FP-Stereo: Hardware-Efficient Stereo Vision for Embedded Applications

Jieru Zhao[1], Tingyuan Liang[1], Liang Feng[2], Wenchao Ding[1], Sharad Sinha[3], Wei Zhang[1] and Shaojie Shen[1]
[1]*Hong Kong University of Science and Technology*, [2]*Alibaba Group*, [3]*India Institute of Technology Goa*
{jzhaoao, tliang, lfengad, wdingae, wei.zhang, eeshaojie}@ust.hk, sharad_sinha@ieee.org

Abstract—**Fast and accurate depth estimation, or stereo matching, is essential in embedded stereo vision systems, requiring substantial design effort to achieve an appropriate balance among *accuracy*, *speed* and *hardware cost*. To reduce the design effort and achieve the right balance, we propose FP-Stereo for building high-performance stereo matching pipelines on FPGAs automatically. FP-Stereo consists of an open-source hardware-efficient library, allowing designers to obtain the desired implementation instantly. Diverse methods are supported in our library for each stage of the stereo matching pipeline and a series of techniques are developed to exploit the parallelism and reduce the resource overhead. To improve the usability, FP-Stereo can generate synthesizable C code of the FPGA accelerator with our optimized HLS templates automatically. To guide users for the right design choice meeting specific application requirements, detailed comparisons are performed on various configurations of our library to investigate the accuracy/speed/cost trade-off. Experimental results also show that FP-Stereo outperforms the state-of-the-art FPGA design from all aspects, including 6.08% lower error, 2x faster speed, 30% less resource usage and 40% less energy consumption. Compared to GPU designs, FP-Stereo achieves the same accuracy at a competitive speed while consuming much less energy.**

I. Introduction

Real-time and robust stereo vision systems for the computation of depth information are increasingly popular in many embedded applications including robotic navigation [1, 2] and autonomous vehicles [3, 4]. Stereo matching methods take a pair of left-right images from stereo cameras as input and generate the disparity map for depth estimation. Typical stereo matching algorithms can be classified into two categories: local and global methods. Local methods aggregate matching costs over a local region, achieving fast speeds but suffering from textureless and discontinuous regions [5, 6]. Global methods estimate the disparity map by minimizing a global energy function, using graph cuts [7] or belief propagation [8, 9], which achieve high accuracy but are computationally inefficient. Semi-global matching (SGM) approximates the global optimization by minimizing a pathwise energy function and aggregating matching costs in multiple directions instead of the whole image [10]. Due to the excellent trade-off between accuracy and speed, SGM has become one of the most widely used stereo matching algorithms, especially in real-world embedded applications.

Most existing SGM designs on FPGAs were implemented at register-transfer level (RTL) [11]–[17], and are hard for reproduction and modification. Relying on high-level synthesis (HLS), which automatically synthesizes C code into RTL designs, the development time can be shortened significantly. Several acceleration approaches [18]–[21] utilize HLS to map *local* stereo matching algorithms on FPGAs, achieving fast speed while incurring considerable accuracy loss. By comparison, SGM is more practical with higher accuracy. However, extra design effort is necessary to achieve real-time speeds on specialized hardware due to the huge pressure on hardware resources caused by intermediate computing results [11]. Rahnama et al. [22] implement an SGM variation on FPGA using HLS with a throughput of 72 frames per second (FPS) for 1242*375 image size and 128 disparity levels on the KITTI dataset [23]. Recently, Xilinx released xfOpenCV [24], and its SGM implementation achieves a faster speed of 81 FPS on KITTI images. Although raising the abstraction level, neither method [22, 24] can fully exploit the parallelism to achieve competitive speeds compared to RTL designs. Moreover, apart from *efficiency* concerns, high-quality stereo matching pipelines should achieve user-required *accuracy* given limited *hardware budgets*. However, the implementations in [22] and [24] only support a single method for each stage of the SGM pipeline, lacking the flexibility to adapt to various user requirements in *accuracy*, *speed* and *hardware cost*.

To this end, we propose FP-Stereo for building high-performance SGM pipelines on FPGAs automatically to meet different accuracy/speed/cost requirements. Our contributions are summarized as follows:

- We provide an open-source hardware-efficient library on FPGA, composed of optimized HLS C kernels, for embedded stereo vision applications.
- We apply effective optimization and implementation techniques to our library templates, fully exploiting the parallelism and greatly reducing the resource overhead.
- We present several methods to improve usability and scalability of our library, including auto-computed data widths, user-friendly interface and automatic code generation.
- We investigate how algorithmic choices, i.e., various functions or parameters, impact hardware performance, and how hardware budgets constrain algorithmic choices. Our findings can serve as guidance for users to select the right design choice which achieves an appropriate accuracy/speed/resource balance on FPGA.
- FP-Stereo supports high-quality SGM implementations. Compared to xfOpenCV, FP-Stereo achieves higher accuracy (6.08% lower error) at 2x speed (161 FPS) while

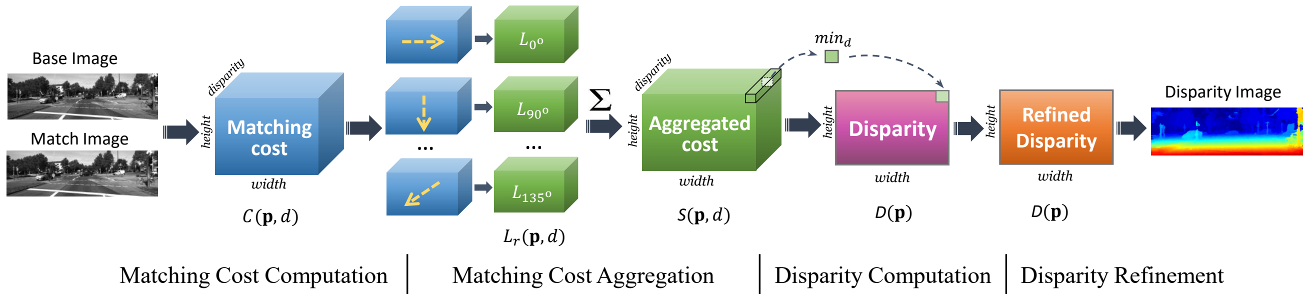


Fig. 1: Diagram of the semi-global matching algorithm.

consuming 30% fewer resources and 40% less energy at full KITTI resolution. Compared to GPU designs, FP-Stereo achieves the best speed/power ratio for the same accuracy.

