View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Infoscience - École polytechnique fédérale de Lausanne

A High-Level Synthesis Flow for the Implementation of Iterative Stencil Loop Algorithms on FPGA Devices

Alessandro Antonio Nacci,
Vincenzo Rana,
Francesco Bruschi,
Donatella Sciuto
Politecnico di Milano
Dipartimento di Elettronica e Informazione (DEI)
Milan, Italy
{nacci, rana, bruschi, sciuto}@elet.polimi.it

Ivan Beretta,
David Atienza
École Polytechnique Fédérale de Lausanne
Embedded Systems Laboratory (ESL)
Lausanne, Switzerland
{ivan.beretta, david.atienza}@epfl.ch

ABSTRACT

The automatic generation of hardware implementations for a given algorithm is generally a difficult task, especially when data dependencies span across multiple iterations such as in iterative stencil loops (ISLs). In this paper, we introduce an automatic design flow to extract parallelism from an ISL algorithm and perform a design space exploration to identify its best FPGA hardware implementation, in terms of both area and throughput. Experimental results show that the proposed methodology generates hardware designs whose performance is comparable to the one of manually-optimized solutions, and orders of magnitude higher than the implementations generated by commercial high-level synthesis tools.

Categories and Subject Descriptors

B.5.2 [**Design Aids**]: *Automatic synthesis*; F.1.2 [**Modes of Computation**]: Process Management—*Parallelism and concurrency*

General Terms

Design, Algorithms, Languages

Keywords

High Level Synthesis, Iterative stencil loops, Symbolic Execution, Performance and Area Estimation

1. INTRODUCTION

A large number of interesting algorithms for scientific computation and multidimensional signals processing, come in the form of iterative applications of a given transformation t. That is, starting from a signal f (a *frame*), the overall transformation T is defined as the repeated application of t:

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

Typically, the desired $T(f)$ is a fixed point of the single step transformation $t : t(T(f)) = T(f)$. 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 it has been analysed within the compiler community to find good implementations targeted to CPUs [6] and GPUs [7].

The design of dedicated hardware circuits for ISL algorithms, on the other hand, still presents unsolved challenges, and no automatic design flow can guarantee high performance implementations, mainly because of their complex data dependencies. The typical state-of-the-art approach for the implementation of generic iterative algorithms (such as ISLs) on FPGAs consists in employing two frame buffers [1] [2] [3], A and B, and a logic to compute t. 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) $\xrightarrow{t}$ f_1 (in B) $\xrightarrow{t}$ f_2 (in A), ...). The procedure continues until the desired number of transformations has been performed. This architecture shows a substantial shortcoming: the area and on-chip memory required by the transformation logic is proportional to the frame size, making it too costly in real-world conditions.

In this work, we propose a high level synthesis (HLS) methodology that specifically targets the class of ISL algorithms, as well as scientific and multimedia algorithms that show similar data dependencies between subsequent iterations. The proposed methodology stems from this observation: most of these algorithms feature a peculiar form of *spatial locality*, since the value of each element p at iteration $i+1$ (p_{i+1}) depends only on a small number of elements in the neighbourhood of p at iteration i (p_i). By exploiting this, the proposed synthesis methodology automatically generates custom hardware modules that work on a portion of the frame, and that output a subset of the intermediate results used by the subsequent iterations. Suppose we want to compute a single element p of the final resulting matrix, obtained after a number n of iterations, and let us call it p_n. The value of p_n depends on a set $P_{n-1} = \{p^1_{n-1}, ...p^m_{n-1}\}$ of elements computed at iteration $n-1$ and, by propagating these data dependency relations back to the starting input frame, we obtain the domain of the function that computes p_n. Since these algorithms are typically *translation invariant*, such function is uniquely determined by the number of levels we want to traverse, and we call it a *cone* of *depth* n, such as the one shown in Figure 1. We can generalise 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 characterised by P_n, which we call *window*.

