Design Methods for Parallel Hardware Implementation of Multimedia Iterative Algorithms

Vincenzo Rana[‖†], Alessandro A. Nacci[‖], Ivan Beretta[†],
Marco D. Santambrogio[‖], David Atienza[†], Donatella Sciuto[‖]
[†]Embedded Systems Laboratory (ESL), EPFL, Lausanne, 1015, Switzerland, {ivan.beretta, david.atienza}@epfl.ch
[‖]Politecnico di Milano, Milano, 20133, Italy, {rana, nacci, santambr, sciuto}@elet.polimi.it

***Abstract*— Traditionally, parallel implementations of multimedia algorithms are carried out manually, since the automation of this task is very difficult due to the complex dependencies that generally exist between different elements of the data set. Moreover, there is a wide family of iterative multimedia algorithms that cannot be executed with satisfactory performance on Multi-Processor Systems-on-Chip or Graphics Processing Units. For this reason, new methods to design custom hardware circuits that exploit the intrinsic parallelism of multimedia algorithms are needed.**

As a consequence, in this paper, we propose a novel design method for the definition of hardware systems optimized for a particular class of multimedia iterative algorithms. We have successfully applied the proposed approach to several real-world case studies, such as iterative convolution filters and the *Chambolle* algorithm, and the proposed design method has been able to automatically implement, for each one of them, a parallel architecture able to meet real-time performance (up to 72 frames per second for the *Chambolle* algorithm), with on-chip memory requirements from 2 to 3 orders of magnitude smaller than the state-of-the art approaches.

Algorithm 1 Generic Iterative Convolution Filter

```
for (iteration = 0; iteration < N; iteration++) do
  for (x = 0; x < X; x++) do
    for (y = 0; y < Y; y++) do
      sum = 0
      for (i = -Kernel_Size; i < Kernel_Size; i++) do
        for (j = -Kernel_Size; j < Kernel_Size; j++) do
          sum = sum + Kernel(j,i) * Image(x-j, y -i)
        end for
      end for
      Temp(x,y) = sum
    end for
  end for
  for (x = 0; x < X; x++) do
    for (y = 0; y < Y; y++) do
      Image(x,y) = Temp(x,y)
    end for
  end for
end for
```

I. Introduction

In the last years, embedded systems have become a valuable platform not only for simple and low-power applications, but also for the implementation of very complex algorithms, especially in the field of image and video processing [1]. Within this context, iterative algorithms that work on large matrices [11] can be implemented today on Graphics Processing Units (GPUs), Multi-Processor Systems-on-Chip (MPSoCs) or Field Programmable Gate Arrays (FPGAs). However, GPUs and MPSoCs often fail in achieving real-time performance, as shown in Section V, on certain families of iterative algorithms ([8], [9]), such as the iterative convolution filters presented in Algorithm 1, where:

- N is the number of iterations to be performed;
- X is the horizontal width of the input image;
- Y is the vertical height of the input image (so $X \times Y$ is the size of the input image).

The reason why these architectures are not suited for the considered family of algorithms is that, on the one hand, it can be very difficult to exploit the Single Instruction Multiple Data (SIMD) paradigm with the static structure of GPUs (which have a fixed number of computational units) to implement algorithms with very complex data dependencies among iterations. On the other hand, MPSoCs are best suited for application or task level parallelism, where the operations on the input data are very sophisticated and fit the high complexity of each processing core. On the contrary, FPGAs provide a fully customizable platform where any kind of custom operation, either complex or very simple, can be implemented in hardware and applied on multiple blocks of data in parallel. Unfortunately, the design of custom systems could be a very challenging task, since methods to drive the designer in the definition of architectures optimized for multimedia algorithms are still missing.

In addition to this, the parallel implementation of an algorithm that works on large amounts of data and, in particular, on matrices, is also inherently difficult because of the nature of the algorithm itself, as it typically contains complex data dependencies. Historically, algorithms for multi-dimensional data processing are expressed in a serial form, either because they are meant to illustrate a way to solve the problem rather than to provide an actual implementation, or because the are originally targeted for a software execution on an instruction set processor. The serial version of such algorithms is generally inefficient and inadequate to process large amounts of data, especially at a very high rate (e.g., in video processing applications [9], [11]).