II. Pipeline of Semi-Global Matching Algorithm

Figure 1 illustrates a high-level overview of the semi-global matching algorithm, which contains four stages: (1) similarity comparison is applied on the base (left) and match (right) images to compute the matching cost for each pixel at each disparity, generating a matching cost volume; (2) to smooth the cost volume, matching costs of neighboring pixels at each disparity are aggregated in different directions; (3) the disparity map of the base image is computed from the aggregated cost volume; and (4) several refinement steps are applied to remove outliers. In this section, we will introduce frequently used methods for each stage of the stereo matching algorithm. All of them are supported by FP-Stereo.

A. Matching Cost Computation

Matching costs measure the similarity of corresponding pixels [25]. Given pixel $\mathbf{p} = (x, y)$ in the base image, the corresponding pixel at disparity d in the rectified match image is $\mathbf{p}$-d, where $d = \{0, 1, 2, ..., d_{max}-1\}$ and d_{max} is the disparity range [25]. Different kinds of cost functions can be used to compute the matching cost volume $C(\mathbf{p}, d)$.

Sum of absolute differences (SAD) adds the absolute differences of intensities over all the pixels in a square window N_p centered at the pixel of interest $\mathbf{p}$. For each pixel $\mathbf{q}$ in N_p in the base image, the absolute difference is computed by comparing the value of $\mathbf{q}$ with its corresponding pixel $\mathbf{q}$-d in the match image. The matching cost is computed as

$$C_{SAD}(\mathbf{p}, d) = \textstyle\sum_{\mathbf{q}\in N_\text{p}} |I_b(\mathbf{q}) - I_m(\mathbf{q} - d)|, \tag{1}$$

where $I_b(\mathbf{q})$ and $I_m(\mathbf{q} - d)$ are the values of corresponding pixels in the base and match images, respectively.

Zero-mean sum of absolute differences (ZSAD) subtracts the mean intensity of the window N_p from each intensity inside the window before computing the sum of absolute differences:

$$C_{ZSAD}(\mathbf{p}, d) = \textstyle\sum_{\mathbf{q}\in N_\text{p}} |I_b(\mathbf{q}) - \bar{I}_b(\mathbf{p}) - (I_m(\mathbf{q}-d) - \bar{I}_m(\mathbf{p}-d))|, \tag{2}$$

where $\bar{I}_b(\mathbf{p})$ and $\bar{I}_m(\mathbf{p} - d)$ are the mean intensities of the windows centered at pixel $\mathbf{p}$ and $\mathbf{p}$-d, respectively.

Census transform indicates the relative order of intensities in a local window centered at the pixel of interest, not the intensity values themselves [26]. It encodes each window into a bit string as follows:

$$CT(\mathbf{p}) = \otimes_{\mathbf{q}\in N_\text{p}} \Phi(I(\mathbf{p}), I(\mathbf{q})), \tag{3}$$

where $\otimes$ denotes the bit-wise concatenation, and $I(\mathbf{p})$ and $I(\mathbf{q})$ are the values of the central pixel and the neighboring pixel in the window N_p, respectively. $\Phi(i, j)$ is set to 1 if $i > j$, or 0 otherwise. After encoding, the matching cost is calculated through the Hamming distance between bit strings of corresponding pixels in the image pair, which is defined as the number of bits that are not equal, as shown in Eq. 4:

$$C_{census}(\mathbf{p}, d) = \mathbb{H}(CT_b(\mathbf{p}), CT_m(\mathbf{p} - d)), \tag{4}$$

where $CT_b(\mathbf{p})$ and $CT_m(\mathbf{p} - d)$ are transformed bit strings of corresponding pixels in the image pair.

Rank transform is also based on the relative ordering of local intensities [26], defined as the number of pixels in the window N_p with an intensity less than the central pixel $\mathbf{p}$:

$$RT(\mathbf{p}) = \|\{\mathbf{q} \in N_\text{p} | I(\mathbf{q}) < I(\mathbf{p})\}\|. \tag{5}$$

The matching cost is then computed with the absolute difference between the transformed values of corresponding pixels in base and match images, as shown in Eq. 6:

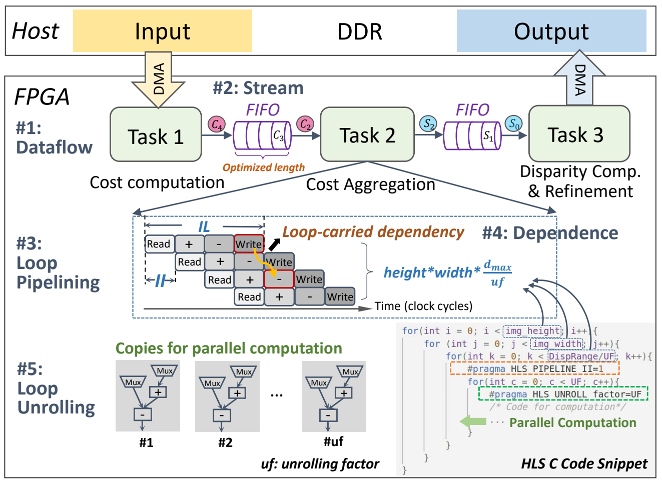
$$C_{rank}(\mathbf{p}, d) = |RT_b(\mathbf{p}) - RT_m(\mathbf{p} - d)|. \tag{6}$$

