A High–Performance Parallel Implementation of the Chambolle Algorithm

Abdulkadir Akin[‡], Ivan Beretta[‡], Alessandro Antonio Nacci[†],
Vincenzo Rana[‡], Marco Domenico Santambrogio[†], David Atienza[‡]
[‡]Embedded Systems Laboratory (ESL), Ecole Polytechnique Fédérale de Lausanne (EPFL), Switzerland.
E-mail: abdulkadir.akin@epfl.ch, ivan.beretta@epfl.ch, vincenzo.rana@epfl.ch, david.atienza@epfl.ch
[†]Dipartimento di Elettronica e Informazione (DEI), Politecnico di Milano, Italy.
E-mail: alessandro.nacci@mail.polimi.it, santambr@elet.polimi.it

***Abstract*—The determination of the optical flow is a central problem in image processing, as it allows to describe how an image changes over time by means of a numerical vector field. The estimation of the optical flow is however a very complex problem, which has been faced using many different mathematical approaches. A large body of work has been recently published about variational methods, following the technique for total variation minimization proposed by Chambolle. Still, their hardware implementations do not offer good performance in terms of frames that can be processed per time unit, mainly because of the complex dependency scheme among the data. In this work, we propose a highly parallel and accelerated FPGA implementation of the *Chambolle* algorithm, which splits the original image into a set of overlapping sub-frames and efficiently exploits the reuse of intermediate results. We validate our hardware on large frames (up to 1024×768), and the proposed approach significantly improves state-of-the-art implementations, reaching up to $76\times$ speedups, which enables real-time frame rates even at high resolutions.**

I. Introduction

The *optical flow* is a vector field representing the velocity of an object in a sequence of frames, and it can be determined by analyzing the variation of the brightness inside a sequence of successive images [1]. The estimation of this vector field is one of the most important problems in image and video processing, as it can be employed for motion estimation [2] and compensation [3], as well as in other fields such as robotics [4] and even medical analysis [5]. Another important application of the optical flow is the correction of an image acquired by CMOS optical sensors using the rolling shutter technique [6], which is nowadays used in most of the cheap photo cameras. In particular, rolling shutter is a method of image acquisition in which each frame is recorded by scanning across the frame either vertically or horizontally, which may generate errors and distortions in the final image.

The optical flow estimation problem is considered to be computationally challenging [5] because of the large amount of movements that can be detected in a frame, and because of the noise that can alter the image brightness. A wide range of different techniques, such as [7] [8] [9], has been proposed in the past, but variational methods [10] have emerged as one of the most successful approaches in recent years. The variational technique we consider in this work is called *TV-L*^{1} [11], which distinguishes itself from other approaches because it can handle highly–varying intensities in the frames.

978-3-9810801-7-9/DATE11/©2011 EDAA

The *TV-L*^{1} method includes both a mathematical definition of the variational problem, and a numerical scheme to compute the solution. The numerical scheme is based on a fixed-point algorithm originally proposed by Chambolle [12], which iteratively refines the solution (which in this case is the optical flow estimation) at different levels of precision. Though *TV-L*^{1} seems to be very promising from a theoretical point of view, its implementations don't reach real-time performances, except for very small images. A multithread software implementation of *TV-L*^{1} we developed and analyzed, for example, can take more than 15 seconds to process just one frame on a standard x86 workstation, and up to 50 seconds are required on the ARM processor of an Apple iPhone 3GS. However, the profiling of the executions of *TV-L*^{1} on both the platforms shows that the *Chambolle* algorithm itself is the bottleneck that generates the poor timing performances. Besides the execution of an outermost loop which does not require any complex matrix operation, approximately 90% of the execution time is spent on the *Chambolle* iterative technique, which proves to be the most critical and computationally intensive part.

Although *Chambolle* is employed in many different domains other than optical flow estimation, no parallel and efficient implementation has been proposed so far, and even the best performing implementations on GPUs are essentially sequential, and they do not achieve real-time frame rates with high resolution images [13]. This lack of performance is mainly due to the complex data dependencies during the execution, as the evaluation of one element of the vector field requires a large amount of intermediate results that were previously computed for that element and for some of its neighbors.