Despite the need for parallelism, the hardware design for algorithms that deal with large sets of data is still carried out

manually, and its automation is discouraged by the complex dependencies that generally exist between different elements of the data set. In this context, we present in this paper an innovative design method that goes beyond the code manipulation performed by the existing loop transformation techniques, as it is specifically tailored to a well-defined class of algorithms and to a fine-grained programmable hardware. Hence, it is able to keep into account the typical dependency schema of multimedia applications, and to exploit them to extract an efficient custom-shaped implementation. With respect to other loop transformation techniques, our approach is able to generate a circuit that directly computes the result of an arbitrary *n-th* iteration, while automatically keeping into account the required data dependencies. The ability of combining the unrolling of multiple iterations with the possibility of partitioning the input (i.e., performing the computation starting from a subset of the input matrix), and to possibly process a redundant amount of data in order to guarantee the data dependencies, make our approach innovative with respect to the existing techniques

II. Target Family of Algorithms

The design method proposed in this paper can be successfully applied to a wide set of iterative algorithms that work on matrices, which are typical in image and video processing. In fact, many algorithms in this field ([9], [11]) aim at finding an output matrix of the same size as the input (e.g., a filtered image) by using an iterative process: each iteration produces an intermediate matrix, which is computed by processing one or more elements of the result produced during the previous iteration. Algorithm 1 provides an illustrative example of this class of algorithms: the pseudo-code emphasizes the iterative behavior (i.e., the N iterations of the outermost loop), as well as the necessity to scan the entire intermediate matrix (consisting of $X \times Y$ elements) to produce the updated result.

Although the pseudo-code in Algorithm 1 might seem very specific, its structure models a large number of existing algorithms, especially in the multimedia field. For instance, all the algorithms presented in [1], [2], [4], [8] and [11] can be expressed in that form. However, if the whole algorithm has to be repeated several times starting from different input data, as it happens in [4], each execution can be considered independent from the others, since they do not share intermediate results. Thus, the approach proposed in this work is still valid under the condition that the generated system is used sequentially for each execution.

In order to formalize the considered family of algorithms, we define the following notations:

- (x, y, n) represents the element (x, y) of the intermediate matrix at iteration n;
- $G(x, y, n)$ represents the set of elements that are necessary to correctly compute the value of the element (x, y, n), in other words it corresponds to the *dependency schema* of the algorithm. Since the elements belonging to $G(x, y, n)$ are used to compute an element at iteration n, they have to be generated at iteration $n - 1$.

An algorithm can be successfully handled by the proposed methodology if the two following conditions hold, which is typically true for iterative algorithms:

1) no read-after-write (RAW) conflicts should exist within an iteration, because the first innermost loop performs the evaluation of the intermediate matrix coming from the previous iteration and stores the results in a temporary variable, while the second innermost loop updates the intermediate matrix. This means that the computation of an element at iteration n can not depend on the value of another element at iteration n, but only on previously generated elements (the ones at iteration $n - 1$);
2) for each generic element *(x,y,n)* , the set *G(x,y,n)* should only depend on the position (x, y) of the element. In other words, it should be possible to compute an element of the intermediate matrix starting from a well-defined and limited neighborhood of the element itself, taken from the intermediate matrix coming from the previous iteration.

III. Implementation Approaches

Several approaches can be adopted to design an optimized hardware system for the considered family of algorithms. In Figure 1 we are considering a quite simple example where an operation k, with two adjacent inputs ($Input_k = 2$) and a single output ($Output_k = 1$), is executed on all the elements of the input matrix (which are $X \times Y$) for four iterations ($N = 4$). Thus, each iteration of the algorithms requires a number $Z = \frac{X \cdot Y}{Output_k}$ of k operators. In this context, the implementation of all the k operators needed to execute the N iterations of the algorithm is not a viable solution in typical realistic scenarios, because of its extremely large area requirements (the area usage is directly proportional to both Z and N), which makes it almost impossible to implement this solution on actual devices for reasonable values of X and Y (e.g., almost 8 millions of k operators would be necessary to compute 10 iterations of the considered family of algorithms on a 1024×768 image).

