FPGA Acceleration of the Horn and Schunck Hierarchical algorithm

Ilias Bournias, Roselyne Chotin, Lionel Lacassagne

FPGA Acceleration of the Horn and Schunck Hierarchical Algorithm

Ilias Bournias*, Roselyne Chotin* and Lionel Lacassagne*
*Sorbonne Université, CNRS, LIP6, F-75005 Paris, France
firstname.surname@lip6.fr

***Abstract*—This work proposes a highly tunable motion estimation architecture. We implement the Horn and Schunck algorithm with the hierarchical extension for larger motion estimations in FPGAs. Different architectures are explored dealing with interpolation, pipeline, parallelism and arithmetic format, in order to fit performance. We show in our exploration, how the different cores of our system should be used to increase the throughput. Our smallest design achieves a 30.8 Mpixel/s in a 1024×1024 resolution and the fastest 507 Mpixel/s which is one of the fastest ever achieved, as far as we know, for FPGAs.**

I. Introduction

A. State-of-the-Art

Optical flow algorithms are used to estimate the velocity of each pixel between a pair of images. These algorithms are used in a variety of applications from object detection, motion compensation, to autonomous driving.

Most of the literature focuses on the accuracy [1], and only few target embedded systems and address the tradeoffs that one has to do for a realtime implementation, namely the number of iterations, the computing format (the number of bits of fixed or floating point number), and the parallelism.

There are a lot of optical flow algorithms according to their typically organized according to their computational speed, accuracy and specific application.

The estimation of the optical flow in real time is a challenging task because it requires a lot of computation efforts and in the same time the hardware to remain low. There were a lot of works in optimizing optical flow algorithms in CPU [2]–[4], GPU [5]–[7] and FPGAs. Especially for FPGAs, some works use the Lucas-Kanade ($L\&K$) method with mono-scale and multi-scale implementations for [8] while [9] remains on the multi-scale Phase-based algorithm and [10] proposes a lower frame memory access to reduce external memory interactions. Finally the works [11]–[13] implement a Horn and Schunck optical flow mono-scale algorithm, the first in an iterative mode and the others also in partial and fully pipelined modes.

B. Horn and Schunck algorithm

The basic scheme of Horn and Shunck ($H\&S$) [14] is an iterative algorithm (Fig. 1(a)) that estimates (u, v) from the first spatio-temporel derivatives I_x, I_y, I_t (of a pair of images) and from the previous average values $(\bar{u}, \bar{v})$, according to (1) and (2) where α is a smoothing parameter.

$$u = \bar{u} - I_x \frac{I_x u + I_y v + T_t}{\alpha^2 + I_x^2 + I_y^2} \quad (1)$$

$$v = \bar{v} - I_y \frac{I_x u + I_y v + T_t}{\alpha^2 + I_x^2 + I_y^2} \quad (2)$$

As the derivatives are estimated with a $2 \times 2 \times 2$ kernel, the computed velocities should be smaller than 1 pixel / frame. That is the reason why, multi-scale (aka hierarchical) scheme should be considered (Fig. 1(b)).

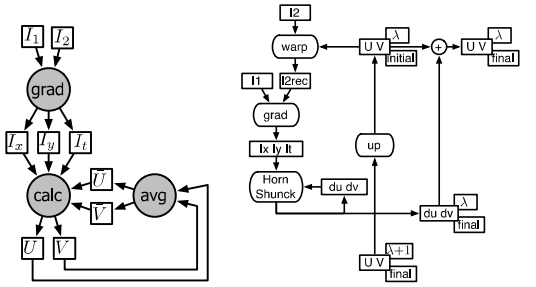
From the computed velocities $(u, v)^{\lambda+1}_{final}$ of level $\lambda + 1$, a new velocity field is initialized : $(u, v)^{\lambda}_{init}$ by up-scaling the previous one with a factor 2, and multiplying it also by a factor 2: $(u, v)^{\lambda}_{init} = 2 \times \text{Upscale}(u, v)^{\lambda+1}_{final}$. These velocities are used to compensate (warp) the motion between the two images (I_{2rec}), thanks to a bi-linear or bi-cubic interpolation. Then $H\&S$ kernel iterates to provide the residual velocities $(\delta u, \delta v)$. After the iterations these residuals are accumulated to the initial estimation: $(u,v)^{\lambda}_{final} = (u, v)^{\lambda}_{init} + (u, v)^{\lambda}$ to provide the final velocity estimations at this level. Then same computations are done for the next level: $(u, v)^{\lambda-1}_{init} = 2 \times \text{Upscale}(u, v)^{\lambda}_{final}$ and so on until level $\lambda = 0$.