In this paper, we propose a novel design of the *Chambolle* algorithm, which merges multiple iterations of the algorithm and resolves the data dependencies in a very efficient way (cfr. Section V), while introducing a negligible amount of redundant computation that does not affect the final frame rates (Section VI). We exploit *loop decomposition* and *sliding windows* to divide the input frames into sub-matrices that can be processed in parallel. Then, we targeted these techniques for an FPGA device to exploit the fine-grained level of hardware

parallelism they offer, and we designed an efficient data reuse mechanism that reduces the number of memory accesses and speeds-up the execution. The proposed implementation proves to be significantly faster than the state-of-the-art approaches, and it can work on larger frames (up to 1024×768) while still reaching real-time frame rates.

The remainder of this paper is structured as follows. In Section II, we introduce the *Chambolle* algorithm and we provide an overview of the related works. Then, we discuss the proposed parallelization techniques in Section III, and the details of the FPGA implementation in Sections IV and V. We propose a set of experimental results in Section VI, and finally Section VII concludes the paper.

II. Context Definition

In this section we briefly introduce the algorithm we aim at accelerating, and then we present, at the best of our knowledge, the most relevant work in the litterature. The mathematical aspects of the algorithm is beyond the scope of this work, but they are deeply discussed in [11], [12] and [13].

A. The Chambolle Algorithm

The optical flow between two images I_0 and I_1, which are given in the form of two input matrices, is represented by a bi-dimensional vector $\mathbf{u} = (u_1, u_2)$, which is the output of the algorithm. The vector $\mathbf{u}$ is initialized at 0, and its final value is computed by means of an iterative way: each iteration is called *level*, and each level refines the estimation by increasing its precision [13]. Each level works as follows. Firstly, a support variable $\mathbf{v} = (v_1, v_2)$ is defined using a thresholding function of I_1 and of the value of $\mathbf{u}$ computed at the previous level [13]. Then, the value of $\mathbf{u}$ at the current level is determined using an iterative technique known as *Chambolle* algorithm [12], whose computational steps are summarized in Algorithm 1 (for sake of simplicity, the pseudo-code only shows the computation of u_1, but u_2 is computed in the same way, by simply substituting u_1 and v_1 with u_2 and v_2).

Algorithm 1 Chambolle Algorithm

1: **for** $i = 1, .., N_{iterations}$ **do**
2: div $p = (Backward_X(px_{u1}) + Backward_Y(py_{u1}))$
3: $Term =$ div $p - v_1/\theta$
4: $Term_1 = Forward_X(Term)$
5: $Term_2 = Forward_Y(Term)$
6: $|\nabla u_1| = \sqrt{Term_1^2 + Term_2^2}$
7: $px_{u1} = [px_{u1} + \tau/\theta \cdot Term_1] \,/\, [1 + \tau/\theta \cdot |\nabla u_1|]$
8: $py_{u1} = [py_{u1} + \tau/\theta \cdot Term_2] \,/\, [1 + \tau/\theta \cdot |\nabla u_1|]$
9: $u_1 = v_1 - \theta \cdot$ div p
10: **end for**

The vector $\mathbf{u}$ is updated by means of two intermediate values, namely $\mathbf{px} = (px_{u1}, px_{u2})$ and $\mathbf{py} = (py_{u1}, py_{u2})$, which are initialized at 0 [13]. In order to simplify the description of the hardware implementation, we also introduce the auxiliary variables $Term$, $Term_1$, and $Term_2$, whereas θ and τ are predefined values that determine the precision. The $Backward_X(z)$ function returns a matrix where each element of z is reduced by its left neighbor, whereas in $Backward_Y$ it is subtracted by its upper neighbor, in $Forward_X$ by its right neighbor, and in $Forward_Y$ by its lower neighbor.

B. Related Works

A few implementations of the *Chambolle* algorithm can be found in literature, as part of the complete *TV-L*^{1} numerical scheme. At the best of our knowledge, a parallel implementation of this approach has never been proposed because of the complex dependencies among the intermediate results.

In [11] and [13], the robust *TV-L*^{1} technique to calculate the optical flow between two frames is proposed and implemented using modern GPUs. The authors proved that a real-time frame rate can be achieved by the most powerful devices for low resolution sequences, but only few frames that are larger than 512×512 can be processed in one second. A Matlab implementation of the technique in [13] performs the estimation in 5 to 6 seconds, and it also shows some limitations in terms of memory usage.