It is intuitive (but we are showing it formally in Section 3.1) that it is possible to perform the desired processing by repeatedly applying a cone to portions of the input matrix. This approach leads to hardware implementations whose on-chip memory requirements

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. DAC '13, May 29 - June 07 2013, Austin, TX, USA.
Copyright 2013 ACM 978-1-4503-2071-9/13/05 ...$15.00.

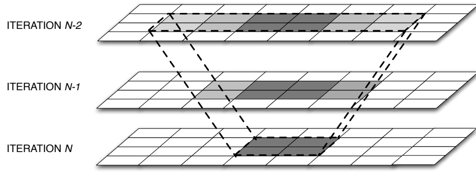

Figure 1: A cone of depth 2 and window size 4

are independent from the frame size. Let us call *cone architecture* a set of cones that perform the processing. The parameters that define a cone are the *cone window size* (let us consider only square windows, for the sake of illustration), the *cone depth*, and the *number of cones* simultaneously present in hardware. Finding the best cone architecture that satisfies a specified throughput constraint (such as a frame rate lower bound) is a challenging problem, as there is a large number of trade-offs at stake that make the space search complex. Moreover, the estimation of the area and the throughput of each cone architecture may require a very long synthesis time, which can reach the order of dozens of hours for realistic values of the window size and the cone depth.

In this paper, we face the previously described problems by presenting an automatic High Level Synthesis (HLS) flow that, given a processing algorithm, generates a set of hardware implementations that are Pareto-optimal with respect to cost and performance (measured as throughput). In particular, the flow: 1) takes a high level description (C language) of the algorithm as input, 2) systematically analyses data dependencies between elements of subsequent iterations, 3) generates synthesizable VHDL descriptions of all the *cones* that are best suited to custom fine-grained programmable devices (e.g., FPGAs), 4) efficiently explores the cone architecture solution space with a minimum number of synthesis runs.

2. ITERATIVE STENCIL LOOPS

Iterative stencil loops are designed to iteratively apply the same core operation (the *stencil*) on an *n*-dimensional matrix . The number of iterations can either be known in advance (as, for instance, in an iterative convolution filter [13], where the amount 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). Without loss of generality, we assume that the number of iterations is known *a priori*, and hence we formally define a generic ISL algorithm with the following pseudo-code:

Algorithm 1 Generic ISL Algorithm

```
for i in {1..N (number of iterations)} do
    for p in f_{i+1} do
        p=t_p(f_i) (t_p is the function that computes only the element p of the target frame)
    end for
end for
```

Many algorithms for multimedia processing follow the pattern presented in Algorithm 1, such as the ones presented in [13, 14, 15, 16, 18] and [3], as well as algorithms for scientific computation, such as convolution and the *Jacobi* iterative algorithm to solve linear eigenvalue problems [17].

If we analyse the typical structure of the elementary function t_p, we find that it shows two interesting properties: 1) the set of elements required to compute an element at the iteration $i+1$ is a small subset of the frame f_i produced at the *i*-th iteration, and these elements are close to element p that has to be computed, and 2) given two target elements that are separated by a translation, the corresponding dependency schemas have the same shape, but they are translated by the same distance as the target element. In this paper, we will refer to property 1) with the term "*domain narrowness*", and to property 2) with "*translational invariance*", which can be observed in all the algorithms we referenced in this section.

2.1 State-of-the-Art Implementations

From a computational point of view, ISL algorithms have traditionally been a challenging problem for the designers, mainly because of the complexity of their data dependencies. In the literature, the problem of designing efficient implementations for this class of algorithms has been addressed for both CPUs ([5], [6]) and GPGPUs ([7], [8]): on such architectures, the main problems that have been faced are the memory organization and the data transfers.