B. Cost Aggregation

To smooth the matching cost volume, SGM aggregates costs along independent paths, as shown in Fig. 1. For each path, SGM optimizes a minimization problem recursively based on the pathwise energy function, defined as

$$L_r(\mathbf{p}, d) = C(\mathbf{p}, d) + \min \begin{Bmatrix} L_r(\mathbf{p} - \mathbf{r}, d), \\ L_r(\mathbf{p} - \mathbf{r}, d \pm 1) + P_1, \\ \min\limits_i L_r(\mathbf{p} - \mathbf{r}, i) + P_2 \end{Bmatrix} - \min_i L_r(\mathbf{p} - \mathbf{r}, i), \tag{7}$$

where $L_r(\mathbf{p}, d)$ denotes the path cost of pixel $\mathbf{p}$ at disparity d in direction $\mathbf{r}$ and $C(\mathbf{p}, d)$ represents the matching cost computed in Section II-A. The second term represents the recursive aggregation from the previous pixel $\mathbf{p}-\mathbf{r}$ in direction



```
for(int i = 0; i < img_height; i++){
    for (int j = 0; j < img_width; j++){
        for(int k = 0; k < DispRange/UF; k++){
            #pragma HLS PIPELINE II=1
            for(int c = 0; c < UF; c++){
                #pragma HLS UNROLL factor=UF
                /* Code for computation*/
            }
        }
    }
}
```

Fig. 2: Optimization with HLS pragmas.

r. Penalties P_1 and P_2 are added for disparity adaption in slanted/discontinuous surfaces [27]. The minimum path cost in the third term is subtracted to avoid a very large value. The aggregated cost is then computed by summing all the path costs, i.e., $S(\mathbf{p}, d) = \sum_r L_r(\mathbf{p}, d)$.

C. Disparity Computation and Refinement

The disparity $D_b(\mathbf{p})$ of each pixel $\mathbf{p}$ in the base image is computed using a winner-takes-all (WTA) strategy [10], which selects the disparity with the minimum aggregated cost, i.e., $D_b(\mathbf{p}) = \min_d S(\mathbf{p}, d)$. After computation, we provide several optional disparity refinement methods to remove outliers.

Left-right (L-R) consistency check removes mismatching and occluded pixels [10]. The disparity of the pixel $\mathbf{p}$ in the based image, i.e., $D_b(\mathbf{p})$, is set to invalid if it differs from the disparity of the corresponding pixel $\mathbf{p}'$ in the match image, i.e., $D_m(\mathbf{p}')$, by more than 1 px. To obtain the disparity map of the match image, an efficient way is to reuse the aggregated cost volume for the base image based on $D_m(\mathbf{p}') = \arg\min_d S(\mathbf{p}' + d, d)$. A more accurate method is to perform cost computation and aggregation from scratch and switch the position of the base and match images. Both methods are supported in our library.

Finally, we also provide a **median filter** with a user-defined window size to remove outliers and smooth the disparity map.

III. Optimization and Automation

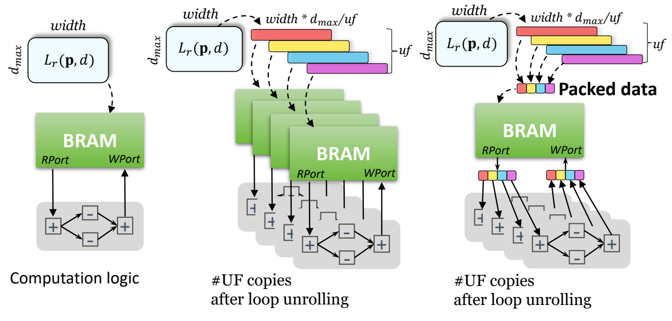
In this section, we will present our optimization and automation techniques which are applied in FP-Stereo to boost performance, save resources and improve usability.

Well-set HLS pragmas. To boost the performance and exploit the parallelism, we optimize the functions in the library with various HLS pragmas, as shown in Fig. 2. *Dataflow* enables coarse-grained parallelism among functions and loops. By inserting FIFOs between each module, data is sent to the consumer immediately once processed by the producer, and different modules execute concurrently. In this case, the overall latency equals the latency of the slowest module [28]. To keep a high data rate while considering limited on-chip memory resources, FIFO lengths are set as short as possible using the *stream* pragma, provided that the data path is not stalled. *Loop pipelining* and *loop unrolling* exploit fine-grained parallelism within loops. A pipelined loop processes new inputs at an initiation interval (*II*), constrained by the loop-carried dependency [28, 29]. To eliminate harmful data dependencies, the *dependence* pragma is applied to avoid the false dependence analysis of HLS tools which may happen under complicated scenarios. Moreover, the hardware implementation should also be well designed to resolve harmful dependence, which will be discussed in Section IV. *Loop unrolling* allows concurrent execution of independent iterations by creating copies of the loop body, and is inserted in the inner-most loop considering both efficiency and hardware cost. In our case, the latency of each module (clock cycles) can be estimated using

$$Cycle = IL + II \times (height \times width \times \frac{d_{max}}{uf} - 1), \qquad (8)$$

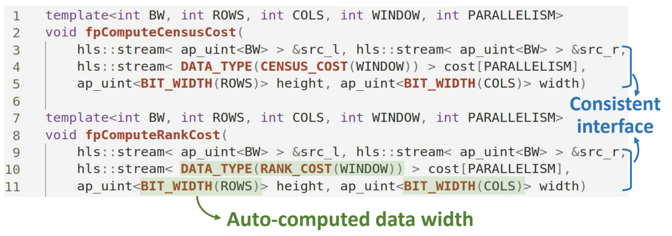
where *IL* is the iteration latency, *height* is the image height, *width* is the image width, d_{max} is the disparity range and *uf* is the unrolling factor in the disparity dimension.

Memory optimization. Simultaneous memory accesses are critical for parallel computation, but may not be ensured by the on-chip memories on FPGA (i.e., BRAMs) due to the limited number of read and write ports. Figure 3 presents how we utilize BRAMs efficiently to maximize performance. We take a row of data with the size $width * d_{max}$ from the path cost volume $L_r(\mathbf{p}, d)$ as an example. The data is stored in an array which is mapped to BRAMs. When the loop is unrolled, to enable concurrent reads and writes required by the copies of computation logic, the *array partitioning* pragma divides



(a) No partitioning (b) Array partitioning (c) Data packing

Fig. 3: Memory optimization.



```
template<int BW, int ROWS, int COLS, int WINDOW, int PARALLELISM>
void fpComputeCensusCost(
    hls::stream< ap_uint<BW> > &src_l, hls::stream< ap_uint<BW> > &src_r,
    hls::stream< DATA_TYPE(CENSUS_COST(WINDOW)) > cost[PARALLELISM],
    ap_uint<BIT_WIDTH(ROWS)> height, ap_uint<BIT_WIDTH(COLS)> width)

template<int BW, int ROWS, int COLS, int WINDOW, int PARALLELISM>
void fpComputeRankCost(
    hls::stream< ap_uint<BW> > &src_l, hls::stream< ap_uint<BW> > &src_r,
    hls::stream< DATA_TYPE(RANK_COST(WINDOW)) > cost[PARALLELISM],
    ap_uint<BIT_WIDTH(ROWS)> height, ap_uint<BIT_WIDTH(COLS)> width)
```

