Efficient Hardware Design Of Iterative Stencil Loops

Rana, V., Beretta, I., Bruschi, F., Nacci, A. A., Atienza, D. and Sciuto, D.

Efficient Hardware Design
Of Iterative Stencil Loops

Vincenzo Rana, Ivan Beretta, Francesco Bruschi,
Alessandro A. Nacci, David Atienza, *Fellow, IEEE,* Donatella Sciuto, *Fellow, IEEE*

Abstract—A large number of algorithms for multidimensional signals processing and scientific computation come in the form of iterative stencil loops (ISLs), whose data dependencies span across multiple iterations. Because of their complex inner structure, automatic hardware acceleration of such algorithms is traditionally considered as a difficult task.

In this paper, we introduce an automatic design flow that identifies, in a wide family of bidimensional data processing algorithms, sub-portions that exhibit a kind of parallelism close to that of ISLs; these are mapped onto a space of highly optimized ad-hoc architectures, which is efficiently explored to identify the best implementations with respect to both area and throughput. Experimental results show that the proposed methodology generates circuits whose performance is comparable to that of manually-optimized solutions, and orders of magnitude higher than those generated by commercial HLS tools.

Index Terms—High-level synthesis, performance optimization, field-programmable gate array (FPGA), dataflow synthesis, embedded systems, multimedia processing, iterative functions.

I. INTRODUCTION

Stencil computing is an important pattern used in a large variety of domains, including multimedia processing [10], [18], [21], and discrete scientific algorithms [5], [9]. These applications rely on regular kernels that consume most of the execution time, showing both iterative nature and complex data dependencies that make them difficult to accelerate using traditional hardware and software methods [5], [44].

Stencil kernels come in the form of an iterated function T applied over a multidimensional signal f (a *frame*). The iterated function is obtained by repeatedly composing a transformation tr with itself:

$$f_1 = tr(f), f_2 = tr(f_1), ..., f_n = tr(f_{n-1}) = T(f)$$

Typically, the desired $T(f)$ is a fixed point of the single step transformation $tr : tr(T(f)) = T(f)$. In this case, the ideal output of the process is the fixed point to which the transformation converges starting from the initial frame. This class of algorithms is known in the literature as *Iterative Stencil Loops* (ISLs) [6], and has been analyzed within the compiler community to find efficient implementations targeted to CPUs [6] and Graphic Processing Units (GPUs) [7].

V. Rana, F. Bruschi, A. A. Nacci and D. Sciuto are with DEI - Politecnico di Milano, Via Ponzio 34/5, Milano, Italy. E-Mail: {vincenzo.rana, francesco.bruschi, alessandro.nacci, donatella.sciuto}@polimi.it

I. Beretta and D. Atienza are with ESL - Ecole Polytechnique Fédérale de Lausanne (EPFL), ESL-IEL-STI-EPFL, Station 11, Lausanne, Switzerland. E-Mail: {ivan.beretta, david.atienza}@epfl.ch

The design of dedicated hardware accelerators for ISL algorithms, on the contrary, still represents an unsolved challenge, as no automatic flow to date can guarantee high-performance implementations, mainly because of the inherently complex data dependencies.

Standardized approaches exist for the acceleration of iterative algorithms – a broad class that also includes ISLs – on Field Programmable Gate Array (FPGA) devices. The typical structure includes two frame buffers [1] [2] [3], A and B, and the logic to compute the transformation tr. The initial frame is loaded in one of the buffers, and then the following iteration is computed and stored in the other buffer (f (in A) $\overset{tr}{\to} f_1$ (in B) $\overset{tr}{\to} f_2$ (in A), ...). The procedure continues until the desired number of iterations has been performed. However, this simple architecture shows a substantial shortcoming: area and on-chip memory required are lower bounded by the frame size, making it too costly in real-world conditions.

In this work, we propose a High-Level Synthesis (HLS) flow that addresses the problem of automated hardware acceleration of ISL algorithms, combining architectural aspects and a novel algorithm analysis technique. The rationale behind the proposed methodology stems from this observation: some algorithms feature a peculiar form of *spatial locality*, where the value of each element p at iteration $i + 1$ (p_{i+1}) depends only on a small number of elements in the neighborhood of p at iteration i (p_i). This feature has been instrumental to develop compiler techniques, known as tiling [45] [46], for algorithm manipulation. In this work, we leverage the flexibility of reconfigurable hardware to push this locality even further, exploiting it to generate custom modules that work on a portion of the frame, and that output a subset of the intermediate results used by the subsequent iterations. Suppose, for example, that we want to compute a single element p of the final resulting matrix, obtained after n number of iterations (let us call it p_n). The value of p_n depends on a set $P_{n-1} = \{p_{n-1}^1, ... p_{n-1}^m\}$ of elements computed at iteration $n - 1$. Propagating these data dependency relations backwards until the input frame, we obtain the domain of the function that computes p_n. Since ISL algorithms are *uniform* over the whole domain, such function is uniquely determined by the number of levels we want to traverse and, inspired by its geometric representation (see Figure 1), we call it a *cone* of *depth n*. We can generalize this concept considering cones that compute a set P_n of elements of the n-th iteration: in this broader definition, a cone is also characterized by the set of output points P_n. In the remainder of the paper, we will refer to the set P_n of a cone as its *output window*.

It can be observed that the desired processing can be performed by repeatedly applying a cone to portions of the
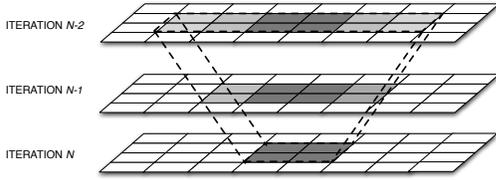
Fig. 1. A cone of depth 2 and window size 4

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)
            Temp(x,y) = sum
    for (x = 0; x $< X$; x++) **do**
        for (y = 0; y $< Y$; y++) **do**
            Image(x,y) = Temp(x,y)

---

input matrix. This approach leads to hardware implementations whose on-chip memory requirements are *independent of the frame size*. The parameters that define a *"cone-based" architecture* are the *cone window size* (for the sake of illustration we consider a square window), the *cone depth*, and the *number of cones* simultaneously present in hardware. Finding the best architecture (i.e. the best combination of cones) that satisfies a specified constraint, such as a minimum frame rate, is a challenging task, due to the large number of trade-offs at stake that make the space search complex. In this work, we address this problem by proposing an efficient estimation of area and throughput of a given cone architecture, thus avoiding to actually synthesize them, which would require dozens of hours for realistic window size and cone depth values.

The proposed methodology starts from the hardware structure we proposed in [41] and, following the design principles we first outlined in [43] and [44], complements our preliminary research efforts in this field by defining a novel methodology to automatically identify ISLs even if these are present only in a subsection of the input algorithm.

II. TARGET FAMILY OF ALGORITHMS

Many image and video processing algorithms [10], [21] aim at finding an output matrix of the same size as the input (e.g., a filtered image) by means of an iterative process: each step produces an intermediate matrix, which is computed by processing one or more elements produced during the previous iteration. Algorithm 1 provides an illustrative algorithm belonging to this class: 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 [3], [13], [15], [16], [18], [20] and [10] can be expressed in that form, as well as algorithms

for scientific computation, such as convolution and the *Jacobi* iterative algorithm to solve linear eigenvalue problems [17].

Even when the whole algorithm has to be repeated multiple times starting from different input data, as it happens in [18], each execution can be considered an independent instance of the pattern, unless they share intermediate results. To formalize the properties that characterize the members of the considered family of algorithms, let us define the following notations:

- let (x, y, n) represent the element (x, y) of the intermediate matrix at iteration n;
- let $G(x, y, n)$ represent the set of elements that are necessary to correctly compute the value of the element (x, y, n), that is the *domain* of (x, y, n) (when some regularity conditions are met we will refer to G as to the *dependency schema* of the algorithm). Since the elements belonging to $G(x, y, n)$ are those required to compute an element at iteration n, they have to be generated at iteration $n - 1$.

Now, let us define the following properties:
1) *Uniform Dependencies* [48]: the shape of $G(x, y, n)$ is independent of (x, y). This property is the key of stencil computing, and states that the relative position of the elements that are necessary to compute any (x, y) do not change across the input matrix.
2) *Domain Narrowness*: the elements of $G(x, y, n)$ are a proper subset of the input matrix S, with $|G(x, y, n)| << |S|$. The traditional definition of uniform dependency [48] does not explicitly exclude a situation where all the elements of S are necessary to compute an element (x, y). Although our approach can handle such pathological case, we argue that a more interesting scenario arises when the cardinality of $G(x, y, n)$ is smaller than the cardinality of S, because in this case computation can be split and parallelized.
3) *Uniform Inter-Iteration Dependencies*: for each iteration n, and for each element (x, y), $G(x, y, n) = G(x, y, n + 1)$, i.e. the dependency pattern remains the same at every iteration.

Remarkably, among the algorithms that satisfy these three conditions, there is the family of *Iterative stencil loops* (ISL) algorithms [5]–[8], that iteratively apply the same core operations (the *stencil*) on uniform patterns of dependent data. The number of iterations can either be known in advance (as, for instance, in an iterative convolution filter [13], where the amount of desired blur corresponds to a number of filtering steps), or potentially unbounded (as in fixed point algorithms, where one would ideally iterate until an equilibrium is reached).