On the other hand, in the context of configurable devices (e.g., FPGAs), no complete design flow has been proposed so far, at best of our knowledge. In fact, the existing approaches for the hardware implementation of ISL algorithms either apply generic and ineffective optimizations, or impose very strict and limiting constraints. For instance, the work in [9] proposes a methodology to generate a hardware pipeline that spans across multiple iterations, but it is limited to only one floating-point operation per iteration, and no design space exploration is possible as the depth of the pipeline is uniquely determined. Conversely, generic HLS tools such as Xilinx Vivado [25] or Synopsys Synphony C Compiler [24] are able to handle any instance of ISL algorithms, but they perform a set of predefined and *general purpose* array and loop optimizations (unrolling, merging, flattering, pipelining, array partitioning, etc.) on the input algorithm, which is described in C. Since these frameworks do not take into account the peculiarities of the specific algorithm, the performance of the FPGA implementations they generate are generally unsatisfying for ISL algorithms, especially when compared to manually optimized implementations (as shown in Section 4).

Given the lack of support for the automatic generation of 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 [19] provides an efficient hardware approach for the Chambolle algorithm [18]. However, since these solutions are manually tailored for a specific algorithm, they lack of generality and reusability, and the effort required to adapt one of these solutions to a different problem (if possible) is generally not negligible, even if the algorithms are structurally similar.

The high level synthesis flow proposed in this paper fills the lack of automation for the implementation of ISL algorithms on FPGAs. The flow is based on the abstract methodology presented in [20], and it addresses issues that are different from the ones faced by the existing approaches targeting CPUs or GPGPUs. In fact, the hardware structure (e.g., number and kinds of cores) of the latter architectures is fixed and cannot be modified or adapted to the characteristics of the input algorithm. On FPGAs, on the other hand, the definition of an efficient hardware architecture is a crucial aspect, which must be specifically handled by the synthesis tool.

2.2 Design Challenges for ISLs

In this subsection, we provide an overview of the main design challenges that arise while implementing a generic ISL algorithm (like the one shown in Algorithm 1) on a custom hardware device, such as an FPGA. When performing two subsequent iterations, the intermediate results produced by the first one have to be stored (typically in an on-chip memory for performance reasons), since they will be the input to the second iteration. The straightforward way to implement this iterative structure on a custom hardware platform uses a temporary buffer to store the intermediate

data ([1] [2] [3]). However, if the dependencies of the particular algorithm are not taken into account, *it is necessary to completely compute an intermediate frame (f_i) before continuing with the following one (f_{i+1}).* In this case, if the on-chip memory is not large enough to hold a frame, it is necessary to transfer part of the intermediate results to the off-chip memory, and to get them back on-chip as soon as the next iteration starts. In this context, a memory/performance conflict arises. On the one hand, in order to keep high hardware performance, it is necessary to employ an on-chip memory large enough to hold all the intermediate results, but this typically requires several MBs of memory, which leads to expensive and power-consuming solutions. On the other hand, if the on-chip memory size is limited (only a few kBs for most of the devices used in the multimedia field), the performance is bound by the memory transfers that take place between the off-chip and the on-chip memories at each iteration.

The way we propose to handle this conflict exploits the structure of the dependencies in the algorithms that have to be implemented. In particular, by taking advantage of the *domain narrowness* (see Section 2), it is possible to design a new class of architectures, where a small portion of the input is processed *through all the iterations* by modules that we call *cones*. In this way, all the intermediate results can be stored in the on-chip memory and a transfer between on- and off-chip memories is only necessary when the logic starts to process a new portion of the initial frame. For instance, let us consider the case of the Chambolle algorithm [18] for optical flow estimation. At the best of our knowledge, an implementation of the algorithm that could sustain a real time frame rate was never proposed until [19], where the authors introduced an architecture that avoided computing a whole frame at a time, thus solving the memory/performance conflict. However, [19] proposes only a specific architecture, which was designed by hand and by studying the peculiarities of the algorithm, which requires a considerable effort. In this paper, we start from the architecture in [19] and, using the theoretical considerations shown in [20], we extend this approach by creating a high level synthesis flow that automatically analyzes the dependencies among iterations, and generates a set of Pareto-optimal implementations with respect to area and throughput.