Fig. 4: Code snippet of the library.

the array into multiple partitions and stores each partition in different BRAMs, boosting the parallelism while incurring multi-fold memory usage. To save memory resources, we pack the elements required by the parallel computation and store the packed data in a single memory instance, as shown in Fig. 3(c). The required data can still be accessed quickly by fetching the packed data and feeding different bits to the corresponding computation logic. Through *data packing*, the on-chip memories on FPGA can be fully utilized and the memory usage is reduced by 50% compared to *array partitioning*. Note that we still apply *array partitioning* to small arrays which will be mapped to registers.

Auto-computed data width and consistent interface. In general C code, the native data types are all on multiple bytes, which can result in inefficient hardware. In HLS, users can specify an arbitrary data width for each variable. To improve usability, FP-Stereo automatically specifies the optimal data width for each variable without influencing the accuracy. Based on closed-form expressions, the data widths are computed automatically during compilation. Specifically, the data widths depend on the algorithmic characteristics and parameters, such as the type of cost functions and the window size, as shown in Fig. 4. Our flexible optimized data types significantly reduce resource usage, resulting in a faster circuit and allowing for a higher clock frequency. Moreover, function templates in the same stage of the pipeline are implemented with the consistent interface in a unified format, ensuring the scalability of our library.

Automatic Code Generation. To quicken the FPGA development cycle, FP-Stereo can automatically generate the complete stereo matching pipeline based on our optimized HLS templates. Given user-specified algorithmic parameters, FP-Stereo generates function calls, sets parameters and inserts buffers between functions to form a deep pipeline. The input and output images are streamed into and out of FPGA through the direct memory access (DMA) engine. Besides the accelerator code, FP-Stereo also generates the host code to allocate/free the DDR memory space and invoke the accelerator.

IV. Implementation Details

A. Matching Cost Computation

The inputs of the matching cost computation stage are the 8-bit intensities of base and match images and the output is the matching cost volume. As discussed in Section II-A, cost functions rely on window-based computation. Neighboring pixels in a local window are collected to compute the matching costs of the central pixel. As shown in Fig. 5, input image intensities are transferred to FPGA every clock cycle in the raster scanning order and the windows to be processed at time stamps t-1 and t are highlighted. The notations represent the pixels that have been streamed in FPGA and the pixel $\mathbf{p}$ is the coming pixel at clock cycle t. The image data is transferred to FPGA continuously in a single burst after the host receives the transfer request. Each piece of data is transferred once and needs to be stored in the local memory on FPGA for reuse.

We utilize *window and line buffers* for data reuse, as shown

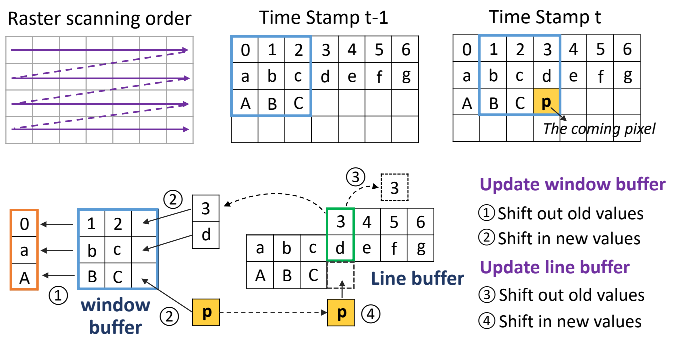
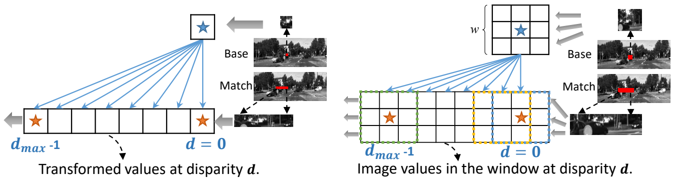


Fig. 5: Buffer reuse strategy. A 3×3 window buffer and a 2×7 line buffer are presented for illustration.



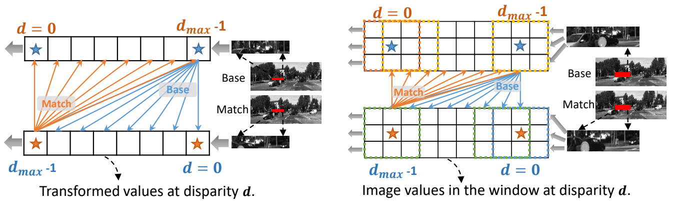
Base
Match
d_{max} -1
$d = 0$
Transformed values at disparity d.
w
Image values in the window at disparity d.

(a) Census and rank transform (b) SAD and ZSAD

Fig. 6: Matching cost computation for the base image

in Fig. 5. A window buffer stores neighboring values in a local window and a line buffer stores multiple rows of values. At each clock cycle, the window buffer is updated by shifting out the left-most column of data and shifting in the data part from the line buffer and part from the coming pixel. The line buffer is then updated to store the most recently visited rows of data by shifting out the old value and shifting in the coming pixel in the same column. This date reuse strategy efficiently utilizes on-chip memories by storing the necessary data required by the computation in the raster order rather than the whole image.