III. STATE-OF-THE-ART IMPLEMENTATIONS

Let us now focus on the design of optimized circuits for the algorithms that exhibit the properties of *domain narrowness*, *uniform dependencies* and *uniform inter-iteration dependencies*. Known approaches consider mostly the family of *Iterative Stencil Loops*, which is a proper subset of the algorithms characterized by the three properties. The most relevant are compared in Table I with respect to their area usage, performance and memory requirements. The main bottlenecks are shown as shaded grey cells: the criticality of the bottleneck grows with the color darkness.
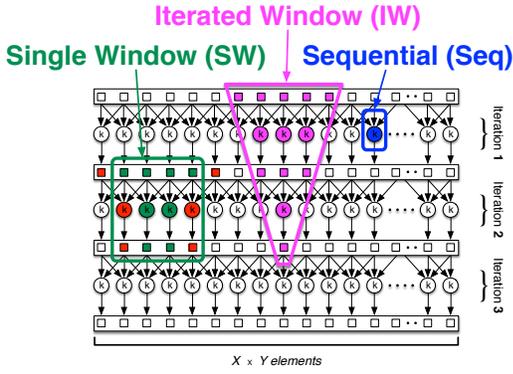
Fig. 2. Different approaches to the design of optimized hardware for ISLs

A. State-of-the-Art Methodologies: A Taxonomy

In order to explain the main differences between the state-of-the-art approaches and the proposed one, let us introduce a simple example: Figure 2 shows an algorithm where an operation k, whose area usage is A_k and whose execution time is E_k, characterized by $I_k = 3$ adjacent inputs and $O_k = 1$ output, is executed on all the $X \times Y$ elements of the input matrix for $N = 3$ iterations. Thus, each iteration of the algorithm requires $Z = \frac{X \cdot Y}{O_k}$ operators of type k. In this context, the implementation of all the k operators required to execute the N iterations of the algorithm is not viable in typical realistic scenarios, because of its extremely large area requirements. In fact 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., 8 millions of k operators would be necessary to compute 10 iterations on a 1024×768 image.

In [32] and [33], indicated as *Sequential (S)* in Figure 2, the authors propose 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, which are the (x, y, n) elements introduced in the previous section. As reported in Table I, the area usage of this approach is very low, since only a single instance of the operator k has to be implemented in hardware. However, the performance of this approach is low as well, since all the operations have to be performed strictly sequentially by the single operator available, and its memory requirements huge, because all the intermediate results have to be stored for the subsequent iteration.

The second approach, indicated as *Single Window (SW)* in Figure 2, aims at taking full advantage of parallel execution by instantiating a *window* of k operators processing multiple inputs coming from the same iteration [10], [35]. The parameter w represents the width of the *window*, which is basically the number of k operators working together at the same time, while S_i and S_o represent the set of input and output elements, respectively. This approach is widely used in literature [31], [34], [36], [37], 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 does not take into account the relations between successive frames, and therefore it is generally suboptimal when multiple

iterations are performed at once. In fact, if the dependencies among the iterations are not considered, it is not possible to ensure that all the output values of an intermediate step are directly used in the following one, thus some of them have to be stored for later use (memory overhead) or discarded and then computed again when necessary (timing overhead).

Finally, the approach indicated as *Iterated Window (IW)* in Figure 2 is the one proposed in the present work and it is shaped to match the dependencies of the algorithm to be implemented. Therefore, its main goal is not to maximize the number of parallel processing elements, but rather to span across different iterations, and ensure that each computed element can be immediately reused. To this end the proposed approach deals at each iteration only with 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 cases the set of elements that can be successfully computed at iteration n is only a subset of the elements computed at iteration $n - 1$, the resulting structure is usually very similar to a 3D *cone*, as shown in Figure 2.

The resulting architecture 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* (t). This value can be computed as $t = \frac{N}{H}$, where N is the total number of iterations and H is the number of cones that are necessary to produce a valid output starting from a set of input values, through all the N iterations. The shape of the cores (e.g., the relationship between w_i and w_o) employed depends on the dependencies schema of the selected algorithm, thus on the size and shape of the $G(x, y, n)$ sets. The proposed flow automatically derives the best fitting ones for each algorithm.

B. Evaluation and Comparison of Existing Implementations

As described in the previous section and shown in Table II for an instance of the ISL family, the algorithms considered here have traditionally been a challenging problem for the designers, mainly because of the complexity of their data dependencies. This is especially true when considering execution time, as proved by the results shown in Table II, which includes the best performing implementations of the Chambolle algorithm. Most of the state-of-the-art solutions fail in achieving real-time performance on reasonable images (with a size $\geq 512 \times 512$) even on GPGPUs platforms. In the literature, the problem of designing efficient implementations for this class of algorithms has been tackled within the compiler community using the concept of loop tiling [45] [46], a platform-independent technique that divides the iteration space into blocks, aiming to maximize data reuse and parallelism. Our approach takes full advantage of tiling principles, and combines them with the benefits provided by fully custom computation on fine-grained reconfigurable platforms.

Exploiting the potential of FPGA devices for stencil computing is a relatively new research direction. While existing implementations of ISLs on CPUs [5] [6] and GPGPUs [7] [8] have ultimately struggled achieving high performance, ground-breaking works on FPGAs (such as [47]), have demonstrated high potential. In fact, CPUs and GPGPUs have rigid architectures in terms of memory organization, which may not map

TABLE I
COMPARISON AMONG DIFFERENT APPROACHES. THE MAIN BOTTLENECKS ARE SHOWN AS SHADED GREY CELLS.

Approach	Pipeline	Memory type	Area	Execution time	On-chip memory
Seq (e.g., [32], [33])	No	Off-chip	A_k	$N \cdot Z \cdot E_k$	$I_k + O_k$
	No	On-chip	A_k	$N \cdot Z \cdot E_k$	$2 \cdot X \cdot Y$
SW(w) (e.g., [34], [35], [10], [31], [36], [37])	No	Off-chip	$w \cdot A_k$	$\frac{Z}{w} \cdot N \cdot E_k$	$S_i + S_o$
	No	On-chip	$w \cdot A_k$	$\frac{Z}{w} \cdot N \cdot E_k$	$2 \cdot X \cdot Y$
IW(w_i, w_o, H)	No	On-chip	$\frac{w_i + w_o}{2} \cdot \frac{N}{H} \cdot A_k$	$\frac{E_k \cdot Z \cdot N \cdot (R_i/S_i + 1)}{2 \cdot w_o}$	$\frac{R_i + R_o}{2} \cdot (H+1)$
	Yes	On-chip	$\frac{w_i + w_o}{2} \cdot \frac{N}{H} \cdot A_k$	$\frac{E_k \cdot Z \cdot H \cdot (R_i/S_i + 1)}{2 \cdot w_o}$	$\left(\frac{N}{H} + 1\right) \frac{S_i + S_o}{2} + \frac{R_i + R_o}{2} \cdot (H+1) - S_i - S_o$

well on all algorithms. FPGAs overcome these limitations, but they nonetheless show well known limitations in terms of area and external memory accesses, which requires the hardware implementing the stencil algorithm to be carefully designed. Our approach addresses this requirement, by performing a meticulous analysis based on both area and overall throughput.

Commercial alternatives to translate algorithms to hardware exist on the market. HLS tools like Xilinx Vivado [23] or Synopsys Symphony C Compiler [22] are commonly used to perform a set of predefined array and loop transformations, such as loop unrolling, merging, flattening, pipelining and array partitioning, on the C description of the input algorithm. However, they are inherently *general purpose* and can apply only generic optimizations without specifically exploiting the peculiarities of the specific algorithm. For this reason the performance of the implementations generated by these frameworks are generally unsatisfying for ISLs, especially when compared to manually optimized implementations.

Given the lack of support for the automatic generation of custom hardware designs for ISLs, many *ad-hoc* implementations have been proposed for specific ISL algorithms. For example, [4] proposes an optimized implementation for non-iterative 2D convolutions, and [41] provides an efficient hardware approach for Chambolle [18]. However, since these solutions are manually tailored for a specific algorithm, they lack generality and reusability, and the effort required to adapt one of these solutions to a different problem (if possible) is generally not negligible.

Our previous research efforts in the field of ISLs have shown that high-performance implementations can be obtained by combining a dependency analysis from the C code [44], and a hardware architecture specifically designed for iterative kernels [43]. In this work, we complement these considerations to obtain a complete and automated HLS flow. Specifically, we herein include techniques to detect an ISL structure within a give C code, and to perform a deep design space exploration to implement it in hardware.

IV. THE PROPOSED ARCHITECTURE TEMPLATE

In order to automatically discover whether a given algorithm presents these three characteristics defined in Section II, we introduced in Section V-A symbolic execution as a means to automatically extract its data dependencies. This technique makes it possible to express the result of the $(i+m)$-th iteration as a function of (some of) the elements computed at the i-th iteration. Then, given the data available from the i-th iteration, instead of trying to compute the whole f_{i+1}, we can focus on a

TABLE II
STATE-OF-THE-ART IMPLEMENTATIONS OF CHAMBOLLE

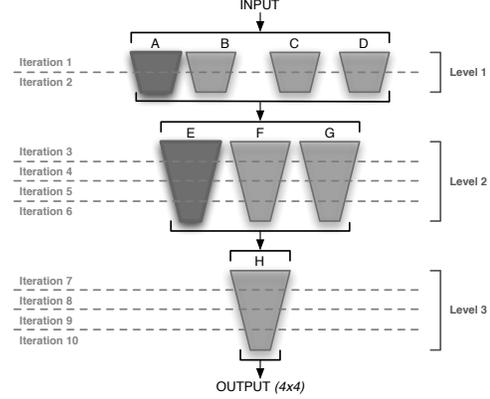| Ref.	Device	Iterations	Image Resolution	Frame Rate (fps)
[20]	GeForce 7800 GS	50; 100; 200	256×256	17.5; 9.6; 5
[20]	GeForce 7800 GS	50; 100; 200	512×512	5; 2.6; 1.3
[20]	GeForce Go 7900 GTX	50; 100; 200	256×256	34.1; 17.5; 8.9
[20]	GeForce Go 7900 GTX	50; 100; 200	512×512	9.3; 4.7; 2.3
[21]	ATI m Radeon HD3650	100	512×512	1-2
[21]	ATI m Radeon HD3650	100	512×512	3-4
[21]	NVIDIA GTX285	100	512×512	5-6

Fig. 3. A simple example of a mixed *cone*-based architectural template

subset of the matrix elements and directly compute the results of a generic $(i+m)$-th iteration (with $m \geq 1$), thus obtaining a subset of f_{i+m}. We refer to the core that performs such multi-iteration computation as a *cone* of *depth* m.