The first approach that can be found in the literature [5], indicated with **A** in Figure 1, corresponds to the implementation of a single instance of the operator k, which is used several times in order to compute all the intermediate and final results of the algorithm, which are the (x, y, n) elements introduced in the previous section. The area usage of this first approach is very low, since only a single instance of the operator k has to be implemented in hardware. However, also the performance of this approach is low, since all the operations have to be performed in a completely sequential fashion (only a single operator is available), and its memory requirements are huge, since all the intermediate results have to be stored for the subsequent iteration.

The second approach, indicated as **B** in Figure 1, involves a *window* of k operators running in parallel [6]. This approach is widely used in the literature [7], especially on algorithms characterized by simple dependencies. However, this approach cannot be considered a viable solution when dealing with algorithms characterized by complex dependencies, since it

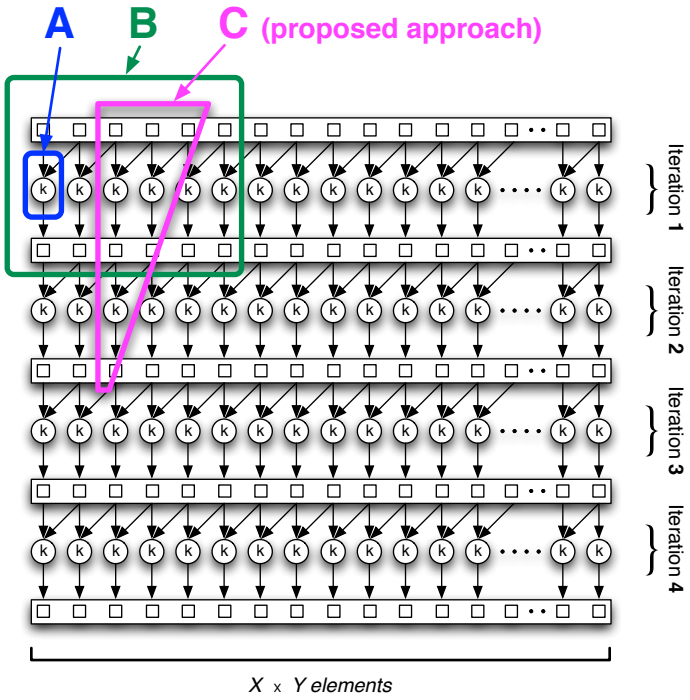

Fig. 1. Different approaches to the design of optimized hardware for the considered family of algorithms

does not take into consideration their dependency schema, thus leading to unacceptable overheads in terms of both computation time and memory requirements. In fact, if the dependencies among the iterations are not taken into account, it is not possible to ensure that all the output values of an intermediate computational step are directly used in the subsequent step, thus it could happen that some of them have to be stored for later use (memory overhead) or discarded and then computed again when necessary (timing overhead). In particular, storing all the data requires a big on-chip or off-chip memory (able to contain the whole input image) is needed. The first case is obviously very demanding in terms of on-chip memories (which are usually not sufficient to hold the whole input image), while the second one, in addition to the memory requirement, introduces a non-negligible timing overhead since each time an *old value* is needed it has to be retrieved from the external memory. A third option is to abolish the memory requirements by discarding all the values that cannot be stored anymore in the small local cache, thus computing again a value every time it is needed (an this could happens hundreds or thousands of times, depending on the complexity of the dependency schema of the target algorithm).

Finally, the approach indicated as **C** in Figure 1 is the one proposed within this research work. It deals, at each iteration, only on the subset of data, namely a collection of overlapping $G(x, y, n)$ values, necessary to compute the information needed by the following iteration. Since in most of the cases the set of elements that can be successfully computed at the iteration n is only a subset of the elements computed at the iteration n–1, the resulting structure is usually very similar to a 3D *cone*, such as in the example shown in Figure 1.

The shape of the architectures generated with the proposed approach is characterized by the size of the *input window* (w_i), the size of the output window (w_o) and the number of iterations computed by each *cone* (which is $t = \frac{N}{H}$, where H is the number of levels in which the computation of the N iterations is split). The shape of the *cones* (e.g., the relationship between w_i and w_o) strictly relies on the data dependencies schema of the selected algorithm (in other words, on the size and shape of the $G(x, y, n)$ sets).