Additional hardware results of the execution of *TV-L*^{1} on GPUs can be also found in [14], but even the fastest implementation cannot top the 6 frames per second (fps), even on 512×512 images.

Fast estimations of the optical flow can be achieved by using different techniques and by simplifying the working domain. For example, the implementation proposed in [15] can process up to 156 fps on 768×576 images, working on a low-cost FPGA device. However, the resulting optical flow is specifically suited for motion detection, and it cannot be used in other applications such as rolling shutter correction. The specific target allows the authors to filter the input frames, for example background subtraction, and to heavily simplify the amount of data to be processed for the optical flow estimation.

III. The Proposed Parallel Solution

Aim of this work is to design a parallel implementation of the *Chambolle* algorithm, which is made difficult by its recursive nature. Algorithm 1 shows that the elements of $\mathbf{px}$ and $\mathbf{py}$ computed at iteration n are immediately required at iteration $n + 1$ to compute functions $Backward_X$ and $Backward_Y$ (line 2). Furthermore, a complex dependency scheme among the elements of $\mathbf{px}$ and $\mathbf{py}$ is generated by the $Backward$ and $Forward$ functions (lines 2, 4 and 5), which relate each element of the matrix to the values of its neighbors computed at the previous iterations. We analyzed which elements at iteration n contribute to the value of a pixel at level $n + 1$, and we report them in Figure 1.a.

Starting from the aforementioned dependency scheme, we introduce two sources of parallelism in the proposed hardware implementation, namely, loop decomposition and sliding window techniques, which are shown in the following paragraphs.

A. Loop Decomposition

The *loop decomposition* technique is conceptually similar to loop unrolling [16], but our approach aims at directly computing each element of $\mathbf{px}$ and $\mathbf{py}$ at iteration $n + x$ by

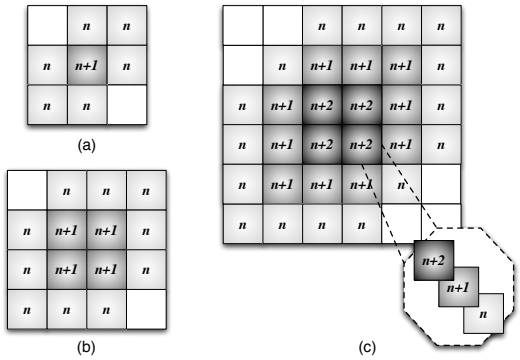

Fig. 1. Data dependencies among the data computed in successive iterations

finding a formula that employs the values available at iteration n. For example, let us consider Figure 1.a: the value of the central element at iteration $n+1$ can be directly computed only if the values at iteration n of the neighbors are also computed. In this way, however, 7 elements at iteration n have to be computed just to obtain one element at iteration $n+1$. The overhead can be reduced by computing more than one element at iteration $n+1$ as shown in Figure 1.b, where 14 elements at iteration n are required to generate four elements at $n+1$, thus reducing the overhead to 3.5 elements at iteration n for each element at iteration $n+1$. Further benefits can be achieved by applying the same technique on a larger number of iterations as shown in Figure 1.c, where elements at iteration $n+2$ are computed from the elements at iteration n. Finally, we observed that the overhead can be reduced if the group of elements that are computed are disposed on a squared shape.

Theoretically, loop decomposition allows the input matrix I_1 to be divided into different sub-matrices, which can be processed by independent cores that resolve the data dependencies locally and produce different parts of the output. However, when a core processes the elements that lie close to the border of a sub-matrix, some of the neighbors may belong to a different sub-matrix, and the dependency scheme illustrated in Figure 1 may not be satisfied. As a consequence, only a subset of the elements of a sub-matrix are computed correctly, and we call them *profitable* elements. It is worth noting that this side effect does not occur when the boundary elements also lie on the border of I_1, because the algorithm inherently treats them as special cases, and their value is computed correctly.

B. Sliding Windows

We solved the problem of the non-profitable elements by developing a technique called *sliding window*. The rationale is to divide I_1 into overlapping sub-matrices, whose profitable areas are contiguous. This approach introduces a slight memory overhead, because certain elements are replicated in multiple sub-matrices. A computation overhead is also introduced, as the cores may process some elements which are not profitable and will not be part of the output. However, the *sliding window* technique enables a coarse-grained parallelization of *Chambolle* in spite of its recursive nature and its complex data dependencies, and this greatly improves the throughput of the proposed implementation, as we will see in Section VI.