In this paper, we first present an architecture that implements the multi-scale $H\&S$ algorithm in FPGA. As this architecture is highly tunable, section III will describe a design space exploration methodology to reach the performance whose results are discussed in section IV. Then, we conclude and explore future works.

C. Contributions

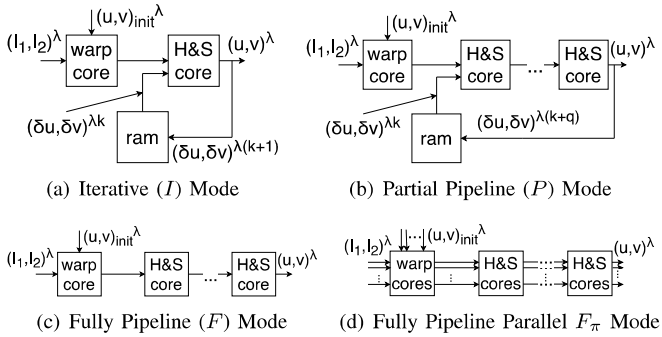
Instead of setting a constant number of iteration for every level, the convergence is better if one puts a high number of iterations at coarse level (small images) and low number of iterations at fine level (large images). Thus a very standard configuration with a 3-level pyramid with a ×2-factor (20,10,5 iterations) has been selected for our implementations which achieves a motion estimation in the range of $(V, U) < |7|$, but a ×4-factor (80,20,5 iterations) can be also considered. The image size is set at 1024×1024 pixels but can be adapted in other sizes too.

Our contribution is to propose a multi-scale implementation of $H\&S$ where the parameters to explore are the parallelism (1,5,10 or 20 computing kernels), the floating point format (16-bit or 32-bit IEEE floating points) and the warp interpolation (bi-linear or bi-cubic). So we do not try to address accuracy,



(a) H&S mono-scale iterative scheme (b) H&S multi-scale iterative scheme at level λ

Fig. 1. Description of the Algorithm



(a) Iterative (I) Mode (b) Partial Pipeline (P) Mode

(c) Fully Pipeline (F) Mode (d) Fully Pipeline Parallel F_π Mode

Fig. 2. Different Implementations Used

as the convergence depends on the observed scene, we only evaluate the impact of the precision (16 or 32 bits) on the size of the design.

II. Proposed Architecture

In this section we will describe the different cores used in our design.

A. Horn and Schunck core

In this paper, we propose four different designs (Fig.2) depending on the number of $H\&S$ cores used and the number of iterations for each pyramid level. The first three implementations are the standard iterative (I), the partial pipeline (P) and fully pipeline (F) used by [12], [13]. The last one is a fully pipeline parallel implementation with π parallel cores (F_π). In the last case the π parallel core is mandatory to access to 3x(π+2) neighboured previous average values ($\bar{u}, \bar{v}$), instead of the 3x3 required for the first 3 designs. With this design π pixels per clock cycle can be processed.

B. Warping core

For the warping core two cases have been explored, the bi-linear and bi-cubic interpolation. For the bi-linear (resp. bi-cubic) interpolation, a neighbourhood of 2x2 (resp. 4x4) pixels in the input image is required. The goal is to interpolate one pixel per clock cycle, so we have to ensure that in every clock cycle all the required neighboured pixels are available in a similar way as in [15]. To do that the worst case scenario has to be examined. It occurs when the optical flow vectors summed from levels 2 and 1 have to be interpolated with the input image in level 0 ($(V,U) < |6|$). This means that for bi-linear (resp. bi-cubic) interpolation, 14 (resp. 16) lines have to be stored in 14 (resp. 16) FIFOs. In every clock cycle a new pixel is read from the external memory and all the remaining pixels inside the FIFOs are moved one position to the right so that all the required pixels for the interpolation are available without latency. In order to choose the right neighbouring pixels in the two cases, the integer parts of the velocity vectors are needed and the interpolation is done with the fraction part of the velocities. In order to process π pixels per clock cycle, we have to parallelize the computation. In that case, π pixels per clock cycle are read from the memory.