We define an *architectural template* as the combination of multiple levels of *cones* that can compute the result of multiple iterations of the elementary transformation tr. The combinations of *cones* of different depths are also considered to cover all the required iterations and explore the different area/throughput trade-offs. These structures, as shown in Figures 3 and 4, work as follows: a small subset of the input data is transferred from the off-chip to the on-chip memory, consisting of multi-port block RAMs, to feed the cones of the first level of the architecture (A, B, C and D in Figure 3). The output of each level is then stored again in the on-chip memory, so that it can be used as input for the subsequent level without the need of performing data transfer from/to the external memory. Finally, the output of the last level (Level 3 in Figure 3) is sent back to the off-chip memory and the whole process starts over on a different window of the input, until the final result has been computed.

The number and depth of the cones in the actual architecture

has to be tailored to the algorithm under consideration, since the dependencies can significantly vary from algorithm to algorithm. Thus, multiple *instances* of the template may exist, and each one is uniquely characterized by the size of the output window of each *cone* and the number of levels in which the computation is divided. Figure 3 shows an instance of the template with an output window of 4×4 elements and 3 levels of computation. The only requirement for an instance to be feasible is that, if cones of different depths are required, at least one cone of each depth must be implemented on the device. For instance, the structure in Figure 3 is feasible if the available resources are sufficient to fit cones A and E.

The proposed template provides an efficient datapath structure for iterative stencil loops, which can be coupled with a simple control logic provided with a standard memory interface. This interface can be used to fetch data from the external memory and forward it to the *cones* and then to store back in the external memory the output of the computation. The only information that the control logic needs in order to perform these tasks is the size of the chunks of data to be read/written from/to the external memory (i.e., the size of the input/output windows of the *cone* architecture). Figure 4 shows a practical example of how a final architecture looks like, using the Chambolle algorithm as an example. For the sake of illustration, the computation has been assigned to only two *cones* that span 2 iterations, and produce an output window of 8×8 pixels (Figure 4.1). The *cones* are then repeated multiple times to cover the required number of iterations (Figure 4.2), thus producing the full datapath. The control logic is very simple and has a negligible complexity with respect to the rest of the design, as it is only in charge of incrementally computing the result by sliding the output windows of the *cones* over the area of the output frame. This is achieved by feeding the two *cones* with the corresponding (possibly overlapping) portions of the input frame, as shown in Figure 4.3. The block view of the complete architecture, and the corresponding RTL implementation on a FPGA device, are finally illustrated in Figure 4.4.

V. THE PROPOSED HLS DESIGN FLOW

The HLS methodology proposed in this paper can be applied to the design of an optimized hardware architecture exploiting two different kinds of optimizations:

- *Cone Collapsing*: a sequence of elaboration steps performed on a window of the input frame is computed with a single hardware component (a *cone*);
- *Horizontal Parallelism*: different elements of the same frame are elaborated in parallel by different cones.

The core idea is to perform the computation of the innermost loop on a subset of elements, instead of considering the whole data set. This in turn makes it possible to split the computation into several different tasks, implemented in hardware as cones that span on more iterations and that can be executed completely in parallel. The main drawback arises from the overhead introduced on the edges between the different subsets, which depends both on the number of iterations of the algorithm and on the shape of the subsets. However, the proposed approach tries to endow the designer
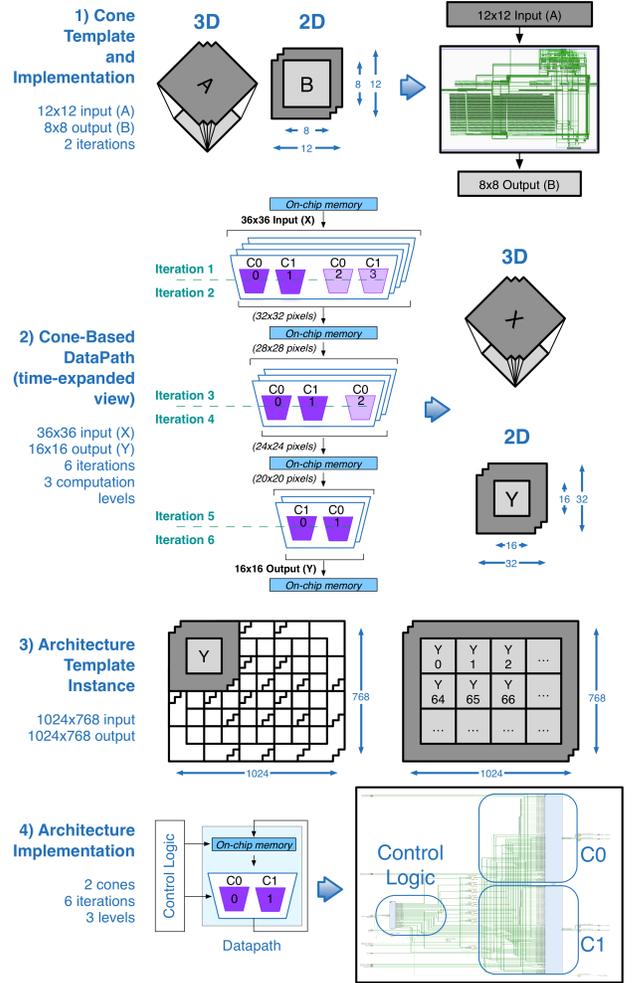

Fig. 4. An example of an architecture for the Chambolle algorithm

with tools to explore the design space and identify the best trade-off between the parallelization of the computation and the overhead generated.

Relying on this idea, the design flow hereby proposed starts from the C description of a given algorithm, analyzes its dependencies and detects whether it can be effectively implemented with a cone-based architecture. The output of the flow is the VHDL description of the Pareto-optimal (area vs. throughput) implementations of the given algorithm, in the vast solution space of all the instantiations of the structural template described in Section IV.

A. Symbolic Execution

To understand if the architecture template proposed in the previous section can be successfully adopted for the implementation of a specific algorithm, it is necessary to discover if the latter exhibits *domain narrowness*, *uniform dependencies* and *uniform inter-iteration dependencies*, at least for a portion of the *input frame* and for a subset of the *computation iterations*.

Our approach exploits **symbolic execution** to automatically extract useful information on the structure and the characteristics of the input algorithm. To generate the *cones*, it is necessary to express the value of an element $p \in f_{i+m}$ as a function of a set of elements of the frame produced at the i-th iteration (i.e., f_i). This functional relation is often computed by

hand (such as in [41]) but, when different numbers of levels are evaluated during the design space exploration, the advantages of an efficient and automatic way to determine the equations for $m = 1, ..., N$ are straightforward.

The first analysis phase is an optimized *symbolic execution* of a C description[1] of the input algorithm, where symbolic expressions are propagated, rather than the actual values of the variables. Thus, output of the iterations from i to $i + m$, is not the numeric value of f_{i+m}, but a set of equations that relate each element of f_{i+m} to a subset of elements of f_i.

To automatically perform symbolic execution with minimum instrumentation effort we exploited C++ (ISO/IEC 14882:2003) templating and overloading capabilities. The key idea is to use templates to define parametric "symbolically executable" types. Then we overload all the relevant operations upon such classes (+, -, *, etc.) so that, while executing the instrumented code, the results of arithmetic operations are logged along with their *symbolic values* (e.g., if $a = 3$ and $b = 4$ the arithmetic result of the $a \cdot b$ is 12, while the symbolic value of the same operation is $a \times b$). In this way, the information that allows to completely reconstruct the data flow is recorded within the variable containing it, which is directly accessible at the end of the computation. A symbolically tracked variable carries information on:

- its arithmetical actual value;
- the index of the iteration it belongs to;
- possibly, information about its location within the data set (usually matrix indexes);
- its symbolic value;
- history of its previous (symbolic and arithmetical) values.

The iteration indexes and the history of all its previous values are used to compare equations at different frames of the symbolic execution, in order to detect and measure *uniform inter-iteration dependencies*.

Exploiting C++ language features keeps the instrumentation overhead to a minimum. The modifications to be performed on the input C code are:

- the modification of the type declaration of the variables to be tracked (in a sense, the declaration section of the algorithm acts as an analyzer configuration);
- the initialization of all the arrays to be tracked. This initialization does not involve the values of the elements of the arrays, but it is needed to store, in each one of those elements, the indexes of their location within the array itself. For instance, the symbolic variable corresponding to an element $matrix[x][y]$ would contain, along with the other data, also the two indexes x and y;
- the (highly-automatable) definition of new overloaded functions – one for each custom or library function to be tracked by the symbolic analysis – that take as input symbolic values instead of the original ones.

Below, a snippet of the definition of the symbolic integer:

```
class symb_int: public symb_element{
```

---

[1]There are no "strict" syntactic constraints on the language used to describe the algorithms. Some language features, though, can generate code for which symbolic execution is less effective, such as accessing arrays by using pointer arithmetic and dereferentiation instead of square brackets.

```
public:
  int n, level, store_n[MAX_LEVEL];
  string svalue, id, temp_svalue;
  string store_svalue[MAX_LEVEL];
  [...]}
```

Let's consider, as an example, the overloaded + operator:

```
symb_int& operator+(symb_int op1, symb_int op2){
  [...]
  ss1 << "(" << op1.svalue <<"+"<< op2.svalue<<")";
  ss2 << "(" << op1.id << "_+_" << op2.id << ")";
  temp->n       = op1.n + op2.n;
  temp->svalue  = ss1.str().c_str();
  temp->id      = ss2.str().c_str();
  temp->store_n[op1.level]=op1.n;
  return *temp;}
```

The stringstream *ss1* is used to propagate the symbolic value (*svalue*) corresponding to the result of the sum operation by concatenating the two *svalue* variables of the two input *symb_int*. Similarly, *ss2* is used for the propagation of the *id*. Finally, the variable *n* holding the actual value of the sum is computed by adding the two numeric values in input.