IV. A General Overview of the Proposed Hardware Solution

The proposed hardware implementation of the Chambolle algorithm is targeted to an FPGA device, which provides an ideal platform for testing the inherent parallelism of our approach. The top-level block diagram of the hardware architecture is shown in Figure 2. The hardware employs two concurrent sliding windows ($SW1$ and $SW2$) that works completely in parallel, each one updating the values of both u_1 and u_2 (we use the notation ${}^{sw1}u_1$ to indicate the value of u_1 computed by sliding window $SW1$). A sliding window is logically divided into two parts: an array of processing elements (PEs), and a dedicated amount of on-chip memory implemented on the BRAMs of the FPGA.

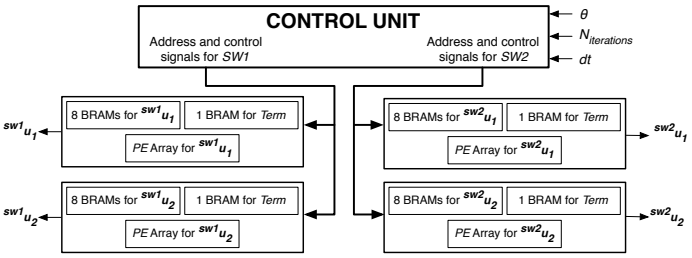

Fig. 2. Top-level block diagram of the proposed hardware implementation

Each SW works on sub-matrices of 88×92 elements, and then slides to span the entire length of the original matrix. The length has been determined according to the proposed memory scheme, which is discussed in section V-B and requires the length to be a multiple of 8, whereas the width has been chosen both for memory reasons and to obtain a shape that is close to a square, like we discussed in the previous section. A detailed view of a SW, and in particular of the circuit that processes ${}^{sw1}u_1$, is shown in Figure 3. The data required to compute the components of $\mathbf{u}$ (i.e., $\mathbf{v}$, px and py, as shown in Algorithm 1) is stored in the on-chip BRAMs, in order to reduce the access to the off-chip memory. A SW can compute 7 elements in parallel for both u_1 and u_2, thus finding 14 elements of vector $\mathbf{u}$ at the same time. This structure not only accelerates the execution, but it also enables a significant data reuse among the PEs, as discussed in the following paragraph, and reduces the access to both on-chip and off-chip memory.

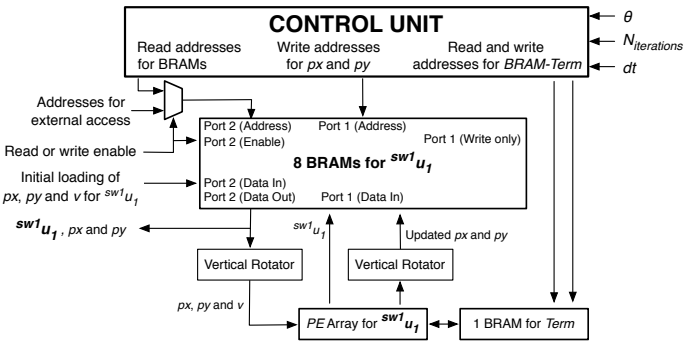

Fig. 3. Computation of ${}^{sw1}u_1$

In fact, the proposed hardware is able to compute the value

of one element in just 18 clock cycles: 1 cycle is required by the Control Unit, 1 cycle by the synchronous read from the BRAM memory, 1 cycle by the vertical rotator, and 15 cycles by the PE array. Furthermore, the processing of each one of ${}^{sw1}u_1$, ${}^{sw1}u_2$, ${}^{sw2}u_1$ and ${}^{sw2}u_2$ requires 8 BRAMs to store the respective px, py and $\mathbf{v}$ values, plus an additional BRAM that is necessary to exchange data between two iterations of the PEs. Hence, only 36 BRAMs are used: all of them are controlled by the control unit, except the initialization of the algorithm, which is performed through the FPGA input pins.

V. Implementation Details of the Chambolle Algorithm

In this section, we provide a detailed top-down description of the hardware architecture. In particular, we initially discuss the structure of the PE arrays and their memory organization, in order to show how the operands are propagated to feed the PEs. Finally, a detailed view of a single PE is provided.