IV. The Proposed Design Method

The methodology proposed in this paper can be successfully applied to the design of hardware architectures based on a hardware template that can be tuned to meet the constraints imposed by the designer and that is able to exploit two different levels of parallelism:

- **Coarse-Grained Parallelism**: this is the parallelism that can be extracted among different iterations of the same loop on the same elements of the data set;
- **Fine-Grained Parallelism**: this is the parallelism that can be extracted when computing different elements in the same iteration of the loop.

Assuming that the two conditions presented in Section II hold for the iterative algorithm taken in consideration, it is sufficient to analyze the instructions (in C or VHDL code) contained in the innermost loop (e.g., *sum = sum + Kernel(j,i) * Image(x-j, y -i)* in Algorithm 1) to identify the sets $G(x, y, N - t)$, one for each (x, y, N) element of w_o, which represents the inputs of the *cone*. In other words, it is possible to trace back the dependencies among the different elements (all the ones needed to compute the desired output window specified by the designer as input) at different iterations (from the first iteration to the iteration specified by the designer as input). In fact, as shown in Algorithm 2, in order to build the complete architecture, the designer can specify the size and shape of the desired output window (w_o), as a set of (x, y, n) elements, as well as the number of iterations to be implemented within a single *cone* (t).

Once the analysis of the dependencies is completed, it is possible to generate and synthesize the basic hardware *cone* able to produce the desired output window starting from the elements at the iteration $N - t$. At this point, if the final hardware *cone* includes all the iterations of the algorithm (thus if $N = t$), it can be simply executed sequentially (shifting the location of the output window in order to cover the whole input data set) to generate all the output windows necessary to build the final result. Otherwise, if only a portion of the iterations are processed by the hardware *cone* (see Figure 1), it is necessary to build the structure shown in Figure 2 in order to cover (following a multi-level approach) all the iterations required by the target algorithm, as described in the *Reverse Scheduling* procedure of Algorithm 2. In particular, Figure 2 shows the structure of an architecture consisting of *cones* spanning 4 iterations ($t = 4$), for an algorithm requiring a total of 12 iterations ($N = 12$).

In other words, in the final solution the computation of the previously introduced innermost loop is performed on a subset of elements, instead of considering the whole data set. This

Algorithm 2 Proposed Design Method

Input from the designer:

- Desired output window w_o
- Total number N of iterations
- Height t of each *cone*

Output:

- Final parallel hardware implementation of the considered multimedia iterative algorithm

Cone Generation:

w_i=$\emptyset$

for each (x_k, y_k, N) belonging to w_o **do**

$w_i = w_i \cup G(x_k, y_k, N-t)$

end for

Synthesize a *cone* able to generate w_o starting from w_i

Reverse Scheduling:

for (p = 0; p $< \frac{Size(InputMatrix)}{Size(w_o)}$; p++) **do**

Schedule 1 *cone* at level $\frac{N}{t}$ in position *p*

for (iteration = $\frac{N}{t} - 1$; iteration > 0; iteration–) **do**

Schedule all the *cones* necessary at level *iteration* in position *p*, in order to feed the *cones* already scheduled at level *iteration+1* in position *p*

end for

end for

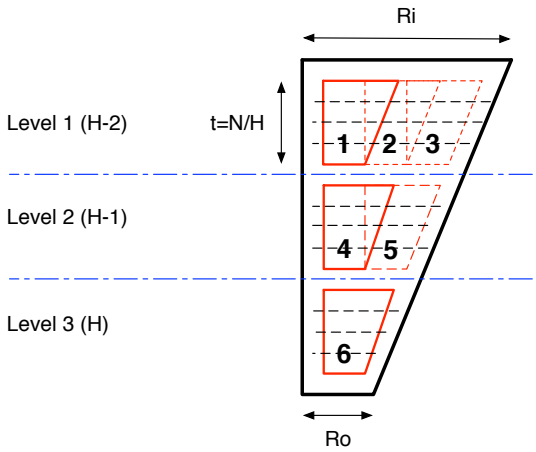

Fig. 2. *Cone* structure for $N = 12$ and $H = 3$ (thus, $t = 4$)

makes it possible to split the computation in several different tasks (implemented in hardware as *cones* that span on more iterations) that can be executed sequentially or completely in parallel, depending on the amount of resources available on the target device.