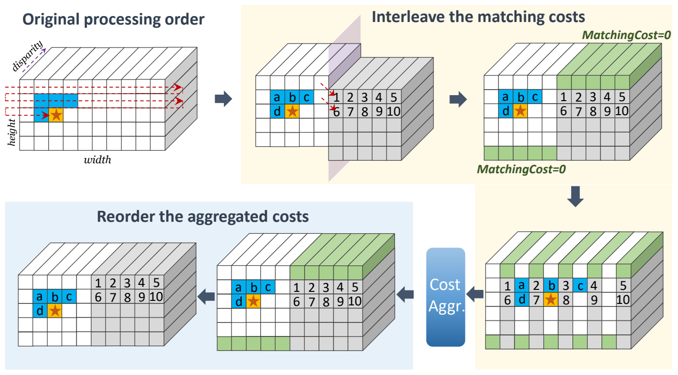
Based on window and line buffers, the neighboring pixels of the target pixel within a local window can be accessed immediately for the matching cost computation in the raster scanning order. For census and rank transform, the neighboring pixels are compared to the central pixel in parallel. After transformation, as shown in Fig. 6(a), the hamming distance or the absolute difference is computed between the transformed values of the target pixel in the base image and its corresponding pixels at each disparity in the match image. Then the FIFO is updated by shifting left to remove



$d = 0$
d_{max} -1
Match
Base
d_{max} -1
$d = 0$
Transformed values at disparity d.
Image values in the window at disparity d.

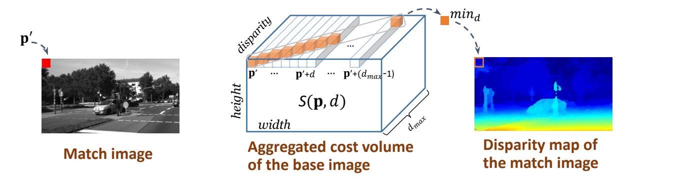
(a) Census and rank transform (b) SAD and ZSAD

Fig. 7: Matching cost computation for base & match images



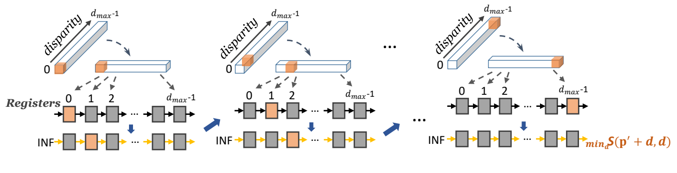
Original processing order
disparity
height
width
Interleave the matching costs
MatchingCost=0
MatchingCost=0
Reorder the aggregated costs
Cost Aggr.

Fig. 8: Interleaving and reordering for cost aggregation. The target pixel and its preceding pixels are highlighted in orange and blue, respectively.



p′
Match image
disparity
height
width
$S(\mathbf{p}, d)$
d_{max}
min_d
Aggregated cost volume of the base image
Disparity map of the match image

(a) Disparity computation of the match image



(b) Hardware implementation

Fig. 9: Disparity computation for the match image

the unused data and store the new incoming data to get prepared for the next computation. For SAD and ZSAD, the window centered at the target pixel is directly compared to the windows centered at corresponding pixels at each disparity. To save resources, we use a window buffer with the size $w * (d_{max} + w - 1)$ to store the windows of the match image, which is updated as shown in Fig. 6(b). In the case of the L-R consistency check, the matching costs of the match image are computed simultaneously, as is shown in Fig. 7. Note that the computation for the match image is slower than that of the base image for $d_{max} - 1$ pixels. FP-Stereo supports both types of matching cost computation in Fig. 6 and Fig. 7 for each cost function. The matching cost computation stage executes as an efficient pipeline with $II = 1$, sending the matching costs to the next stage every clock cycle.

B. Cost Aggregation

In this stage, the matching costs are accumulated along several paths and the path costs are summed to calculate the aggregated cost. The independent pathwise accumulation enables concurrent computation, which may not be fully exploited due to limited on-chip memory resources. To solve this issue, parallelism can be achieved by accumulating along the paths which are consistent with the scanning order. As shown in Fig. 8, since pixels are processed in the raster order, the path costs of the preceding pixels (blue) in the directions of 0^o, 45^o, 90^o and 135^o are available when performing the accumulation at the target pixel (orange). For the 45^o, 90^o and 135^o directions, the accumulation at the target pixel relies on the path costs computed in the previous row. Line buffers are used to store the path costs for each direction and updated with the most recently visited row. To save memory resources, the path costs which are computed in parallel are packed. For the direction of 0^o which exactly follows the raster order, the path costs are stored in registers which are updated at each pixel. When traversing each pixel, the computations in the four paths are performed concurrently and the path costs are summed immediately after computation.

The cost aggregation stage outputs aggregated costs of each pixel at different disparities at an initiation interval (II). In the direction of 0^o, each pixel has to wait for the computation of the left pixel to finish, leading to a tight dependence and increasing II. For other directions, the computations of preceding pixels are completed earlier by one row, which does not influence the II. To minimize II, we propose a novel *interleaving and reordering* strategy to enlarge the distance between dependent pixels, as illustrated in Fig. 8. The matching cost volume is divided in half, and the latter half is shifted down to be interleaved with the former half. The missing locations are padded with zeros. In this way, the matching costs in the latter half are still processed after the costs of the former half in the same row, maintaining their relative positions in the raster scanning order rather than affecting the original algorithm behavior. Following the rescheduled order, the pixels with tight dependence are interleaved and the computation at grey positions can be performed concurrently while the white positions wait for the results of preceding pixels. The aggregated costs are then reordered to follow the original order and transferred to the next stage with $II = 1$.

Four FIFOs are used for interleaving and reordering, and *data packing* is used to save memory resources. Additional registers are used to store results computed at the cutting edge. The number of paths to aggregate matching costs influences both *accuracy* and *speed*. The four-path aggregation achieves high hardware efficiency, while the accuracy may be affected due to the missing backward information from other directions (e.g., 180^o). Based on our scalable library-based framework, we will support the cost aggregation in more directions in the near future.