A. Processing Element Arrays and Data Reuse

The proposed hardware implementation includes four PE arrays, two for each SW, to find the outputs u_1 and u_2 of *Chambolle*, which are subsequently used to update $\mathbf{v}$ by means of the thresholding function. Each PE array contains 14 processing elements, 7 of which are called *PE-T*s and are used to calculate the values of $Term$ and $\mathbf{u}$ (see Algorithm 1), while the other 7 are named *PE-V*s and are used to compute px and py. Overall, there are 56 PEs in the proposed hardware: 28 of them are *PE-T*s, and 28 of them are *PE-V*s.

The proposed *ladder* organization of a PE array that works on the first 7 rows (also called *first region*) of the input matrix is shown in Figure 4, which also illustrates how the same PEs are then reused to process the following 7 rows (second region). In particular, while *PE-T*$_1$ is calculating $Term$ for the elements in uppermost row, *PE-T*$_7$ computes $Term$ for the elements in row 6. Then, after all the PEs have completed the first 7 rows, *PE-T*$_1$ starts computing $Term$ for row 7, while *PE-T*$_7$ shifts to row 13.

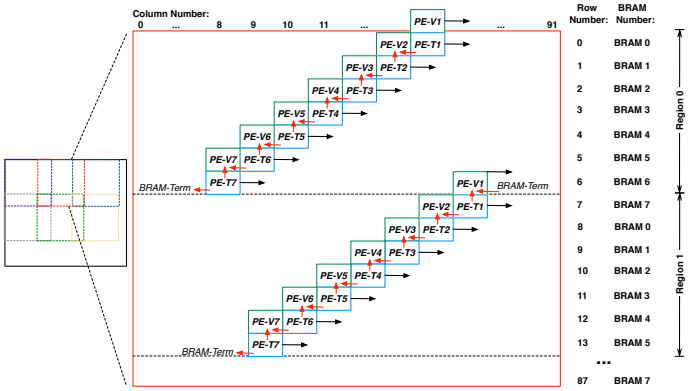

Fig. 4. Organization of 7 *PE-T*s and 7 *PE-V*s in a sliding window, and memory organization during the computation of ${}^{sw1}u_1$

The $Term$ of one element depends on the values of px and py vectors of the same element (we refer to these values as c_px and c_py), the px vector of the element on the left (l_px), and the py vector of the element above (a_py). Without any data reuse policy, each *PE-T* in a PE array requires 4 values, which have to be loaded from the on-chip memory, and consequently 4 PE arrays with 7 *PE-T*s require 112 values to be read from the memory. Thanks to the ladder organization of the PEs, we can limit this data transfer by propagating the intermediate results. Figure 5 shows how the 7 *PE-T*s are disposed, and how they were aligned in the previous cycle (dashed boxes). Since all the PEs require their c_px and c_py vectors computed in a previous iteration, we load them from the BRAMs. Then, as the processing direction in a SW goes from left to right, these vectors can be reused as l_px and a_py vectors for the following cycle without accessing the memory. For instance, *PE-T*$_3$ takes the l_px vector from the flip-flop that stores the c_px vector processed in previous cycle. Similarly, c_py can be reused as a_py by the *PE-T*s which are located below, as for example the c_py vector used by *PE-T*$_2$ is the a_py vector of *PE-T*$_3$ for the next cycle.

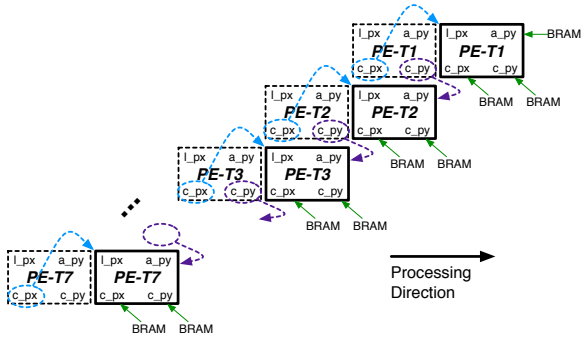

Fig. 5. Data reuse among the 7 *PE-T*s during the computation of ${}^{sw1}u_1$ (the dashed boxes indicate the position of the *PE-T*s in the previous cycle)