In addition to overloaded operators, we have also developed a mechanism to keep track of function calls:

```
int funct(int n); //custom or library function

symb_int funct(symb_int n){ //automatically
    generated
  n.funct("funct",funct(n.n));
  return n;}

void funct(const char*fname="f",int value=0){
  this->n=value;
  stringstream ss1, ss2;
  ss1 << fname << this->svalue;
  ss2 << fname << this->temp_svalue;
  this->svalue=ss1.str();
  this->temp_svalue = ss2.str();}
```

The mechanism consists in automatically defining, for each (custom or library) function used in the given algorithm, a new overloaded function that takes as input a symbolic value instead of the original one (e.g., a *symb_int* instead of a simple *int*) so that every time a function is called on a symbolic value, the function defined by our design flow is invoked instead of the one defined by the user or present in the library. The main tasks performed by the functions generated by the flow are:

- to keep track of the computational flow (i.e., the operations applied to the symbolic variable);
- to retrieve the return value by invoking the original function with the actual value of the symbolic variable as parameter.

In this way, not only simple operations can be tracked by the proposed symbolic execution, but also complex functions defined by the users or imported from the standard libraries.

B. Data dependencies analysis

The output of symbolic execution is then processed to estimate its level of *domain narrowness*, *uniform dependencies* and *uniform inter-iteration dependencies*. As shown in Figure 5, symbolic execution produces a dependencies schema for each element of the data matrix and for each iteration. This dependencies schema is expressed as a formula, where $elem[a][b]\_n$ identifies the element located at coordinates
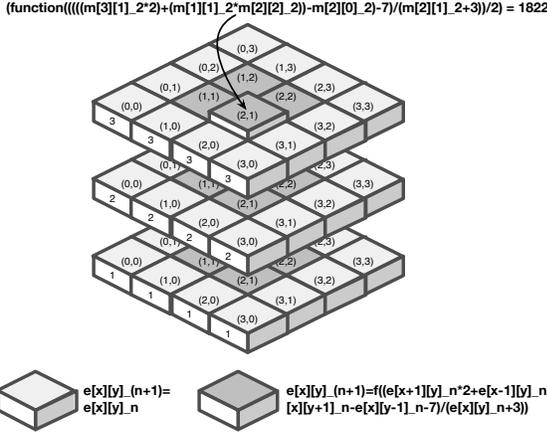
Fig. 5. Graphical representation of the data of the Python analyzer

(a, b) at the iteration level n. We are interested in discovering if a subset (ideally consisting of all the elements) of the data matrix is characterized by the same formula for a certain number of iterations (ideally all), without involving any dynamic control flow (e.g. data-dependent conditionals).

To this end, we developed a *python* tool that identifies the maximum subsets of the domain within which *domain narrowness*, *uniform dependencies* and *uniform inter-iteration dependencies* hold. As shown in Figure 6, the tool produces both a graphical representation of the three properties for each iteration (2 images are produced for each iteration: 1 for *domain narrowness* and 1 for *uniform dependencies/uniform inter-iteration dependencies*) and the corresponding structured description. The graphical representation is useful for the designer to understand at a glance if (and roughly how much) the considered algorithm can benefit from the proposed implementation approach, while the structured description is fed to the following phases of the flow to generate the correct architecture for the algorithm considered.

Domain Narrowness Analysis. In order to evaluate the presence, level, and location of *domain narrowness* in the algorithm, the tool has to check the following fundamental condition: the value of each element at iteration level L should only depend, directly or indirectly (e.g., function calls), on constant values (propagated by the symbolic execution) or on values of other elements computed at previous iteration levels (including the initial values), but not, for instance, on the values of the elements at the same iteration level.

Then the tool has to identify, for any given point P of every step n, what are the points in $n-1$ that P depends upon (i.e. its *domain*). At this point, in order to estimate the *domain narrowness*, a metric that takes into consideration both the size and the shape of the different *domains* has to be evaluated. In our approach, we have decided to assign, to each point P' in the domain of P at iteration $n-1$ ($domain(P)$), a *penalty* w that is directly proportional to the distance (e.g., Cartesian distance, Manhattan distance, etc.) between P and P':

$$w_{PP'} = distance(P, P')$$

Then, the aggregation (e.g., the sum) W of all the w penalties is a measure of the inverse of the domain narrowness (DN) of point P ($DN(P)$):

$$W = \sum_{P' \in domain(P)} w_{PP'} = \frac{1}{DN(P)}$$

Finally, the output is stored for the following phases of the design flow and a graphical representation is produced where the darker is the color of a cell, the higher is the number and the penalties of the elements of its dependencies schema (that is, the less it is characterized by the *domain narrowness* property); in particular, a white cell identifies an element that only depends on itself (at the previous iteration), while a black cell identifies an element that does not satisfy the *domain narrowness* property (e.g., it depends on other elements at the same iteration level). The exact values of the *domain narrowness* of each element is coded in the picture as a scale of grey, while it is stated explicitly in the textual representation of the output. Figure 5 shows that the tool is able to detect that all the elements of the matrix have high *domain narrowness* at all iteration levels, and also that the border depends only on one element of the previous iterations, while the other cells have a larger dependencies domain.

Dependencies Analysis. To estimate the *uniform dependencies* and the *uniform inter-iteration dependencies* properties, the first task performed by the tool is the generation, starting from the input formulas, of a tree representing the dependencies of each element of the input matrix, for each level of iteration. Then, all the syntax trees of the elements of each iteration level are compared in order to partition them into classes, each of whom is characterized by the same syntax tree, (that is, by the same dependencies schema). In order to perform a fair comparison among all the syntax trees, they are pre-processed in the following way:

- the absolute indexes of the array members of the syntax tree are transformed into indexes relative to the target element. Let us consider the following syntax trees:
  - the first one is $array[5][7] + array[5][8]$ and refers to the element $array[5][7]$;
  - the second one is $array[6][2] + array[6][3]$ and refers to the element $array[6][2]$;

  Even though the two syntax trees are different, they can be both expressed (by using indexes relative to the element $array[x][y]$) as $array[x][y] + array[x][y+1]$, and thus they can be correctly detected as equivalent;
- the resulting formula is simplified, in order to facilitate the detection of equivalent expressions. For instance, the operations among actual values are performed and the result is stored in the tree (e.g., $3*2+1$ is substituted with the corresponding result, 7, so that the two syntax trees $a+b+3*2+1$ and $a+b+2*2+3$ can be correctly detected as equivalent);
- each formula is normalized (by taking into account the lexical order of the operands) in order to convert all its possible representations into a canonical form (so that the two syntax trees $a+b$ and $b+a$ can be correctly detected as equivalent).

An example of the output of this phase is shown in Figure 5, where two distinct classes (each one characterized by cells of the same color) of elements are detected:

- class A, characterized by: $e[x][y]\_(n+1) = e[x][y]\_n$;

- class B, characterized by: $e[x][y]\_(n+1) = f((e[x+1][y]\_n*2+e[x-1][y]\_n*[x][y+1]\_n-e[x][y-1]\_n-7)/(e[x][y]\_n+3))$.

The *partition* of the matrix in the equivalence classes of points that have domains with equal shape is an inverse measure of the *uniformity* of the dependencies schema: if all the pixels (over all the different iterations) have same shaped domains, there will be only one class of equivalence. On the other hand, if they are all different, the number of pixels becomes equivalent to the number of classes (which is the situation with the lowest *uniformity* possible).

Note that the metrics proposed can detect the presence of the properties even in subparts of the algorithm, and to different degrees. This significantly widens the set of algorithms that can be addressed by the proposed synthesis flow. Moreover, it is not necessary that the properties are evident in the algorithm code upon inspection, since the tool will exploit the information coming from the symbolic execution.

For instance, consider the following synthetic example (which is not a simple ISL where a single stencil is repeated on the whole data for all the iterations):

```
for (i=0;i<N;i++){
  for(x=0;x<DIM;x++)
    for(y=0;y<DIM;y++)
      if(x>0 && x<DIM-1 && y>0 && y<DIM-1
      && x<(DIM-1)/2 && i>=(N-1)/2)
        m2[x][y]=funct((m1[x+1][y]*2+m1[x-1][y]*
          m1[x][y+1]-m1[x][y-1]-7)/(m1[x][y]+3));
      else if(x>0 && x<DIM-1 && y>0 && y<DIM-1
      && x>DIM/2 && i>=(N-1)/2)
        m2[x][y]=m1[x+1][y]+m1[x-1][y]-funct(m1[x][y]);
      else if(x>0 && x<DIM-1 && y>0 && y<DIM-1 &&
        x>DIM/2)
        m2[x][y]=(m1[x+1][y]+m1[x-1][y])/2+m1[x][y];
      else
        m2[x][y]=m1[x][y];
  for(x=0;x<DIM;x++)
    for(y=0;y<DIM;y++)
      if(x>0 && x<DIM-1 && y>0 && y<DIM-1)
        m1[x][y]=m2[x][y]/2;
      else
        m1[x][y]=m2[x][y];}
```

When executed on an input data matrix of $15x15$, it produces the output shown in Figure 6, where it is possible to observe that:

- the first 2 iterations considerably differ from the other 3;
- the *domain narrowness* is verified by all the elements at all the iteration levels, even though with different levels of locality (3 different classes are identified by the tool);
- without considering the borders, the first two iterations satisfy both *uniform dependencies* and *uniform inter-iteration dependencies* in the whole data set, while in the last three iterations the tool automatically identifies two huge subsets where the two properties locally hold;
- the borders are correctly detected as "special elements", so that they can be handled as a separated class of elements by the proposed approach, since they satisfy the *domain narrowness* and *uniform (inter-iteration) dependencies* properties for all the 5 iterations.