C. Sum, Up-scaling core

The sum core is used to sum the velocities calculated in the previous levels with those of the current level. These values are stored in an on chip memory to reduce the interaction with the external memory. Another solution would be to store the velocities in an external memory, but in this case both read and write operations are needed in not neighboured addresses:

- read operations to provide the warp core with the adequate velocities,
- and write operation to merge the new calculated velocities with the ones from the previous level.

This might make the full pipeline of one pyramid level more difficult. Then, results from the sum are extended (up-scaling) in order to be used in the next pyramid level. For up-scaling two FIFO line memory are needed as was made in [8], [9].

D. Pyramid Creation

Each image used for each level of the pyramid is being built after the convolution from the coarser pyramid level with a 5x5 Gaussian kernel (down-sampling) and then stored in the external memory in order to be used for the next steps of the algorithm. Seven FIFOs are used for the pyramid built:

- five for the Gaussian kernel,
- and two to ensure continuous streaming because each coarser level image is four times smaller than the finer level image so for every 4 pixels read from the memory one is written.

E. Pipeline and Parallelism in each Pyramid Level

It is obvious from the sections above that the up-scaling, warping, $H\&S$ and sum can be performed in a pipeline way for each pyramid level. This has a major advantage: interpolated pixels do not need to be written back to the external memory and then read again, but they can be directly processed by the $H\&S$ core in a pipeline way. It is also possible to compute the optical flow vectors in a fully pipeline parallel way.

III. Design space exploration

As depicted in the previous section, there are a lot of possibilities with the architecture to manage performance. In this sense, a design space exploration has been performed to find the architecture that matches as closely as possible

the designer's requirements. The exploration is based on the number of the $H\&S$ cores used in our design. Depending on the number of iterations in each level of the pyramid these cores can be used in different modes: iterative (I), partial pipeline (P), fully pipeline (F) and fully pipeline parallel (F_π) as shown in Fig 2. These modes have different impacts on computation time. The same cores used in iterative mode in one pyramid level can be used in another mode in a different pyramid level by changing the pipeline depth of the FIFOs memories used and by adding more of them in series or in parallel. For example with respectively 20, 10, 5 iterations for $level_2$, $level_1$, $level_0$ and 10 available cores, then:

- in $level_2$, the cores are used in partial pipeline mode,
- in $level_1$, in fully pipeline mode with the FIFOs used in $level_2$ doubled in depth,
- and in $level_0$, in fully pipelined parallel and the number of FIFOs used in $level_1$ doubled and used in parallel.

Another critical part is that if in one level π $H\&S$ cores are used in parallel then π interpolation cores have to be used in parallel for a continuous computation of π pixels per clock cycle. The up-scaling and sum components can also be adapted as to be used in parallel mode. We should also mention that when we divide the image width by π, the residue has to be zero in order to ensure that the π parallel pixels are in the same line.

By taking all these into account, depending on the number of cores (Π) used and the number of iterations, the total time T for the multi-scale algorithm calculation of the image, without the down-sampling (which takes less than 15% of the total time), can be estimated by (3).

$$T = \frac{Height \cdot Weight \cdot (i_0 + \frac{i_1}{4} + \frac{i_2}{16})}{f \cdot \Pi} + lat \cdot \frac{1}{f} \quad (3)$$

where lat is the latency added from every core which is low regarding the total computation time, i_k the iterations for $level_k$ and f the running frequency.