The method proposed within this paper (intended as a set of subsequent steps to be performed in order to implement the Approach C of Figure 1) can be followed by any designer in order to build a system that follows the previously described architectural template. However, we also developed a design framework that makes it possible for the designer to automatically generate the desired architecture by simply specifying the size of the output window and the number of iterations to be implemented within a single cone. Thus, the designer can exploit this tool in order to explore the design space, selecting the architecture with the highest performance among the ones that fulfill the area constraints imposed by the designer. However, the description of this design framework is out of the scope of the paper, since the method is more generic and it is valid also when applied manually by the designer.

A. Memory Requirements Analysis

The main advantage of employing the proposed approach with respect to approaches **A** and **B** of Figure 1 is that it makes it possible to exploit the knowledge of the dependency schema of the target algorithm in order to obtain high-performance architectures (as shown by the experimental results on several real-world case study, such as the one presented in Section V) with very low requirements in term of on-chip memory.

In fact, from the internal memory usage perspective, the on-chip memory requirement of the approaches **A** and **B** in their basic formulation is $2*X*Y*e$, where *e* is the size of a single element of the input. This is because it is necessary to keep into the internal memory at least 2 arrays (containing each one $X * Y$ elements): the first one stores all the intermediate values computed at the previous iteration, while the second one stores the results of the current iteration.

However, by optimizing the memory management it is possible to drastically reduce the memory required to store the intermediate results, but the total memory requirement is bound to a minimum of $X * Y * e$. Thus, the actual memory requirement *M1* of the different formulations of the approaches **A** and **B** is:

$$X * Y * e \leq M1 \leq 2 * X * Y * e. \quad (1)$$

On the other hand, the memory requirement of the proposed approach is given by the following formula:

$$(\frac{N}{H}+1)*\frac{(Si+So)}{2}+(H+1)*\frac{(Wi+Wo)}{2}-Si-So \quad (2)$$

that is based on (see Figure 2 for a graphical representation of the architecture):

- the number of levels of the whole architecture (*H*);
- the number of iterations performed by a single cone (*N/H*);
- the number of inputs (*Si*) and the number of outputs (*So*) of a single cone;
- the number of inputs (*Wi*) and the number of outputs (*Wo*) of the whole architecture.

In general, the memory requirement of the proposed approach is from 2 to 3 orders of magnitude smaller than the one of the architectures following the approaches **A** and **B**, as proved in Section V on a real-world case study.

V. Case Study: Chambolle Algorithm for Optical Flow Estimation

We successfully applied the proposed approach to several real-world case studies based on multimedia applications, such as iterative filters and complex image processing algorithms.

For the sake of illustration, one of them is presented and analyzed in this section: the *Chambolle* [4] algorithm, which is very popular in image processing (especially for the estimation of the optical flow [8]).

The *optical flow* is a vector field representing the movement 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 [3]. 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 and compensation, as well as in other fields such as robotics and even medical analysis.

From the practical point of view, the optical flow between two images I_0 and I_1, which are given in the form of two input matrices, is a vector $\mathbf{u} = (u_1, u_2)$. The vector $\mathbf{u}$ is initialized at 0, and its final value is computed by means of an iterative way in order to increase the precision of the solution [8]. The value of $\mathbf{u}$ can be determined using an iterative technique known as *Chambolle* algorithm [4], which is summarized in Algorithm 3, where the vector $\mathbf{v}$ is basically a support variable defined by using a thresholding function based on I_1 and on the value of $\mathbf{u}$ computed at the previous iteration (for the sake of simplicity, the pseudo-code only shows the computation of u_1, but u_2 is computed in a similar way).

Algorithm 3 Chambolle Algorithm

1: **for** $i = 1, .., N$ **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**

In the execution of *Chambolle*, the vector $\mathbf{u}$ is updated by two intermediate values, namely $\mathbf{px} = (px_{u1}, px_{u2})$ and $\mathbf{py} = (py_{u1}, py_{u2})$ [8], as shown in Lines 7 and 8 of Algorithm 3. Variables θ and τ are predefined values that determine the precision, while N is the number of iterations to be performed (usually set to 200). The $Backward_X(z)$ function returns a matrix where each element of z is reduced by its left neighbor, while in $Backward_Y$ it is subtracted from its upper neighbor, in $Forward_X$ it is subtracted from its right neighbor, and in $Forward_Y$ from its lower neighbor.