C. Cones Generation

As shown in Algorithm 2, the design flow exploits the information about the maximum size of the data matrix subsets that
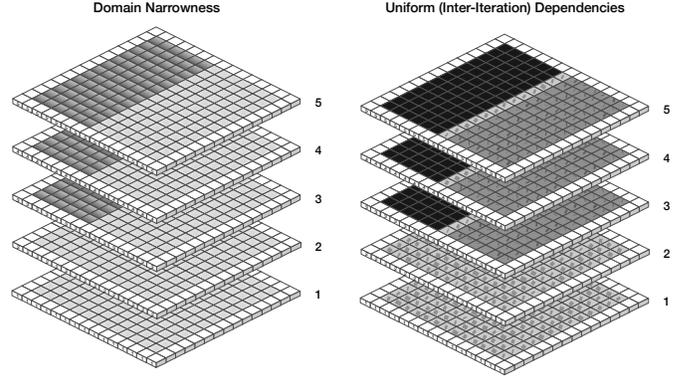

Fig. 6. Example of a possible graphical output of the python analyzer

---

Algorithm 2 Cones Generation

Input (from the *symbolic execution phase*):
- Sets of output window sizes w_o
- Possible heights t of the computational *cone*

Output (to the *architectures generation* phase):
- Equation and code implementing the sets of possible *cone* structures

$w_i=\emptyset$

for each (x_k, y_k, N) belonging to w_o **do**
   $w_i = w_i + G(x_k, y_k, N-t)$

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

---

locally satisfy both *domain narrowness* and *uniform (inter-iteration) dependencies* on a certain number of consecutive iterations to generate all the possible *cones*. These cones are characterized by a higher number of output elements (w_o) that spans different numbers of iterations (which is $t = \frac{N}{H}$). For each combination of output window (w_o) and cone height ($\frac{N}{H}$), a specific *computational cone* is implemented. To derive the input window (w_i) from w_o, it is possible to proceed by identifying the sets $G(x, y, N-t)$, one for each (x, y, N) element of w_o, which represents the inputs of the *cone*. After this phase, it is possible to generate and synthesize the hardware *cone* able to produce the desired output window starting from the elements at the iteration $N-t$.

The main issues that arise during the synthesis of a *cone* is the exponential growth, while performing symbolic execution, of the number of symbols included in the expressions, that makes it impractical for complex algorithms. In the proposed flow, we overcome this issue by exploiting the properties defined in Section II, which enable an efficient symbolic execution for the targeted class of algorithms. First, it is not necessary to find an equation for *all* the elements of f_{i+m}: if the *uniform dependencies* property holds, the dependencies of the elements in the frame are uniform, which allows tracking only one element in order to get the desired expressions for the whole f_{i+m}. Second, if the *uniform inter-iteration dependencies* property holds data dependencies between two consecutive iterations i and $i+1$ are the same for each value of $i \in \{1, ..., N-1\}$. As a consequence, it suffices to perform symbolic execution for just one iteration to find the relation between f_{i+1} and f_i, which in turn can be used as a building block to compute the dependencies between any pair of f_{i+m}
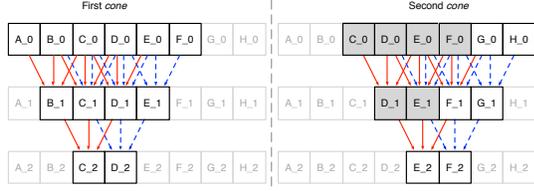
Fig. 7. Example of the data-reuse technique

Algorithm 3 Architectures Generation

Input (from the *data dependencies analysis* phase):
- Total number N of iterations
- Dependencies structure of the algorithm

Input (from the *cones generation* phase):
- Equation and code implementing the sets of possible *cone* structures

Output (to the *design space exploration* phase):
- Sets of possible parallel hardware implementations
- of the considered multimedia iterative algorithm

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

Fig. 8. *Cone* structure and scheduling for $N = 12$ and $H = 3$ (thus, $t = 4$)

and f_i during the VHDL generation.

The equations returned by the symbolic execution are exploited to automatically generate a synthesizable VHDL description of the cones. During the equations-to-VHDL translation, the exponential explosion of the number of symbols is avoided by enforcing data reuse. In fact, a large number of operations on the same elements is repeated multiple times to satisfy the data dependencies, as shown in the example in Figure 7. The first *cone*, in fact, in order to compute its output window consisting of C_2 and D_2, would require to compute some intermediate results multiple times: for instance, C_1 and D_1 would have been computed 2 times, B_0 and E_0 3 times, while C_0 and D_0 5 times. This redundancy is not detected by the symbolic execution itself, which would instead introduce a large number of repeated symbols and operations in the equations. In our flow, we handle it by unrolling the dependencies between f_{i+m} and f_i through m iterations and, for each operation between two elements, we store the result in a register: whenever the operation appears more than once, the register is reused (as a sort of *cache*). This generates a VHDL code with a high degree of resource reuse, which can be handled by any synthesis tool for FPGAs.

D. Architectures Generation

Algorithm 3 describes how it is possible to generate the space of all the possible architectures (intended as combinations of different *cones*) that implement the given input algorithm. When generating an architecture with a specific set of *cones*, two different scenarios are possible, depending on the height of the considered *cones*.

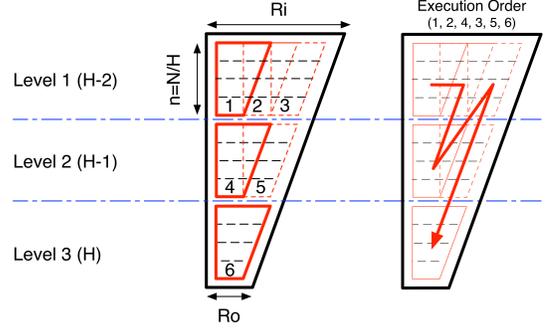If at least one of the *cones* works on all the iterations of the algorithm (thus if $t = N$ for that *cone*), then it is sufficient to deploy on the target device one or more of those *cones* (depending on the available area) that can work in parallel on different portions of the input data. Then, as soon as their computation is concluded, the *control logic* simply stores their output and feeds them with other portions of the input data, shifting the location of their output window in order to cover the whole output data set.

Otherwise, if all the considered *cones* only span a subset of the iterations (see e.g. Figure 2), then the flow has to combine more *cones* to produce a single output window of the final iteration starting from the input data fetched by the *control logic*. The resulting architecture is similar to the one shown in Figure 8, where 6 *cones* spanning $t = 4$ iterations are necessary to cover all the $N = 12$ iterations of the algorithm. As stated in Algorithm 3, the first *cone* to be considered is *cone 6*. Then *cones 4 and 5* are instantiated in order to generate the data required by *cone 6*. Finally, *cones 1 and 2* are required by *cone 4*, while *cones 2 and 3* by *cone 5*. The execution order the flow exploits to perform the whole computation is handled by the *control logic* and is then the following: 1, 2, 4, 3, 5, 6. Note that the output of intermediate *cones*, such as *cone 2*, is temporary stored in the on-chip memory in order to avoid multiple executions of the same computation.

For what concerns the area and computation overhead, Figure 7 shows what happens when two or more *cones* are executed on adjacent output windows. While the area and computation overhead within a single cone is detected and resolved during the *cones generation* phase (see Section V-C), this does not happen at the *architectural* level. As shown in Figure 7, in fact, a portion of the computation (consisting of C_0, D_0, E_0, F_0, D_1 and E_1 in this case) is performed in both the cones, thus leading to area and computation overhead. This redundancy is an essential and unavoidable part of our approach, which makes it possible to split the computation and distribute it on a set of cores working in parallel. The percentage of this overhead can be computed with the following formula:

$$\frac{(Op_{cone-based} - Op_{initial}) \cdot 100}{Op_{initial}} \quad (1)$$

where $Op_{cone-based}$ is the number of operations effectively performed by the cone-based architectures generated with our flow and $Op_{initial}$ is the number of operations of the initial algorithm (which computes one iteration at a time). The *domain narrowness* parameter is a good proxy to estimate

this overhead, since it directly depends on the shape of the dependencies schema of the input algorithm. In addition to this, the computational overhead can be minimized by tuning some of the parameters that characterize each architectural solution, such as:

- increasing the size of the output window of each cone;
- reducing the height of each cone (i.e., the number of iterations it computes).

In particular, experimental results have shown that the computational overhead never exceeds 50% in all the cone-based architectures characterized by cones with an output window of more than 7×7 pixels and spanning less than 5 iterations. However, in order to automatically select the best trade-offs among redundant computation, on-chip memory requirements, area usage and performance, the proposed flow performs a deep design space exploration. over all the instances of the proposed architectural template, as described in Section V-E.

E. Design Space Exploration

The main issues that arise when trying to extract the Pareto-optimal architectures (see Figures 11 and 15) from all the ones generated in the previous phase of the flow (as described in Algorithm 3), is the evaluation of the cost and throughput of each architecture of the solution space. This would indeed require, in principle, to synthesize each possible combinations of *cones*: for typical problem sizes this may take days of CPU time, making a complete design space exploration unfeasible. As an example, in a simple scenario with *cones* spanning 1, 2, 4, 5 or 10 iterations, considering architectures obtained combining 2 different kinds of *cones* with output windows ranging from 1×1 to 100×100 elements, the solution space consists of $5 \times 5 \times 100 = 2500$ possible architectures.