Given the number of iterations i_0 at $level_0$, the total number of logic elements N can be estimated by (4) and (5).

$$N = \Pi \cdot N_{HS} + \lceil \frac{\Pi}{i_0} \rceil \cdot (N_{warp}) + N_1 \quad (4)$$

$$N_1 = \lceil \frac{\Pi}{i_0} \rceil \cdot N_{sum} + \lceil \frac{\Pi}{i_0} \rceil \cdot N_{upscaling} + N_{downsampling} \quad (5)$$

where N_{HS}, N_{warp}, N_{sum}, $N_{upscaling}$ and $N_{downsampling}$ are respectively the numbers of logic elements of $H\&S$, interpolation, sum, up-scaling and down-sampling cores.

The total memory M can also be estimated by (6), (7) and (8). M_2 represents the extra memory used by P mode in $level_1$ to avoid storing the intermediate velocity values in the external memory. This happens because the memory used for sum core for this task is not enough.

$$M = \Pi \cdot M_{HS} + M_{warp} + M_1 + M_2 \quad (6)$$

$$M_1 = M_{sum} + \cdot M_{upscaling} + M_{downsampling} \quad (7)$$

$$M_2 = Width \cdot 2 \cdot \frac{Height \cdot Weight}{16} \quad (8)$$

where M_{HS}, M_{warp}, M_{sum}, $M_{upscaling}$ and $M_{downsampling}$ are respectively the memories used to store values of $H\&S$, warp, sum, up-scaling and down-sampling cores. $Width$ represents the word length depending on the format used (F_{16}, F_{32}).

The maximum bandwidth of the external memory required is defined by $level_0$ and $level_1$ of the pyramid. In $level_0$ the input pixels from the images are 8 bit wide whereas the pixels in $level_1$ are depending on the format used (F_{16}, F_{32}). This happens because the finest level of the pyramid does not get down-sampled. As a result if the calculation in $level_0$ is done in F_π mode with F_{16} then the maximum bandwidth is determined by $level_0$, otherwise by $level_1$.

TABLE I
MONO-SCALE & MONO-CORE RESULTS FOR F_{32} AND F_{16}

core	logic blocks	registers	memory
bi-linear interp. F_{16}	4,543 (2%)	13,261	292,244 (< 1%)
bi-cubic interp. F_{16}	17,919 (8%)	51,375	326,876 (< 1%)
Horn Schunck F_{16}	4,599 (2%)	13,759	150,352 (< 1%)
sum F_{16}	1,841 (< 1%)	5,777	9,437,184 (18%)
up-scaling F_{16}	1,768 (< 1%)	2,077	67,100 (< 1%)
down-sampling F_{16}	13,344 (5.6%)	23,110	94,358 (< 1%)
bi-linear interp. F_{32}	7,648 (3%)	18,261	605,644 (1%)
bi-cubic interp. F_{32}	44,991 (19%)	105,845	624,288 (1%)
Horn Schunck F_{32}	9,686 (4%)	19,720	270,483 (< 1%)
sum F_{32}	4,351 (2%)	11,717	16,777,216 (32%)
up-scaling F_{32}	2,840 (< 1%)	9,798	135,628 (1%)
down-sampling F_{32}	24,387 (10%)	68,666	190,469 (< 1%)

IV. RESULTS OF IMPLEMENTATION

For the implementation of the algorithm, the FPGA Altera Stratix V 5SGXEA7H3F35C3 was used. The images from the computer where written to two 64-bit data bus DDR3 memories with a maximum speed of 800 MHz using a PCI express interface. The communication of the external memory with the FPGA was done with the help of two DDR3 SDRAM Controllers with UniPHY provided from Altera. Each DDR3 memory is used for the storage of each image.

In Table I we can see the information about all the key components used in our design. All the components are implemented in VHDL and without DSP to increase clock working frequency. Half and single precision floating point numeric formats are used in the same way as in [19] and all the units are built with the help of the FloPoCo library [20].

In Table II we can see the total resources used regarding the number of $H\&S$ cores, the type of interpolation used and the fps achieved by each implementation. There are also 2 mono-scale fully pipelined versions (v_1),(v_8) with half (F_{16}) and single precision (F_{32}) floating point format in order to do a fair comparison in the same arithmetic system.