The *PE-V*s start computing px and py for one element one cycle after the *PE-T*s, and they also exploit a massive reuse of data. Algorithm 1 shows that, in order to compute px and py vectors for an element, three $Term$ values are required: the one of the corresponding element, the one of its right neighbor, and the one of the bottom neighbor. We reuse the values of $Term$ which processed by the battery of *PE-T*s, and we propagate them using pipelining flip-flops. For instance, in order to compute px and py for the element at position $(2, 11)$, the $Term$ values of elements in $(2, 11)$, $(2, 12)$ and $(3, 11)$ are required. *PE-T*$_3$ calculates the $Term$ value at $(2, 11)$, and at the same time *PE-T*$_4$ calculates the $Term$ value for $(3, 10)$. In the next clock cycle, *PE-T*$_3$ and *PE-T*$_4$ compute the $Term$ values for $(2, 12)$ and $(3, 11)$, respectively Then, *PE-V*$_3$ takes the required $Term$ values from *PE-T*$_3$ and *PE-T*$_4$, as well as the synchronized result of *PE-T*$_3$ that was computed in previous clock cycle, and determines the new px and py for element $(2, 11)$, without reading any data from BRAM. Once the values of px and py have been determined, they are stored in BRAM for the following iterations.

B. Memory Organization

The proposed data reuse scheme reduces both the number of accesses to the BRAMs and the amount of memory required

to store the intermediate results. As shown in Figure 5, the battery of *PE-T*s needs to read 15 vectors from BRAMs, but 28 vectors would be required if data reuse is not implemented. In this subsection, we show how those BRAMs are organized.

According to Figure 4, *PE-V*s from 2 to 7 take the required values of $Term$ from the two adjacent *PE-T*s and from the result computed in previous clock cycle by the *PE-T*s that are on their right. Therefore, the computation of these six *PE-V*s does not require any additional BRAM to store the intermediate values of $Term$ computed by the *PE-T*s. Only *PE-V*$_1$ needs to load the $Term$ values computed by *PE-T*$_7$ in the previous region, which has to be stored in a BRAM block (called *BRAM-Term*). For instance, in order to calculate px and py for row 6, the values of $Term$ for rows 6 and 7 are required, but they cannot be computed in successive clock cycles because the two rows belong to two different regions (see Figure 4), and are processed by the PE array in two separate moments. Therefore, the $Term$ values of row 6 are stored in a dual-port BRAM, and they are read back when *PE-T*$_1$ computes the $Term$ values of row 7.

As a PE array requires 8 BRAMs for px, py and $\mathbf{v}$ (which explains why the length of the sub-matrix has been set as a multiple of 8, in this case to 88 elements), and also a *BRAM-Term* block is required as a bridge between two different regions, 9 BRAMs are required to process each region. The results computed by each *PE-V* are stored in the corresponding BRAMs according to the addressing shown in Figure 4. When the array completes a region and starts processing the following one, the address used to access the BRAMs needs to be increased by an offset of 92, and this step is performed by a *vertical rotator*, which is shown in Figure 3.

Overall, the 8 BRAMs of each region are indexed using 1012 addresses, and 32 bit blocks of data are stored in each address. The 32 bits encode $\mathbf{v}$, which requires 13 bits, followed by c_px and c_py, which require 9 bits each. After the *PE-V*s find the new values of px and py, we update the values in the BRAMs by using the write ports of the BRAMs, overwriting the vector values that have been read in previous cycles.

C. Processing Elements

In this section, we describe the *PE-T* and *PE-V* processing elements. The hardware architecture of a *PE-T* is shown in Figure 6, and the one of a *PE-V* is shown in Figure 7.

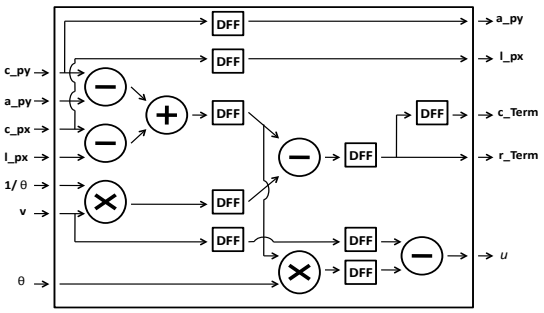

Fig. 6. Hardware architecture of a *PE-T*

The implementation of a *PE-T* includes the $Backward$ operations for px and py, which are performed in parallel

