, execution time ≤ 15 ms, LUT ≤ 180000 , FF ≤ 140000 , BRAM ≤ 700 and frequency ≤ 150 MHz. From Fig. 3, we could deduce that design #17 succeeded to satisfy all the system constraints. Design #15 had relatively less hardware utilization and acceptable execution time in compare with design #17; however, it failed to meet two design constraints (power consumption and frequency).

5 Conclusion

Using HLS tools for complex system design becomes mandatory to increase the design productivity and to shorten the time-to-market. As a future work, we will automatically explore designs at higher level of parallelism. In addition to that we will build a model to predict if that design is feasible or not for a given set of system constraints.

References

1. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: from prototyping to deployment. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst.* **30**(4), 473–491 (2011)
2. Coussy, P., Gajski, D.D., Meredith, M., Takach, A.: An introduction to high-level synthesis. *IEEE Des. Test Comput.* **26**(4), 8–17 (2009)
3. Gonzalez, I., El-Araby, E., Saha, P., El-Ghazawi, T., Simmler, H., Merchant, S.G., Holland, B.M., Reardon, C., George, A.D., Lam, H., Stitt, G., Alam, N., Smith, M.C.: Classification of application development for FPGA-based systems. In: *IEEE National Aerospace and Electronics Conference, NAECON 2008*, pp. 203–208, July 2008
4. Hirschmuller, H.: Improvements in real-time correlation-based stereo vision. In: *IEEE Workshop on Stereo and Multi-Baseline Vision, (SMBV 2001)*, Proceedings, pp. 141–148 (2001)
5. McDonnell, M.: Box-filtering techniques. *Comput. Graph. Image Process.* **17**(1), 65–70 (1981)
6. Meeus, W., Van Beeck, K., Goedemé, T., Meel, J., Stroobandt, D.: An overview of today's high-level synthesis tools. *Des. Autom. Embed. Syst.* **16**(3), 31–51 (2012)
7. Monson, J., Wirthlin, M., Hutchings, B.L.: Optimization techniques for a high level synthesis implementation of the Sobel filter. In: *2013 International Conference on Reconfigurable Computing and FPGAs (ReConFig)*, pp. 1–6, December 2013
8. Nane, R., Sima, V.M., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y.T., Hsiao, H., Brown, H., Ferrandi, F., Anderson, J., Bertels, K.: A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans. Comput.-Aided Des. Integr. Circ. Syst.* **35**(99), 1 (2016)
9. Rupnow, K., Liang, Y., Li, Y., Min, D., Do, M., Chen, D.: High level synthesis of stereo matching: productivity, performance, and software constraints. In: *2011 International Conference on Field-Programmable Technology (FPT)*, pp. 1–8, December 2011
10. Scharstein, D., Szeliski, R.: A taxonomy and evaluation of dense two-frame stereo correspondence algorithms. *Int. J. Comput. Vis.* **47**(1–3), 7–42 (2002)