Before applying the design method described in this paper, it is necessary to check if the conditions presented in Section II hold:

1) no RAW conflicts exist in the *Chambolle* algorithm, since the output vector is updated by means of two intermediate values, $\mathbf{px}$ and $\mathbf{py}$;
2) according to the definition of the $Backward$ and $Forward$ operations, it can be derived that the dependencies schema is the one depicted in Figure 3. In particular, an element depends on 7 surrounding elements computed at the previous iteration, as outlined in Figure 3 (a). As a consequence, the dependencies of each element can be delimited in a 3×3 square.

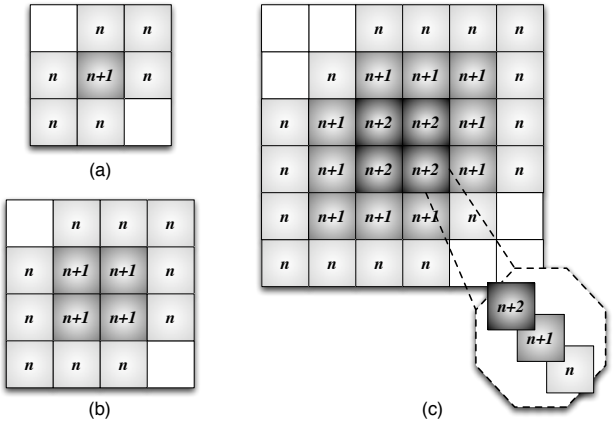

Fig. 3. Data dependencies among the elements of consecutive iterations of the *Chambolle* algorithm for 1 (a), 2 (b) or 3 (c) levels

Since all the conditions are met, the proposed approach can be successfully applied to the *Chambolle* algorithm. Among the possible solutions that can be generated thanks to the method proposed in this paper, we present here the one considering an 8×8 output window ($w_o = 64$) and able to parallelize 2 iterations in a single *cone* ($H = 5$, $\frac{N}{H} = 2$), since it is the one characterized by the best performance among all the solution of the design space we have explored. With respect to the schema presented in Figure 2, the complete architecture, which occupies around 95% of the Slices of a Xilinx Virtex 6 VLX760 FGPA, consists of 5 levels (since $H = 5$) for a total of 21 executions of the *hardware cone* (8 for Level 1, 6 for Level 2, 4 for Level 3, 2 for Level 4 and finally 1 for Level 5) in order to generate the target 8×8 window of valid final results.

In order to realize a *proof-of-concept* design, we exploited a Xilinx Virtex 6 (XC6VLX760) FPGA device as underlying prototyping platform, thus synthetizing and running on this FPGA device the system obtained by following the methods presented in this paper. Table I shows the comparison between the performance achieved by the proposed approach and the ones obtained by other state-of-the-art implementations on FPGAs or on other powerful parallel architectures, such as GPGPUs. In particular, the results presented in Table I underline the frame rate that can be achieved by the different approaches (thus considering, for each frame, both the time necessary to load the input and store the output and the time needed to perform the whole computation) when executing the Chambolle algorithm (100 or 200 iterations) on a flow of images with a resolution of 512×512 or 1024×768. In [8], 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. Additional hardware results of the execution of *TV-L*^{1} on GPUs can be also found in [9], but even the

fastest implementation cannot process more than 6 frames per second, even with just 512×512 images. Finally, the approach presented by the authors of [10] is an ad-hoc parallel solution that is able to perform the *Chambolle* algorithm with a very high frame rate on a Virtex-V FPGA device.