From Table II we can see that by increasing the number of $H\&S$ cores a better fps is achieved. What is also interesting is when for the finest pyramid level 4 pixels per clock cycle are computed then 4 interpolation cores are used as described in section III. That highly impacts the resource usage. For a smaller design the bi-linear interpolation should be chosen for the hardware area to remain small compared to bi-cubic.

TABLE II
DESIGN SPACE EXPLORATION RESULTS

Implementation F_{16}	mono-scale v_1	multi-scale v_2	multi-scale v_3	multi-scale v_4	multi-scale v_5	multi-scale v_6	multi-scale v_7
Iterations	20	20,10,5	20,10,5	20,10,5	20,10,5	20,10,5	20,10,5
HS cores (Π)	20	1	5	10	20	10	20
Mode	F	I^{20},I^{10},I^5	P^4,P^2,F	P^2,F,F_2	F,F_2,F_4	P, F,F_2	F,F_2,F_4
interpolation	-	bi-cubic	bi-cubic	bi-cubic	bi-cubic	bi-linear	bi-linear
Logic blocks	93,570(40%)	53,531(23%)	67,055(29%)	109,258(47%)	192,386(82%)	82,430(35%)	148,537 (63%)
registers	250,418	150,706	189,719	293,683	529,456	208,720	382,203
memory	3,195,788 (6%)	10,232,738 (20%)	13,264,090 (25%)	11,650,850 (22%)	13,589,222 (26%)	11,617,698 (22%)	13,350,693 (25%)
frequency	339	276	293	281	270	286	274
fps	310	29.4	149	275	477	277	484
Implementation F_{32}	mono-scale v_8	multi-scale v_9	multi-scale v_{10}	multi-scale v_{11}	multi-scale v_{12}	multi-scale v_{13}	multi-scale v_{14}
Iterations	20	20,10,5	20,10,5	20,10,5	20,10,5	20,10,5	20,10,5
HS cores (Π)	20	1	5	10	20	10	20
Mode	F	I^{20},I^{10},I^5	P^4,P^2,F	P^2,F,F_2	F,F_2,F_4	P, F,F_2	F,F_2,F_4
interpolation	-	bi-cubic	bi-cubic	bi-cubic	bi-cubic	bi-linear	bi-linear
Logic blocks	187,211(80%)	98,384(42%)	138,012(59%)	not implementable	not implementable	168,127(71%)	not implementable
registers	368413	215,633	295,138	-	-	340,993	-
memory	5,932,160(11%)	17,230,659(33%)	23,571,910(45%)	-	-	20,854,914(40%)	-
frequency	209	228	224	-	-	208	-
fps	191	24.3	114	-	-	202	-

TABLE III
COMPARISON TO STATE-OF-THE-ART IMPLEMENTATION ON FPGA (SORTED ON THE THROUGHPUT)

Implementation	Algorithm	size	format	Π	frame rate	Throughput (Mpixel/s)	Architecture
This work v_2	Multi-scale H&S	1024×1024	F_{16}	1	29.4 (4.2)	30.8 (4.4)	Stratix V 291Mhz (40Mhz)
This work v_4	Multi-scale H&S	1024×1024	F_{16}	10	275 (38.8)	288 (40.6)	Stratix V 296Mhz (40Mhz)
This work v_7	Multi-scale H&S	1024×1024	F_{16}	20	484 (70.6)	507 (70.4)	Stratix V 278Mhz (40Mhz)
(I) [13]	Mono-scale H&S	640×512	Q	1	30.0	9.2	Stratix IV 295 Mhz
[9]	Multi-scale Phase-based	640×480	$Q_{8.0\rightarrow 8.4}$	-	31.5	9.6	Virtex-4 45MHz
[8]	Multi-scale L&K	640×480	$Q_{9.0\rightarrow 29.8}$	-	32.0	9.8	Virtex-4 83 MHz
[16]	Phase-based	512×512	$Q_{8\rightarrow 12}$	-	40.0	10.4	Virtex-4 42MHz
[17]	HBM + Refinement	640×480	-	-	39.0	12.0	Virtex-7 200MHz
[10]	Mono-scale L&K	800×600	$Q_{4.6\rightarrow 26}$	-	170.0	81.6	Virtex-6 94 MHz
(F) [12]	Mono-scale H&S	1920×1080	$Q_{7.10}$	128	84.0	174.2	Virtex-7 150MHz
(F) [12]	Mono-scale H&S	1920×1080	F_{32}	32	96.5	200	Virtex-7 200MHz
(F) [13]	Mono-scale H&S	4096×2304	Q	20	30.0	283.1	Stratix IV -
[18]	Mono-scale L&K	1024×1024	$Q_{10\rightarrow 32}$	-	1000.0	1048	Virtex-2 90 MHz