C. Disparity Computation and Refinement

The disparity of each pixel is computed using a multi-level multiplexer which selects the disparity with the minimum aggregated cost. The median filter is implemented based on window and line buffers. For the L-R consistency check, the disparity map of the match image, D_m, is required to compare with that of the base image. FP-Stereo provides two methods to compute D_m, as discussed in Section II-C. Figure 9 illustrates the method of reusing the aggregated cost volume of the base image and our hardware implementation. Given a pixel $\mathbf{p}' = (x, y)$ in the match image, its disparity is computed by comparing the aggregated costs of pixels $\mathbf{p}', ..., \mathbf{p}' + (d_{max} - 1)$ in the base image, at disparities $0, ..., d_{max} - 1$, respectively. As shown in Fig. 9(b), when the aggregated costs of each pixel come in the raster order, the registers are updated with the lower values after comparison to the costs at each disparity, and then shifted right for comparison with the aggregated costs of the next pixel. For illustration, the aggregated costs involved in the disparity computation for $\mathbf{p}'$ in Fig. 9(a) are also highlighted in orange in Fig. 9(b). After d_{max} comparisons, the disparity of $\mathbf{p}'$ is computed by recording the corresponding disparity of the minimum aggregated cost stored in the rightmost register. Note that the computations for other pixels are performed concurrently and the results are updated in the grey registers. Therefore, the disparities can be produced every clock cycle. For the method of computing from scratch, the cost aggregation function is instantiated twice, for the base and match images, respectively. Then the base and match disparity maps can be computed using their individual aggregated cost volume. This method increases the accuracy further but doubling the resource usage. The appropriate approach can be chosen depending on the practical design requirements posed by the trade-off between accuracy and hardware budgets.

V. Experimental Results

To investigate the trade-off among *accuracy*, *speed* and *resource usage*, we test different configurations of our library on FPGA, analyze the impact of various design choices and provide users guidance for the right design choice. We also compare our implementations with state-of-the-art FPGA and GPU solutions, demonstrating the superior performance of FP-Stereo in real-time embedded systems with strict *power* constraints. The experiments are performed on Xilinx Ultrascale+ ZCU102 FPGA and the KITTI 2015 dataset [23] is used for evaluation. The FPGA development toolkit we use is Xilinx SDSoC 2018.3. The power is measured on board using Xilinx Zynq UltraScale+ MPSoC Power Advantage tool [30].

A. Analysis on the impact of design choices

We evaluate 576 configurations each of which includes a choice of the cost function and its window size (5×5, 7×7), the disparity range (64, 128), the unrolling factor in the disparity dimension (4, 8, 16, 32), image resolution (1242*374, 900*260) and refinement methods. *Accuracy* is measured by the percentage of erroneous pixels on non-occluded areas (D1-all error from KITTI), *speed* is reported through the execution time (s) of processing one image pair, and *resource usage* is evaluated by averaging the utilization ratios of each FPGA component, i.e., BRAMs, DSPs, LUTs and flip-flops (FFs). For the analysis in this section, the results of execution times and resources are from HLS reports.

The impact of adjusting the cost function. The execution time is computed through dividing the latency (clock cycles) by the frequency (300MHz). The latency can be estimated

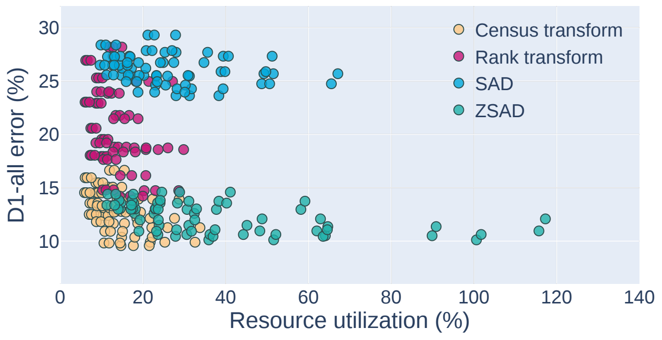


Fig. 10: D1-all error vs resource utilization colored by different cost functions.

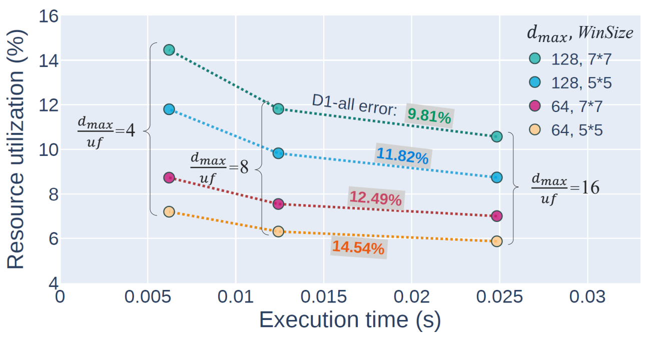


Fig. 11: Comparison given different *WinSize*, d_{max} and *uf*. Configuration setting: census transform, image resolution of 1242*374, median filter and no L-R consistency check.

with Eq. 8 which can be used for theoretical analysis. When adjusting the type of cost function, the execution times are close with negligible differences caused by iteration latency *IL*, which depends on the latencies of computing operations in each formulation in Section II-A. Figure 10 plots all the tested configurations based on *accuracy* vs *resource utilization*, categorized by different cost functions. Census transform and ZSAD achieve higher accuracy and SAD performs the worst. Rank transform exhibits a large variance in *accuracy* and some configurations lead to small errors. Compared to SAD and ZSAD, census and rank transform consume fewer resources and are more suitable to be mapped onto hardware. This is because census and rank transforms require smaller bit widths to represent matching costs without losing information, leading to less consumption of computation logic and memories. Compared to other methods, census transform is the most robust and hardware-efficient method considering both *accuracy* and *resource utilization*.

The impact of adjusting the window size (*WinSize*), the disparity range (d_{max}) and the unrolling factor (*uf*). We vary the three variables and plot the configurations in Fig. 11, while fixing other design choices as in the configuration setting. The execution time is related to d_{max}/uf, as shown in Eq. 8, which reflects the parallelism degree on hardware. With smaller d_{max}/uf, the parallelism is improved, leading to faster speeds while consuming more resources. The accuracy

Method	Config.	Accuracy			Speed		Resource Utilization				Power (W)	Energy (J)
		D1-bg	D1-fg	D1-all	Runtime (s)	FPS	BRAM	DSP	LUT	FF		
xfOpenCV [24]	S0	15.51%	27.42%	17.31%	0.0124	81	13.8%	3.0%	16.7%	5.7%	3.6	8.9
	S1	13.77%	21.85%	14.99%	0.0124	81	24.3%	5.6%	25.2%	7.3%	-	-
FP-Stereo	C0	12.62%	25.31%	14.54%	**0.0062**	**161**	**9.2%**	**2.9%**	**13.4%**	**3.2%**	**4.1**	**5.1**
	C1	10.53%	**19.04%**	11.82%	**0.0062**	**161**	16.9%	5.5%	20.1%	4.7%	5.7	7.1
	C2	8.16%	19.11%	9.81%	**0.0062**	**161**	19.6%	5.5%	26.6%	6.1%	6.6	8.2
	C3	7.78%	19.61%	9.57%	0.0068	147	19.9%	5.6%	52.8%	7.7%	6.7	9.1
	C4	**7.69%**	20.37%	**9.49%**	0.0068	147	38.7%	10.8%	68.7%	12.1%	9.8	13.3