To address this issue we hereby propose a novel technique that quickly (generally in the orders of a few minutes) estimates the area requirements of all the cone architectures. The proposed evaluation only requires a very small number (as low as two) of circuit syntheses (considering only a single *cone* for each synthesis), and its accuracy is related to the number of syntheses that the designer is willing to perform (the higher the number, the more accurate the estimation). Describing the area requirements analytically presents several challenges, the main one arising from the non-linear growth of the area with respect to the number of cones in the architecture, due to the optimization and the logic reuse performed by the synthesis tool. However, we observed that the trend of the area occupation follows the growth of the number of registers allocated into the cones. We captured the observed trend with the following relation:

$$A_x^{est} = A_{x-1}^{est} + (Reg_x - Reg_{x-1}) \cdot Size_{reg} \cdot \alpha \quad (2)$$

Where A_x^{est} is the estimated area requirement for an architecture whose cones have an output window consisting of x elements. Reg_x represents the number of registers that have been used to build the HDL of a *cone* with an output window consisting of x elements: this quantity is known as soon as the VHDL description of the algorithm is generated and data reuse is enforced. $Size_{reg}$ represents the average size (typically 32-bit) of the registers allocated during the generation of the

HDL of the *cone* architecture. Finally, the α correction factor takes into account the degree of logic reuse performed by the synthesis tool, which can be experimentally evaluated by interpolating two initial syntheses X and Y in the following way (if a higher accuracy is needed, more initial synthesis can be performed):

$$\alpha = \frac{A_X - A_Y}{(Reg_X - Reg_Y) \cdot Size_{reg}} \quad (3)$$

However, we have observed in our experiments that the results of two synthesis are generally sufficient to obtain a value of α that makes it possible to perform very accurate estimations, as proved in Section VI. On the other hand, appraising throughput follows the traditional approach of summing the delays of the operations included in each cone, and counting the number of cones running in parallel. This information is immediately available, as well as the information about latency, after the VHDL generation phase. Once the architectures space is completely characterized (that is, area and throughput of each possible implementation have been estimated) the flow finally extracts the Pareto set, exhaustively exploring a set of a (typically) few hundreds/thousands solutions.

VI. EXPERIMENTAL RESULTS

As mentioned in Section I, we validated proposed flow on different case studies, of which we discuss the most significant two, characterized by different complexity: an iterative gaussian filter [13] and the Chambolle algorithm [18].

A. Iterative Gaussian Filter (IGF)

The first case considered is the *blur* effect, essentially consisting in convolving an image f with a Gaussian kernel G. Convolution is a fundamental mathematical operation that is used in many common image processing operators [10]. In image processing, it is used to implement operations in which the values of the output pixels are calculated as linear combination of a subset of the input pixels.

Convolution belongs to a class of algorithms commonly referred to as *spatial filters*: a kernel (or *mask*) is moved across the original image, and each pixel is computed as the weighted sum of the neighboring elements, where the weights are the values in the mask. The semantic of the weights in the mask depends on the operation to be performed on the original image. For example, a blurring filter can be obtained by taking the kernel values from a gaussian distribution. In the case of blurring filters, and more generally in gaussian kernels, the convolution with a large kernel can be approximated by an iterative application of a smaller kernel [11].

The first task performed by the proposed design flow is the symbolic execution of the input algorithm, immediately followed by the data dependencies analysis phase. In order to show how it is possible to automatically extract, from the input algorithm, information about the portions of code that can be optimized thanks to the proposed architecture, we have performed the symbolic execution and the data dependencies analysis phases on a piece of code performing five iterations of the IGF applications, thus consisting of the following tasks:

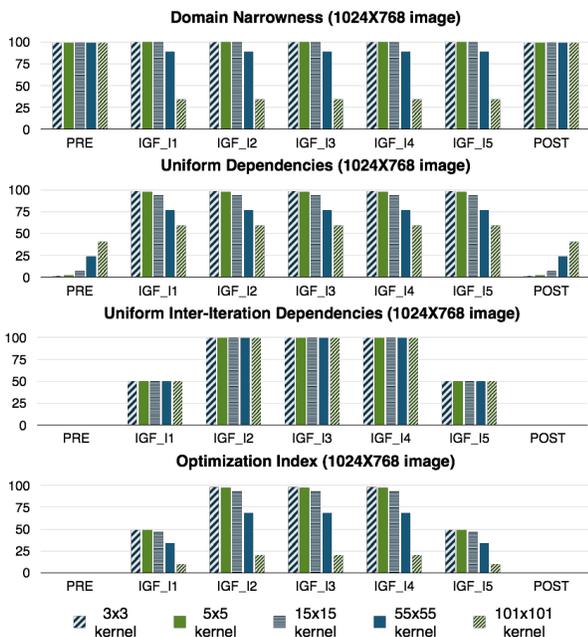- PRE: generic not-parallelizable pre-processing;

Fig. 9. Domain narrowness, uniform (inter-iteration) dependencies and optimization index estimation for the IGF on different kernel sizes

- IGF_In: $n - th$ iteration of the IGF;
- POST: generic not-parallelizable post-processing.

Figure 9 summarizes the results obtained by the proposed design flow when provided with the same IGF running on 1024×768 images with kernels of different sizes (each bar of Figure 9 refers to a different kernel size: 3×3, 5×5, 15×15, 55×55 or 101×101). The level *DN* of *domain narrowness* has been computed as 100 - the percentage of elements belonging to the dependencies schema of an element, averaged on all the elements of the data structure. Thus, bars close to 100 correspond to computational iterations that, for a particular kernel size, show a very high level of *domain narrowness*. The level *UD* of *uniform dependencies* has been computed as the percentage of elements that share the same dependencies schema within each computational iteration (in case multiple sets of such elements exist, the larger one has been selected): a value close to 0 represents a situation in which almost each element is characterized by a different dependencies schema, while a value close to 100 refers to a scenario in which almost all the elements share the same dependencies schema. The level *UIID* of *uniform inter-iteration dependencies* has been computed by taking into consideration, for each iteration i, both the previous $(i-1)$ and the subsequent $(i+1)$ iterations (if existing): then, if the three iterations perform the same computation (i.e., they would produce the same output if fed with the same input data), the *UIID* of iteration i is set to 100. If only two of them (i and $i-1$, or i and $i+1$) perform the same computation, the *UIID* of iteration i is set to 50. Otherwise, the *UIID* of iteration i is set to 0. Finally, an *optimization index* has been computed as $\frac{DN*UD*UIID}{10000}$ in order to estimate how much a particular configuration of image and kernel size is suitable for the proposed optimizations (the higher, the better). Of course other more complex aggregation functions can be used in place of the proposed one, even though in all our

experiments a simple multiplication among the three metrics has proved to be sufficient to correctly identify the iterations to be processed with the proposed flow.

As can be seen in Figure 9, the iterations of the IGF have been correctly recognized by the proposed design flow as portions of code that can be optimized with the proposed technique (high *optimization index* values). In addition, the *optimization index* provides the designer with an estimate of how much the different algorithm tasks are suitable for the proposed synthesis. The *optimization index* corresponding to the iterations of the IGF decreases with the size of the kernel, while it grows with the size of the input image.

Once the algorithm portion that is most susceptible to optimization has been selected, the flow proceeds with the generation of the different cones and with the architecture space exploration, which in turn requires performance and area estimation of each possible configuration. To test the precision of the estimation technique, we previously performed most of the syntheses and compared them with the corresponding estimations [44]: results are presented in Figure 10, w.r.t. different output window sizes and number of iterations. The maximum estimation error is 6.58%, and the average error is 2.93%, suggesting that the proposed model provides a very accurate evaluation without requiring a full synthesis. Let us now analyze the Pareto set of optimal cone architectures. Figure 11 shows the Pareto curve with respect to performance (in this case, the time to process a single frame) and area requirements (i.e., the number of slices on a FPGA), for the convolution of a 1024×768 image. The set of Pareto solutions is reported into the zoomed window.

If the design is targeted to a specific FPGA device, and hence the amount of resources is known in advance, the proposed design flow tries to identify, by exploiting the estimated area usage of each kind of *cone*, the number of *cone* instances that would fit the target device. In this way, the flow is able to estimate the maximum throughput achievable for each kind of *cone*, as shown in Figure 12. In particular, this chart shows the throughput variation on a Xilinx Virtex-6 XC6VLX760 FPGA when the size of the output window is varied. The cones that lead to best performances are those whose depth is a divider of the number of overall iterations (in the example, 10 iterations are best performed with cones of depth 1, 2 and 5). The reason why cones of depth 3 and 4 perform worse is that they are not dividers of 10, causing allocation of additional specific cones (of depth 1 and 2, respectively) to implement the remaining iterations, and making the exploitation of the available area suboptimal. Even by considering a single cone depth, the trend reported in Figure 12 is not monotone because, although larger cones typically lead to better throughputs, it may happen that smaller cones allow to better fit the device area. It is finally worth noting that, among all the implementations, the area required by the control logic never exceeds 2.3% of the resources required by the full system.

A comparison between our cone-based solutions and the ones presented in the literature shows a significant speed-up on the same device (or a comparable one) and with a similar resource usage. For instance, [16] presents a 20 iterations convolution with a 3×3 kernel working on a Xilinx Virtex-
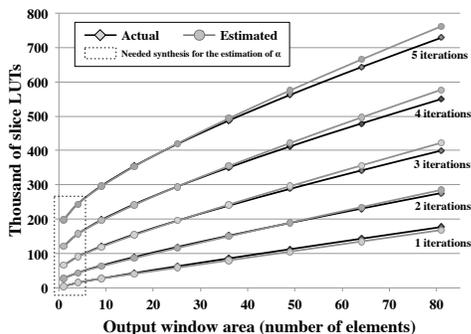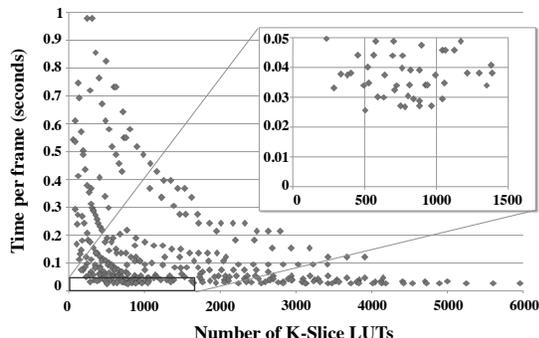
Fig. 10. IGF area estimation

Fig. 11. IGF Pareto curve (image size: 1024×768)

II Pro at 13.5fps with 1024×768 images and at less than 5fps with Full-HD images, while our architecture is able to achieve, on the same FPGA device, up to 35fps on Full-HD images. With a more modern FPGA such as a Virtex 6, our architecture is able to reach 110fps on 1024×768 images.