TABLE I
COMPARISON WITH RESPECT TO STATE-OF-THE-ART IMPLEMENTATIONS

Ref.	Device	Iterations	Image Resolution	Frame Rate (fps)
[8]	GeForce 7800 GS	100 200	512×512	2.6 1.3
[8]	GeForce Go 7900 GTX	100 200	512×512	4.7 2.3
[9]	NVIDIA GTX285 (OpenGL only)	100	512×512	5-6
[10]	Xilinx Virtex-V XC5VLX110T	200	512×512	99.1
[10]	Xilinx Virtex-V XC5VLX110T	200	1024×768	38.1
Proposed Approach	Xilinx Virtex-VI XC6VLX760	200	512×512	72
Proposed Approach	Xilinx Virtex-VI XC6VLX760	200	1024×768	24

For what concerns the memory requirement analysis, when considering 1024×768 images (*X*=1024, *Y*=768, *e*=8 Bit), the memory requirement *M1* of approaches **A** and **B** is, according to Equation 1:

$$X*Y*e = 768Kbyte \leq M1 \leq 2*X*Y*e = 1.5MByte \quad (3)$$

However, since it could be difficult to completely satisfy the internal memory requirements of approaches **A** and **B** in realistic scenarios, the number of accesses to the external memory of this kind of approaches considerably raises (in order to write and then read back the intermediate results produced once the internal memory is completely full), thus leading to a huge degradation of the performance (as shown in the results presented in Table I).

Considering the solution we have proposed in Section V for the Chambolle algorithm (with *N*=10, *H*=5, *N/H*=2, *Si*=138, *So*=64, *Wi*=674, *Wo*=64), the memory requirement *M2* of the architecture generated with the proposed design method is, according to Equation 2:

$$M2 = 2.26KByte \quad (4)$$

Thus, in conclusion, the memory requirement of the proposed solution is at least 2 orders of magnitude smaller than a generic architecture following the design approaches **A** or **B**.

VI. CONCLUSION

In this paper, we proposed a design method that can be successfully applied to all the iterative algorithms that fulfill the two conditions presented in Section II. Thus, if the target algorithm fits the *class model* presented in this paper, the proposed design method makes it possible for the designer to define a custom architecture able to execute the specified algorithm with very-high performance.

We successfully applied the proposed design method to several case studies, such as iterative filters and complex image processing algorithms, outperforming most of the alternative implementations that can be found in the literature. In particular, the experimental results presented in Section V show that the proposed approach, when applied to a real-world case study such as the one of the Chambolle algorithm, is the first one that guarantees a real-time execution without a manual optimization of the algorithm (such as the one in [10]) and at the same time it presents an on-chip memory requirement from 2 to 3 orders of magnitude smaller than the other approaches.

REFERENCES

[1] Mohammad K., Agaian S., "Efficient FPGA implementation of convolution", Systems, Man and Cybernetics (SMC) 2009, pp. 3478-3483

[2] Ben Cope, "Implementation of 2D Convolution on FPGA, GPU and CPU", Internal Report

[3] S. Behbahani et al., "Analysing optical flow based methods," IEEE International Symposium on Signal Processing and Information Technology, 2007, pp. 133 –137.

[4] A. Chambolle, "An algorithm for total variation minimization and applications," *Journal of Mathematical Imaging and Vision*, 2004.

[5] Jaiminkumar Chavda, Pranav Tank, Shrikant N. Pradhan, and Swati Jain. 2010. "Performance Measure of Image Processing Algorithms on DSP Processor & FPGA Based Coprocessor". In Proceedings of the 2010 International Conference on Advances in Communication, Network, and Computing (CNC '10). IEEE Computer Society, Washington, DC, USA, 173-176.

[6] ALTERA, "Video and Image Processing Design Using FPGAs", ALTERA White paper, 2007

[7] Dillinger, P.; Vogelbruch, J.F.; Leinen, J.; Suslov, S.; Patzak, R.; Winkler, H.; Schwan, K.; , "FPGA-Based Real-Time Image Segmentation for Medical Systems and Data Processing," Nuclear Science, IEEE Transactions on , vol.53, no.4, pp.2097-2101, Aug. 2006

[8] C. Zach et al., "A duality based approach for realtime tv-l1 optical flow," in Proc. of DAGM Symposium on Pattern Recognition, 2007.

[9] A. Weishaupt, et al., "Tracking and Structure from Motion,", available at: http://infoscience.epfl.ch/record/146572, 2010.

[10] A. Akin, et al., "A high-performance parallel implementation of the Chambolle algorithm," Design, Automation & Test in Europe (DATE), 2011

[11] Seok Lai Park, "Retinex method based on CMSB-plane for variable lighting face recognition", International Conference on Audio, Language and Image Processing, 2008. ICALIP 2008, pp.499-503