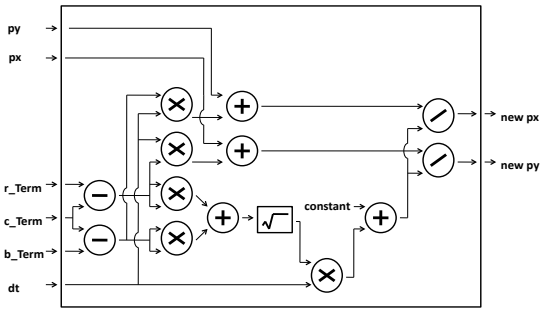

Fig. 7. Hardware architecture of a *PE-V*

before computing the value of $Term$, which is then used as r_Term for the *PE-V*s that is processing the same row, whereas b_Term can feed the *PE-V*s that is processing the upper row. Moreover, the value of $Term$ is pipelined for 1 clock cycle in order to use it as c_Term.

The hardware architecture for *PE-V*s implements the $Forward$ operations between c_Term, r_Term and b_Term in parallel, and then computes the new px and py vectors. The main issue in the design of the *PE-V* architecture is the square root function to compute px and py, as shown on line 6 of Algorithm 1. An efficient and precise hardware implementation of the square root is still an open problem [17][18], and there are two main techniques to handle it: iterative techniques, which achieves better precisions, and look-up tables, which are faster.

We opted for a look-up table implementation as we mainly focus on speed, while the achieved precision is still acceptable for *Chambolle*: the error of the approximated square root is below 1% in more than 90% of the samples we tested. The look-up table takes 32 bit signal represented using fixed point notation, where the integer part takes 24 bits, and the decimal part takes 8 bits. The entries of the table are 8-bit values, thus the table contains $2^8 = 256$ pre-computed values, and only requires 70 LUTs to be deployed on the FPGA. Instead of dividing the input value into 4 pieces of 8 bits each, which can index 4 different tables, we developed a technique that increases the precision while using only one table (thus saving approximately 12200 LUTs over the 28 *PE-V*s). In particular, we take the 8 most significant bits of the input value, and we use them to get the result from the table, discarding the remaining bits. The 8-bit block we use starts in an odd position (counting from left to right), and finishes in an even one: if the first non-zero bit is located in the n-th position, where n is even, then the 8 bit block will start from the zero bit at position $n-1$. In this way, if the decimal value of the 8 bit block is equal to m, and if the rightmost bit of the block is in position $2k$, then the number is equal to $m \cdot 2^{2k}$, and its square root can be computed by accessing the table with value m, and by left-shifting the output by k positions.

VI. Experimental Results

The proposed parallelization approach of the *Chambolle* algorithm has been fully implemented in *Verilog* and synthesized for a Xilinx Virtex 5 FPGA (XC5VLX110T). Table I shows the resource usage of the *Chambolle* core, which reaches a working frequency of 221 MHz after place and route.

If needed, the number of required DSPs can be reduced by mapping part of the multiplications on the LUTs.

TABLE I
AREA USAGE ON A XC5VLX110T FPGA

	FlipFlops	LUTs	BRAMs	DSPs
Used	23143	32829	36	62
Total	69120	69120	128	64
Percentage	33%	47%	28%	96.8%

Table II shows the comparison between the performance achieved by the proposed approach the ones obtained by the other state-of-the-art implementations. We assumed that the images to be processed are pre-loaded in the device memory, in order to focus the measures on the *Chambolle* algorithm itself. The estimated speedup achieved by the implementation proposed in this paper ranges from $16.5\times$ to $76\times$ w.r.t. images with a resolution of 512×512. Furthermore, the proposed hardware proves to scale very well with the frame size, still achieving more than 30 fps on 1024×768 images.

TABLE II
COMPARISON W.R.T. STATE-OF-THE-ART IMPLEMENTATIONS

Ref.	Device	Iterations	Image Resolution	Frame Rate (fps)
[13]	GeForce 7800 GS	50	128×128	56
		100		32.1
		200		17.5
[13]	GeForce 7800 GS	50	256×256	18
		100		9.6
		200		5
[13]	GeForce 7800 GS	50	512×512	5
		100		2.6
		200		1.3
[13]	GeForce Go 7900 GTX	50	128×128	95
		100		57
		200		30.9
[13]	GeForce Go 7900 GTX	50	256×256	34.1
		100		17.5
		200		8.9
[13]	GeForce Go 7900 GTX	50	512×512	9.3
		100		4.7
		200		2.3
[14]	ATI mobility Radeon HD3650 (OpenCV+OpenGL)	100	512×512	1-2
[14]	ATI mobility Radeon HD3650 (OpenGL only)	100	512×512	3-4
[14]	NVIDIA GTX285 (OpenGL only)	100	512×512	5-6
Proposed Approach	Xilinx Virtex-V XC5VLX110T	200	512×512	99.1
Proposed Approach	Xilinx Virtex-V XC5VLX110T	200	1024×768	38.1