B. Chambolle Algorithm

For what concerns Chambolle [18], we have applied the proposed design flow to a portion of the optical flow algorithm, consisting of the following steps:

- Acquisition: data acquisition and filtering phase;
- PRE: generic not-parallelizable pre-processing;
- Chamb_In: $n-th$ iteration of Chambolle;
- POST: generic not-parallelizable post-processing;
- Visualization: data rendering and visualization phase.

The results are presented in Figure 13. Even though the data acquisition and visualization phases are parallel by construction, since they work on each element of the data matrix without taking into account the values of the other elements, the flow is able to automatically detect that the phases that
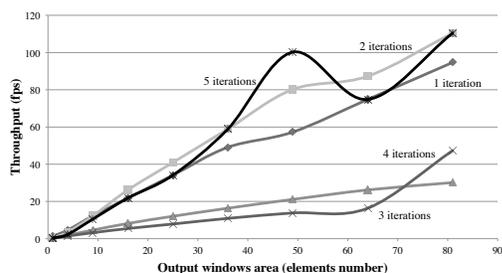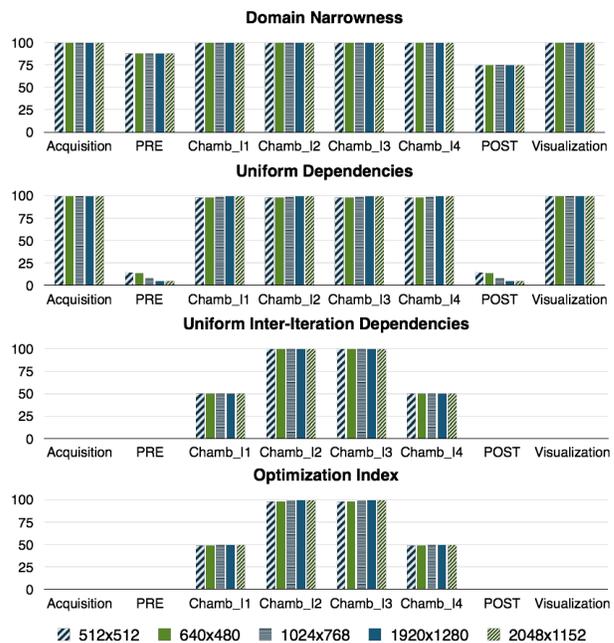

Fig. 12. IGF throughput (image size: 1024×768)

Fig. 13. Domain narrowness, uniform (inter-iteration) dependencies and optimization index estimation for Chambolle on different image sizes
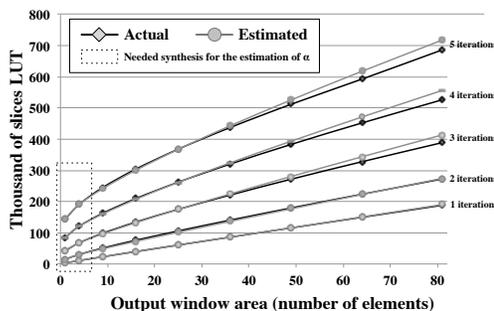

Fig. 14. Chambolle area estimation

present a high value of the *optimization index* (computed as described in Section VI-A) are only the four iterations of the Chambolle algorithm, whose dependencies schemas are then selected to be the input of the following steps of the synthesis flow (*cones and architecture generation* and *design space exploration*).

As for the gaussian filter, we initially estimated the area of each possible cone architecture for Chambolle and compared the results with respect to the actual synthesis results [44]. Figure 14 reports the results, which are again very accurate, as the maximum area estimation error we observed is 6.36%, and the average one is 2.19%.

After characterizing the architectures, we extracted the Pareto curve illustrated in Figure 15. When a specific FPGA is targeted, the behavior of the throughput is similar to that discussed for the iterative filter. In this example, it can be observed that the best solution in terms of throughput is not the one with the largest output window (9×9), but rather the solution with 8×8 cones, since in this case 2 instances of the cone can be deployed simultaneously on the device (see Figure 16). The performance of the proposed architectures are competitive with respect to state-of-the-art implementations.
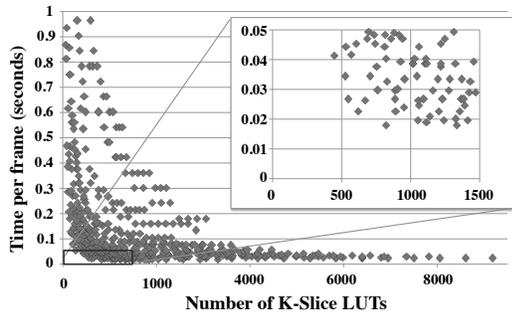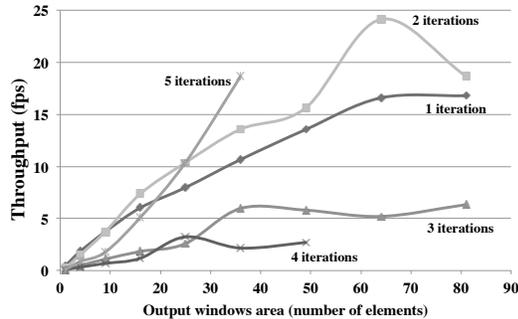
Fig. 15. Chambolle Pareto curve (image size: 1024×768)

Fig. 16. Chambolle throughput (image size: 1024×768)

required area. Since the framerate metric implicitly captures the (bad) side effects of computational redundancy (such as resource wasting, throughput reduction, etc.), this shows how well redundancy is compensated by the proposed approach.

For example, the architectures in [3], [20] and [21], are unable to reach the real-time threshold (i.e., $30fps$) even on small images because of their intrinsic lack of parallelism.

As a baseline comparison, we implemented the Chambolle kernel using Vivado HLS, a state-of-the-art commercial tool. The loop manipulation techniques offered by the tool (flattening, unrolling and pipelining) are not effective on the complex stencil shape, leading to an implementation that processes one pixel at a time. The corresponding hardware requires a relatively low area ($33kLUTs$), but performs poorly in terms of throughput ($0.14fps$). Another comparison point we considered is the manually-optimized implementation discussed in [41]. The work represents a reasonable upper bound, because it features a partial loop unrolling, but also a set of application-specific optimizations – such as an approximated LUT-based square root – that cannot be replicated by an automatic HLS tool. The hardware in [41] reaches $38fps$ on 1024×768 images and $99fps$ on 512×512 resolutions, but it required several months of manual design efforts. Our flow *automatically* obtained comparable results – $24fps$ on 1024×768 images and $72fps$ on 512×512 images – using an automated procedure, thus positioning itself as an ideal tradeoff between performance and productivity.

VII. Concluding Remarks

In this paper, we consider the synthesis of a wide set of iterative bidimensional data processing algorithms on custom hardware platform. We propose both a novel architectural template and an analysis tool that automatically extracts subparts of the algorithm susceptible of optimization.

Experimental results show that the performance of the solutions synthesized are at least comparable to (and, in some cases, significantly better than) state-of-the-art manual implementations, both in performance (in terms of fps) and

References

[1] D. Crookes and K. Benkrid, "FPGA implementation of image component labelling", in Reconfigurable Technology: FPGAs for Computing and Applications, SPIE vol 3844, 17- 23 (1999)

[2] K. Benkrid, S. Sukhsawas, D. Crookes, and A. Benkrid, "An FPGA-based image connected component labeller", in Field-Programmable Logic and Applications. Springer Berlin, 1012- 1015 (2003)

[3] T. Pock et al., "A duality based algorithm for TV-L1 optical-flow image registration," in Proc. of *MICCAI*, 2007, pp. 511–518.

[4] J. Fowers at al., "A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications", *FPGA '12*, pp. 47-56

[5] M. Christen , "PATUS: A Code Generation and Autotuning Framework for Parallel Iterative Stencil Computations on Modern Microarchitectures", *IPDPS 2011*, pp. 676-687

[6] Z. Li and Y. Song, "Automatic tiling of iterative stencil loops", *ACM Trans. Program. Lang. Syst.* 26, Nov. 2004, pp. 975-1028.

[7] J. Meng and K. Skadron, "Performance modeling and automatic ghost zone optimization for iterative stencil loops on GPUs.", *ICS*, 2009, pp. 256-265.

[8] J. Meng and K. Skadron, "A Performance Study for Iterative Stencil Loops on GPUs with Ghost Zone Optimizations", International Journal of Parallel Programming, 2011, 39, pp. 115-142.

[9] J. Holewinski, L. Pouchet, and P. Sadayappan, "High-Performance Code Generation for Stencil Computations on GPU Architectures", *In Proc. of the ACM ICS '12*, pp. 311-320, 2012.

[10] D. V. Rao et al. , "Implementation and evaluation of image processing algorithms on reconfigurable architecture using c-based hardware descriptive languages," *JATIT*, vol. 1, pp. 9–34, 2006.

[11] Y. Park et al., "A new method of illumination normalization for robust face recognition," in *Progress in Pattern Recognition, Image Analysis and Applications*, Springer, 2006, vol. 4225, pp. 38–47.

[12] S. L. Park, "Retinex method based on cmsb-plane for variable lighting face recognition," in Proc. of *ICALIP*, 2008, pp. 499 –503.

[13] E. Jamro et al., "Convolution operation implemented in FPGA structures for real-time image processing," in Proc. of *ISPA*, 2001, pp. 417–422.