TABLE I: Comparison on FPGA. S0, C0: *WinSize* $= 5 \times 5$, $d_{max} = 64$, *uf* $= 16$, NLR. S1, C1: *WinSize* $= 5 \times 5$, $d_{max} = 128$, *uf* $= 32$, NLR. C2-C4: *WinSize* $= 7 \times 7$, $d_{max} = 128$, *uf* $= 32$ and NLR, LR1 and LR2 are applied respectively. Median filter is applied to C0-C4.

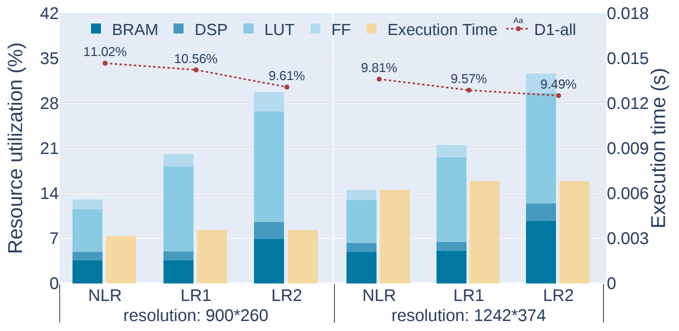


Fig. 12: Comparison for L-R check methods and image size. Configuration setting: census transform, *WinSize* $= 7 * 7$, $d_{max} = 128$, $uf = 32$ and median filter.

is positively correlated with d_{max} and *WinSize*. As shown in Fig. 11, with larger d_{max} and *WinSize*, more image information is involved to reduce the estimation error, while incurring a heavier computation burden. This leads to more resource consumption to achieve the same speed.

The impact of adjusting L-R check methods and image size. In Fig. 12, we compare various L-R check methods in terms of *accuracy*, *speed* and *resource usage*, given different image resolutions. NLR represents that the L-R consistency check is not applied. For the L-R consistency check, LR1 and LR2 denote different methods of computing the match image disparities, i.e., computing using the aggregated costs of the base image and computing from scratch, respectively. Given the same resolution, both LR1 and LR2 improve the accuracy but consume more resources and slightly increase the runtime due to additional computation. Specifically, LR2 consumes more resources than LR1 because of the duplication of functions to exploit parallelism. Different image resolutions mainly impact the *speed* and the BRAM usage, because the latency can be smaller given fewer pixels to be processed, and the length of line buffers is reduced given images with shorter image widths.

B. Comparison with the state-of-the-art on KITTI

To illustrate the performance of FP-Stereo, we compare our configurations with state-of-the-art implementations. KITTI images with the full resolution of 1242*374 are tested. We evaluate our configurations on board and Table I lists the

Method	Speed (FPS)	Power (W)	Energy (J)	$\frac{\text{FPS}}{\text{Watt}}$	Platform
OpenCV [31]	0.27	1.6	1185	1x	ARM Cortex-A53 CPU
SGM [32]	**238**	101	84.9	14x	Nvidia Titan X GPU
SGM [32]	29	11.7	80.7	15x	Nvidia JetsonTX2 GPU
FP-Stereo-C2	161	**6.6**	**8.2**	**145x**	Xilinx ZCU102 FPGA

TABLE II: Comparison on different platforms.

results tested on the Xilinx Ultrascale+ ZCU102 FPGA board. The *accuracy* is measured on 200 KITTI image pairs following the default error criterion of KITTI. The *speed* is evaluated with the runtime (s) of processing one image pair and frames per second (FPS). The *power consumption* is measured using the Power Advantage Tool [30] and the *energy* denotes the total energy consumption for the KITTI dataset with 200 image pairs, which is computed as *power* $*$ *runtime* $*$ 200. Compared to the SGM configurations provided by Xilinx xfOpenCV [24] (S0, S1) and given the same algorithmic parameters, FP-Stereo configurations (C0, C1) achieves 3.24% lower error, 2x faster speed, 30% less resource usage and 40% less energy consumption. FP-Stereo can also provide more options, such as C2-C4, which can further improve the accuracy (6.08% lower error) while maintaining a fast speed (1.8x) at the cost of more resources and energy. The different configurations provided by FP-Stereo demonstrate the flexibility of our library to meet different practical requirements and the capability of FP-Stereo to balance different design metrics for real-world embedded applications.

We also compare our method to state-of-the-art SGM designs on different platforms, as shown in Table II. We run the OpenCV implementation [31] on the embedded ARM CPU as the baseline and compute speed/power ratios (FPS/Watt) for each method. SGM [32] is the fastest GPU implementation on the KITTI leaderboard. We test SGM [32] on both high-end GPU (Nvidia Titan X) and low-power embedded GPU (Nvidia Jetson TX2) with the same parameters as the configuration C2 of FP-Stereo. We can see that SGM [32] runs at the fastest speed on a power-hungry GPU but the performance is degraded to 29 FPS when running on the low-power embedded GPU. Given the same parameters as [32], FP-Stereo-C2 achieves the same accuracy and executes at a fast speed with 161 FPS at the full KITTI resolution. Regarding energy efficiency, ours consumes the least energy and achieves the best speed/power ratio with 145x FPS/Watt.

VI. Conclusion

We propose FP-Stereo for building high-performance stereo matching pipelines on FPGAs automatically to meet different user requirements in *accuracy*, *speed* and *hardware cost*. Multiple methods are supported for each stage of the stereo matching pipeline and effective approaches are proposed to fully exploit the parallelism, efficiently utilize the resources and adequately ensure the usability. We analyze the impact of different design choices to provide informative guidance for users to select suitable implementations. The comparisons on different platforms demonstrate the superior performance of FP-Stereo compared to the state-of-the-art methods. FP-Stereo is open-source and available at https://eeweiz.home.ece.ust.hk/.