VII. CONCLUSIONS

In this paper we have proposed a high-performance parallel implementation of the Chambolle algorithm, by exploiting both loop decomposition and sliding windows techniques. Experimental results show that the performance of the proposed approach are up to two orders of magnitude higher with respect to state-of-the-art solutions on 512×512 images. Furthermore, real-time frame rates can be achieved even on large images (e.g. 1024×768), which have never been targeted by the existing works. Finally, the area usage of the proposed approach is quite low, as it occupies less than half of the slices in the target FPGA device.

VIII. ACKNOWLEDGEMENTS

The authors would like to thank Prof. Pierre Vandergheynst (LTS2-EPFL) for all his support and discussions about the possible optimization benefits in the parallel implementation of the Chambolle algorithm. This research has been partially funded by the Swiss NSF Research Grant 20021-109450/1.

REFERENCES

[1] A. Verri and T. Poggio, "Motion field and optical flow: qualitative properties," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 11, no. 5, pp. 490 –498, may. 1989.
[2] S. Sun et al., "Motion estimation based on optical flow with adaptive gradients," in Proc. of International Conference on Image Processing 2000, vol. 1, 2000, pp. 852 –855 vol.1.
[3] S. Lin, et al., "An optical flow based motion compensation algorithm for very low bit-rate video coding," in Proc. of ICASSP 1997, vol. 4, apr. 1997, pp. 2869 –2872 vol.4.
[4] S. Kim et al., "Mobile robot velocity estimation using an array of optical flow sensors," in Proc. of ICCAS 2007, pp. 616 –621.
[5] S. Behbahani et al., "Analysing optical flow based methods," IEEE International Symposium on Signal Processing and Information Technology, 2007, pp. 133 –137.
[6] S. Baker et al., "Removing rolling shutter wobble," in Proc. of CVPR 2010, pp. 2392 –2399.
[7] B. K. P. Horn and B. G. Schunck, "Determining optical flow," *Artificial Intelligence*, vol. 17, pp. 185–203, 1981.
[8] M. Black and P. Anandan, "A framework for the robust estimation of optical flow," in Proc. of Computer Vision 1993, pp. 231 –236.
[9] N. Papenberg et al., "Highly accurate optic flow computation with theoretically justified warping," *International Journal of Computer Vision*, vol. 67, pp. 141–158, 2006.
[10] G. Aubert et al., "Computing optical flow via variational techniques," *SIAM Journal on Applied Mathematics*, vol. 60, pp. 156–182, 1999.
[11] T. Pock et al., "A duality based algorithm for tv-l1-optical-flow image registration," in Proc. of *MICCAI 2007*, pp. 511–518.
[12] A. Chambolle, "An algorithm for total variation minimization and applications," *Journal of Mathematical Imaging and Vision*, 2004.
[13] C. Zach et al., "A duality based approach for realtime tv-l1 optical flow," in Proc. of DAGM Symposium on Pattern Recognition, 2007.
[14] A. Weishaupt, et al., "Tracking and Structure from Motion,", available at: http://infoscience.epfl.ch/record/146572, 2010.
[15] M. Abutaleb et al, "A reliable fpga-based real-time optical-flow estimation," in Proc. of NRSC 2009, pp. 1 –8.
[16] J. W. Davidson and S. Jinturkar, "An Aggressive Approach to Loop Unrolling," in Proc. of Compiler Construction '96.
[17] I. Sajid et al., "Pipelined implementation of fixed point square root in fpga using modified non-restoring algorithm," in Proc. of ICCAE 2010, vol. 3, pp. 226 –230.
[18] Y. Li and W. Chu, "Implementation of single precision floating point square root on fpgas," in Proc. of FPGAs for Custom Computing Machines 1997, pp. 226–232.