[14] C. Charoensak and F. Sattar, "A single-chip FPGA design for real-time ica-based blind source separation algorithm," in Proc. of *ISCAS*, 2005, pp. 5822–5825, vol. 6

[15] K. Mohammad and S. Agaian, "Efficient FPGA implementation of convolution," in Proc. of *SMC*, 2009, pp. 3478 –3483.

[16] B. Cope, "Implementation of 2D Convolution on FPGA, GPU and CPU," Master's thesis, Imperial College, London, 2006.

[17] L. Gerard et al., "A Jacobi-Davidson Iteration Method for Linear Eigenvalue Problems," SIAM Review , Vol. 42, No. 2, 2000, pp. 267-293.

[18] A. Chambolle, "An algorithm for total variation minimization and applications," *J. of Mathematical Imaging and Vision*, vol. 20, pp. 89–97, 2004.

[19] J. C. King, "Symbolic execution and program testing," *Commun. ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[20] C. Zach et al., "A duality based approach for realtime TV-L1 optical flow," *DAGM conference on Pattern recognition*, 2007, pp. 214–223.

[21] A. Weishaupt et al., "Tracking and Structure from Motion," Master's thesis, EPFL, 2010.

[22] Synopsys, "Synphony C Compiler," 2012.

[23] Xilinx Inc., "Vivado Design Suite User Guide, HLS", UG902, 2012

[24] 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.

[25] S. Sun et al., "Motion estimation based on optical flow with adaptive gradients," International Conference on Image Processing 2000, vol. 1, pp. 852 –855

[26] S. Lin, et al., "An optical flow based motion compensation algorithm for very low bit-rate video coding," ICASSP 1997, vol. 4, pp. 2869–2872

[27] S. Kim et al., "Mobile robot velocity estimation using an array of optical flow sensors," in Proc. of ICCAS 2007, pp. 616 –621.

[28] S. Behbahani et al., "Analysing optical flow based methods," IEEE ISSPIT, 2007, pp. 133 –137.

[29] M. B. Taylor, et al. "The Raw Microprocessor: A Computational Fabric for Software Circuits and General-Purpose Programs." IEEE Micro 22, 2, pp. 25-35.

[30] A. V. Aho, et al., "Compilers: Principles, Techniques, and Tools (2nd Edition)," Addison Wesley, 2nd edition, 2006.

[31] G. Rudy et al., 2010. "A programming language interface to describe transformations and code generation". Languages and compilers for parallel computing, Springer-Verlag, Berlin, Heidelberg, 136-150.

[32] J. Chavda, P. Tank, S. N. Pradhan, and S. Jain. 2010. "Performance Measure of Image Processing Algorithms on DSP Processor & FPGA Based Coprocessor". Advances in Communication, Network, and Computing, pp. 173-176.

[33] Yao Si-Cong; Zhang Zhi-Jiang; Wen Wei; Zeng Dan; , "DSP image process with optimized data schedule model," Computer Science and Information Technology (ICCSIT), vol.1, pp.393-398, July 2010

[34] Draper, B.A.; Beveridge, J.R.; Bohm, A.P.W.; Ross, C.; Chawathe, M., "Implementing image applications on FPGAs," Pattern Recognition, 2002. Proceedings, pp. 265- 268 vol.3, 2002

[35] ALTERA, "Video and Image Processing Design Using FPGAs", 2007

[36] P. Dillinger et al, FPGA-Based Real-Time Image Segmentation for Medical Systems and Data Processing, Nuclear Science, 2006, v. 53, n. 4, p. 2097-2101

[37] Steffen Klupsch et al, "Real Time Image Processing based on Reconfigurable Hardware Acceleration", Proceedings of IEEE Workshop Heterogeneous reconfigurable Systems on Chip, 2002

[38] Pineo, P.P., "An efficient algorithm for the creation of single assignment forms", System Sciences, 1996, Vol.1, pp. 213-222

[39] Stoutchinin A., de Ferriere F., "Efficient static single assignment form for predication", Microarchitecture, 2001, pp. 172-181

[40] M. Abutaleb et al, "A reliable fpga-based real-time optical-flow estimation," in Proc. of NRSC 2009, pp. 1 –8.

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

[42] Biemond, J.; Lagendijk, R.L.; Mersereau, R.M.; , "Iterative methods for image deblurring," Proceedings of the IEEE, vol.78, no.5, pp.856-883, 1990

[43] Rana, V.; Nacci, AA; Beretta, I; Santambrogio, M.D.; Atienza, D.; Sciuto, D., "Design Methods for Parallel Hardware Implementation of Multimedia Iterative Algorithms," Design & Test, IEEE , vol.30, no.4, pp.71,80, 2013.

[44] A. A. Nacci, V. Rana, F. Bruschi, D. Sciuto, I. Beretta, and D. Atienza. 2013. A high-level synthesis flow for the implementation of iterative stencil loop algorithms on FPGA devices. DAC 2014, Article 52.

[45] Beletska, A.; Bielecki, W.; Cohen, A.; Palkowski, M.; Siedlecki, K., "Coarse-Grained Loop Parallelization: Iteration Space Slicing vs Affine Transformations,", in Proc. of ISPDC '09., pp. 73-80, 2009.

[46] Krishnamoorthy, S.; Baskaran, M.; Bondhugula, U.; Ramanujam, J.; Rountev, A.; Sadayappan, P., "Effective automatic parallelization of stencil computations," in Proc. of PLDI '07, pp. 235-244, . 2007.

[47] Pouchet, L. N.; Zhang, P.; Sadayappan, P.; Cong, J., "Polyhedral-based data reuse optimization for configurable computing," in Proc. of FPGA '13, pp. 29-38, 2013.

[48] Karp, R. M.; Miller, R. E.; Winograd, S., "The Organization of Computations for Uniform Recurrence Equations," Journal of the ACM, volume 14, issue 3, pp. 563-590, July 1967.

Alessandro A. Nacci received his Bachelor, Master of Science and PhD in Computer Engineering from the Politecnico di Milano respectively in September 2009, July 2012 and January 2016. During his master he made two internship period as a visiting student at the Swiss Federal Institute of Technology in Lausanne (EPFL). During his PhD he has been a visiting graduate student at University of California at San Diego where he worked on tools for smart and complex buildings. His research focuses on mobile devices, Internet of Things and design methodologies for smart spaces.

Vincenzo Rana received his Laurea Specialistica in Computer Engineering in 2006 and his PhD in Information Engineering in 2010 from Politecnico di Milano, Italy. He is currently a Research Associate at the Dipartimento di Elettronica, Informazione e Bioingegneria (DEIB) of the Politecnico di Milano, Italy. His research interests include embedded systems design methodologies and architectures, reconfigurable and quantum computing, Networks-on-Chip and neural networks.

David Atienza (M'05-SM'13-F'16) is associate professor of electrical and computer engineering, and director of the Embedded Systems Laboratory (ESL) at the Swiss Federal Institute of Technology in Lausanne (EPFL), Switzerland. He received his MSc and PhD degrees in computer science and engineering from UCM, Spain, and IMEC, Belgium, in 2001 and 2005, respectively. His research interests include system-level design methodologies for high-performance multi-processor system-on-chip (MP-SoC) and low-power embedded systems, including new 2-D/3-D thermal-aware design for MPSoCs, ultra-low power system architectures for wireless body sensor nodes, HW/SW reconfigurable systems, dynamic memory optimizations, and network-on-chip design. He is a co-author of more than 200 publications in peer-reviewed international journals and conferences, several book chapters, and five U.S. patents in these fields. He has earned several best paper awards and he is (or has been) an Associate Editor of IEEE TC, IEEE D&T, IEEE T-TCAD, IEEE T-SUSC and Elsevier *Integration*. He was the Technical Programme Chair of IEEE/ACM DATE 2015 and General Programme Chair of IEEE/ACM DATE 2017. Dr. Atienza received the IEEE CEDA Early Career Award in 2013, the ACM SIGDA Outstanding New Faculty Award in 2012, a Faculty Award from Sun Labs at Oracle in 2011, and was Distinguished Lecturer (period 2014-2015) of IEEE CASS. He is an IEEE Fellow and Senior Member of ACM.

Ivan Beretta received his MSc in computer engineering from Politecnico di Milano, and in computer science from the University of Illinois at Chicago. He received his PhD in electrical engineering from the Swiss Federal Institute of Technology in Lausanne (EPFL) in 2014. He currently serves as a research associate in the Circuits and Systems group at Imperial College London. His research interests include hardware/software codesign, CAD methodologies for hardware acceleration, and ultra-low power techniques, particularly in the areas of wireless networks and wearable devices.

Donatella Sciuto received her Laurea in Electronic Engineering from Politecnico di Milano and her PhD in Electrical and Computer Engineering from the University of Colorado, Boulder, and an MBA from Bocconi University. She is currently Vice Rector of the Politecnico di Milano and Full Professor in Computer Science and Engineering. Her main research interests cover the methodologies for the design of embedded systems and multicore systems, from the specification level down to the implementation of both the hardware and software components, including reconfigurable and adaptive systems. She has published over 300 scientific papers. She is a Fellow of IEEE and has served as President of the IEEE Council of Electronic Design Automation from 2011 to 2013. She is member of the IEEE Awards Committee. She is associate editor of Embedded Systems Letters and has been associate editor of different other journals in the field. She is in the executive committee of the conference IEEE/ACM Design Automation and Test in Europe, for which she has been Program Chair in 2006 and General Chair in 2008.She has been General Co-Chair of ESWEEK in 2009 and 2010. She has been Program Co-Chair for the IEEE/ACM Design Automation Conference in 2012 and 2013. She is in the main board of the European Design Automation Association.

Francesco Bruschi has a degree in Electrical Engineering and is PhD in Information Engineering. He currently works as assistant professor at the Politecnico di Milano, at the Dipartimento di Elettronica, Informazione e Bioingegneria. His current research interests include EDA methodologies, HW/SW architectures and digital systems for the teaching of computer science.