If accuracy is the point then bi-cubic interpolation is better. Moreover, we can see that the v_6 and v_{13} have less logic block usage than mono-scale v_1 and v_8 with small impact in fps. So these 2 designs can replace the mono-scale implementations if memory usage is not important and better range for velocities is needed. Finally, the F_{32} requires more than the double of the resources than the F_{16} meaning that designs v_{11}, v_{12} and v_{14} are not implementable.

In Table III we make a comparison of our works with the state of the art Optical flow algorithms implemented in FPGA. In our works, we have 2 clock frequencies of which the second one is almost the same with the ones used by the previous works in order to do a fair comparison. We can see from this table that v_7 outperforms in terms of throughput all the other designs except those of Ishii [18]. The reason is that it implements a Mono-scale $L\&K$ algorithm with pseudo-variable frame rate by using two FPGAs, one for the transformation of the serial input to parallel, the other for the calculation of the product sums and a PC for the rest of the steps of the algorithm. Especially now for the $H\&S$ algorithm, we can see that our iterative v_2, v_4 and v_7 implementations compared to the ones of [13] I, F achieve a $\times 3.34$, $\times 1.02$ and $\times 1.79$ throughput. Finally we notice that v_4 and v_7 compared to the other multi-scale implementations [9], [8], [17] achieve a $\times 4.2$, $\times 4.1$, $\times 3.9$ and $\times 7.3$, $\times 7.18$, $\times 6.76$ throughput respectively, even with lower frequency.

V. CONCLUSION

We proposed a parametric hierarchical implementation of the gradient Based $H\&S$ motion estimation algorithm in the Stratix V FPGA which, as far as we know, has never been done before. Our exploration showed that we have to switch from single precision to half precision floating point format to fit all the designs we propose. Furthermore we showed that our smallest design can reach 30.8 Mpixel/s with 23% usage of the Logic Blocks of the Stratix V and that our fastest outperforms in terms of throughput all the existing state of the art optical flow designs that use solely FPGAs, achieving 507 Mpixel/s with a cost of 63% of the Stratix V FPGA Logic Blocks. Finally we showed that contrary to the limited range of all the previous $H\&S$ mono-scale designs, all our multi-scale designs achieve a range of 7 for the calculated velocities.

In the future we plan to do more design space exploration on the multi-scale $H\&S$ design by exploring fixed point format, ways to reduce the on-chip memory usage and we also plan to deal with its accuracy.

REFERENCES

[1] D.Scharstein and R. Szeliski, "A Taxonomy and Evaluation of Dense Two-Frame Stereo Correspondence Algorithms," *International Journal of Computer Vision*, vol. 47, pp. 7–42, 2002.

[2] T. Kroeger, R. Timofte, D. Dai, and L. Van Gool, "Fast Optical Flow using Dense Inverse Search," in *Computer Vision – ECCV 2016*.

[3] A. Petreto, A. Hennequin, T. Koehler, T. Romera, Y. Fargeix, B. Gaillard, M. Bouyer, Q. L. Meunier, and L. Lacassagne, "Energy and Execution Time Comparison of Optical Flow Algorithms on SIMD and GPU architectures," in *2018 Conference on Design and Architectures for Signal and Image Processing (DASIP)*, 2018, pp. 25–30.