VII. Acknowledgements

The work was supported by the GRF grant 16213118 from the Research Grants Council in Hong Kong.

References

[1] T. Huntsberger, H. Aghazarian, A. Howard, and D. C. Trotz, "Stereo vision–based navigation for autonomous surface vessels," *Journal of Field Robotics*, vol. 28, no. 1, pp. 3–18, 2011.

[2] W. Gao, K. Wang, W. Ding, F. Gao, T. Qin, and S. Shen, "Autonomous aerial robot using dual-fisheye cameras," *Journal of Field Robotics*, 2020.

[3] M. Menze, C. Heipke, and A. Geiger, "Object scene flow," *ISPRS Journal of Photogrammetry and Remote Sensing (JPRS)*, 2018.

[4] L. Zhang, W. Ding, J. Chen, and S. Shen, "Efficient uncertainty-aware decision-making for automated driving using guided branching," *arXiv preprint arXiv:2003.02746*, 2020.

[5] T. Tao, J. C. Koo, and H. R. Choi, "A fast block matching algorthim for stereo correspondence," in *2008 IEEE Conference on Cybernetics and Intelligent Systems*. IEEE, 2008, pp. 38–41.

[6] K.-J. Yoon and I. S. Kweon, "Adaptive support-weight approach for correspondence search," *IEEE transactions on pattern analysis & machine intelligence*, no. 4, pp. 650–656, 2006.

[7] V. Kolmogorov and R. Zabih, "Computing visual correspondence with occlusions via graph cuts," Cornell University, Tech. Rep., 2001.

[8] J. Sun, N.-N. Zheng, and H.-Y. Shum, "Stereo matching using belief propagation," *IEEE Transactions on Pattern Analysis & Machine Intelligence*, no. 7, pp. 787–800, 2003.

[9] A. Klaus, M. Sormann, and K. Karner, "Segment-based stereo matching using belief propagation and a self-adapting dissimilarity measure," in *18th International Conference on Pattern Recognition (ICPR'06)*, vol. 3. IEEE, 2006, pp. 15–18.

[10] H. Hirschmuller, "Stereo processing by semiglobal matching and mutual information," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 30, no. 2, pp. 328–341, 2007.

[11] W. Wang, J. Yan, N. Xu, Y. Wang, and F.-H. Hsu, "Real-time high-quality stereo vision system in fpga," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 25, no. 10, pp. 1696–1708, 2015.

[12] Y. Li, Z. Li, C. Yang, W. Zhong, and S. Chen, "High throughput hardware architecture for accurate semi-global matching," *Integration*, 2017.

[13] M. Roszkowski and G. Pastuszak, "Fpga design of the computation unit for the semi-global stereo matching algorithm," in *17th International Symposium on Design and Diagnostics of Electronic Circuits & Systems*. IEEE, 2014, pp. 230–233.

[14] C. Banz, S. Hesselbarth, H. Flatt, H. Blume, and P. Pirsch, "Real-time stereo vision system using semi-global matching disparity estimation: Architecture and fpga-implementation," in *2010 International Conference on Embedded Computer Systems: Architectures, Modeling and Simulation*. IEEE, 2010, pp. 93–101.

[15] S. K. Gehrig, F. Eberli, and T. Meyer, "A real-time low-power stereo vision engine using semi-global matching," in *International Conference on Computer Vision Systems*. Springer, 2009, pp. 134–143.

[16] M. Buder, "Dense real-time stereo matching using memory efficient semi-global-matching variant based on fpgas," in *Real-Time Image and Video Processing 2012*, vol. 8437. International Society for Optics and Photonics, 2012, p. 843709.

[17] J. Hofmann, J. Korinth, and A. Koch, "A scalable high-performance hardware architecture for real-time stereo vision by semi-global matching," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition Workshops*, 2016, pp. 27–35.

[18] K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *2011 International Conference on Field-Programmable Technology*. IEEE, 2011, pp. 1–8.

[19] Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: productivity, performance, and software constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, 2012.

[20] O. Rahnama, D. Frost, O. Miksik, and P. H. Torr, "Real-time dense stereo matching with elas on fpga-accelerated embedded devices," *IEEE Robotics and Automation Letters*, vol. 3, no. 3, pp. 2008–2015, 2018.

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

[22] O. Rahnama, T. Cavalleri, S. Golodetz, S. Walker, and P. Torr, "R3sgm: Real-time raster-respecting semi-global matching for power-constrained systems," in *2018 International Conference on Field-Programmable Technology (FPT)*. IEEE, 2018, pp. 102–109.

[23] M. Menze and A. Geiger, "Object scene flow for autonomous vehicles," in *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, 2015, pp. 3061–3070.

[24] Xilinx, "Xilinx xfopencv library," 2018.

[25] H. Hirschmuller and D. Scharstein, "Evaluation of stereo matching costs on images with radiometric differences," *IEEE transactions on pattern analysis and machine intelligence*, vol. 31, no. 9, pp. 1582–1599, 2008.

[26] R. Zabih and J. Woodfill, "Non-parametric local transforms for computing visual correspondence," in *European conference on computer vision*. Springer, 1994, pp. 151–158.

[27] Y. Boykov, O. Veksler, and R. Zabih, "Fast approximate energy minimization via graph cuts," *IEEE Transactions on pattern analysis and machine intelligence*, vol. 23, no. 11, pp. 1222–1239, 2001.

[28] J. Zhao, L. Feng, S. Sinha, W. Zhang, Y. Liang, and B. He, "Performance modeling and directives optimization for high level synthesis on fpga," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 2019.

[29] J. Zhao *et al.*, "Comba: A comprehensive model-based analysis framework for high level synthesis of real applications," in *Proceedings of the 36th International Conference on Computer-Aided Design*. IEEE Press, 2017, pp. 430–437.

[30] Xilinx, "Zynq ultrascale+ mpsoc power advantage tool 2018.1," *Xilinx Wiki*, 2018.

[31] "https://opencv.org."

[32] D. Hernandez-Juarez, A. Chacon, A. Espinosa, D. Vazquez, J. C. Moure, and A. M. Lopez, "Embedded real-time stereo estimation via semi-global matching on the gpu," *Procedia Computer Science*, 2016.