[4] A. Garcia-Dopico, J. Pedraza, M. Nieto, A. Perez, S. Rodrigeuz, and J. Navas, "Parallelization of the optical flow computation in sequences from moving cameras," *EURASIP Journal on Image and Video Processing*, p. 18, 2014.

[5] A. Plyer, G. Le Besnerais, and F. Champagnat, "Massively parallel Lucas Kanade optical flow for real-time video processing applications," *Journal of Real-Time Image Processing*, vol. 11, pp. 713–730, 2016.

[6] J. D. Adarve and R. Mahony, "A Filter Formulation for Computing Real Time Optical Flow," *IEEE Robotics and Automation Letters*, vol. 1, no. 2, pp. 1192–1199, 2016.

[7] T. Senst, R. H. Evangelio, I. Keller, and T. Sikora, "Clustering Motion for Real-Time Optical Flow Based Tracking," in *2012 IEEE Ninth International Conference on Advanced Video and Signal-Based Surveillance*, 2012, pp. 410–415.

[8] F. Barranco, M. Tomasi, J. Diaz, M. Vanegas, and E. Ros, "Parallel Architecture for Hierarchical Optical Flow Estimation Based on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 6, pp. 1058–1067, 2012.

[9] M. Tomasi, M. Vanegas, F. Barranco, J. Diaz, and E. Ros, "High-Performance Optical-Flow Architecture Based on a Multi-Scale, Multi-Orientation Phase-Based Model," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 20, no. 12, pp. 1797–1807, 2010.

[10] H. Seong, C. E. Rhee, and H. Lee, "A Novel Hardware Architecture of the Lucas–Kanade Optical Flow for Reduced Frame Memory Access," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 26, no. 6, pp. 1187–1199, 2016.

[11] R. Rustam, N. H. Hamid, and F. A. Hussin, "FPGA-based hardware implementation of optical flow constraint equation of Horn and Schunck," in *2012 4th International Conference on Intelligent and Advanced Systems (ICIAS 2012)*, vol. 2, 2012, pp. 790–794.

[12] M. Komorkiewicz, T. Kryjak, and M. Gorgon, "Efficient hardware implementation of the Horn-Schunck algorithm for high-resolution real-time dense optical flow sensor," *Sensors*, vol. 14, no. 2, pp. 2860–2891, 2014.

[13] M. Kunz, A. Ostrowski, and P. Zipf, "An FPGA-optimized architecture of horn and schunck optical flow algorithm for real-time applications," in *2014 24th International Conference on Field Programmable Logic and Applications (FPL)*, 2014, pp. 1–4.

[14] B. K. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, no. 1, pp. 185 – 203, 1981.

[15] M. Tomasi, M. Vanegas, F. Barranco, J. Diaz, and E. Ros, "Real-Time Architecture for a Robust Multi-Scale Stereo Engine on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 12, pp. 2208–2219, 2012.

[16] M. Tomasi, M. Vanegas, F. Barranco, J. Daz, and E. Ros, "Massive Parallel-Hardware Architecture for Multiscale Stereo, Optical Flow and Image-Structure Computation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 2, pp. 282–294, 2012.

[17] K. Seyid, A. Richaud, R. Capoccia, and Y. Leblebici, "FPGA-Based Hardware Implementation of Real-Time Optical Flow Calculation," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 28, no. 1, pp. 206–216, 2018.

[18] I. Ishii, T. Taniguchi, K. Yamamoto, and T. Takaki, "High-Frame-Rate Optical Flow System," *IEEE Transactions on Circuits and Systems for Video Technology*, vol. 22, no. 1, pp. 105–112, 2012.

[19] S. Piskorski, L. Lacassagne, S. Bouaziz, and D. Etiemble, "Customizing CPU Instructions for Embedded Vision Systems," in *2006 International Workshop on Computer Architecture for Machine Perception and Sensing*, 2006, pp. 59–64.

[20] F. de Dinechin and B. Pasca, "Designing Custom Arithmetic Data Paths with FloPoCo," *IEEE Design Test of Computers*, vol. 28, no. 4, pp. 18–27, 2011.