3. THE PROPOSED HLS FLOW

The solutions space in which we seek a Pareto-optimal set of hardware implementations is composed by instantiations of the structural template we propose in Section 3.1, which allows to achieve very high performance even with modest on-chip memory requirements. The design flow that we use to generate and explore the design space is shown in Figure 2, and it consists of two main phases, which are described in the following sections: (1) analysis of the data dependency of the algorithm; (2) estimation of performance and area requirements for each architecture and design space exploration. As described in Section 3.2, the dependency analysis is performed with a novel combination of *symbolic execution* and register reuse. Then, in Section 3.3 we propose an original method to estimate the area usage of a generic cone architecture, starting from a high level representation of its structure, that provides a realistic estimation even with a low number of synthesis runs.

3.1 Architecture Template

If the input algorithm features *domain narrowness* (defined in Section 2), then it is possible to build a computational structure that is different from the straightforward one-whole-frame-at-a-time approach. To this end, in the proposed approach, data dependencies are extracted automatically by using the *symbolic execution* described in the following section, which makes it possible to express

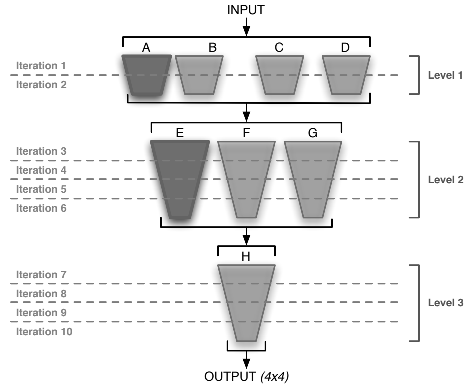

Figure 3: An example of the *cone*-based architectural template

the result of the $(i+m)$-th iteration as a function of (part of) the elements computed at the i-th iteration. As a consequence, given the data available from the i-th iteration, instead of trying to compute the whole f_{i+1}, we can focus on a subset of the matrix elements and directly compute the results of a generic m-th iteration (with $m \geq 1$), thus obtaining a subset of f_{i+m}. In accordance to what anticipated in the introduction, we refer to the core that performs such multi-iteration computation as a *cone* of *depth m*.

We define an *architectural template* by combining multiple levels of cones of different depths, which are able to compute the result of multiple iterations of the elementary transformation t. The proposed template (an instance of which is shown in Figure 3) works as follows: a small subset (window) of the input data (stored in the off-chip memory) is transferred to the on-chip memory to feed the cones of the first level of the architecture (A, B, C ad D in Figure 3). The output of each level is then used as input for the subsequent level, until all the necessary iterations are performed. The output of the last level (Level 3 in Figure 3) is finally sent back to the off-chip memory and the whole process starts over on a different window of the input data, until all the matrices has been computed.

The number and the depth of the cones in the actual architecture has to be tailored to the algorithm to be implemented, 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: (1) the size of the output window of each *cone*; (2) the number of levels in which the computation is divided or, equivalently, the number of iterations that are performed at once by each cone.

Figure 3 shows an instance of the template with an output window of 4×4 elements and 3 levels of computation: the first one involves 2 iterations, while the other two levels involve 4 iterations each. It is worth noting that, since the amount of data exchanged between two levels x and $x+1$ (the output of level x is the input of level $x+1$) only depends on the size of the output of level $x+1$ and on the number of iterations considered by the two levels of computation, the parameters previously introduced suffice to completely specify any architecture. 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 instance in Figure 3 is feasible if the available resources are sufficient to fit cones A and E because, in this case, the first level can be implemented by sequentially executing cone A four times (in order to cover B, C and D as well), and cone E four times (3 executions are required for level 2, and one for level 3). Many instances are generally feasible, and the same instance may be implemented in different ways by instantiating different numbers of cores of different depths, according to the resources availability.

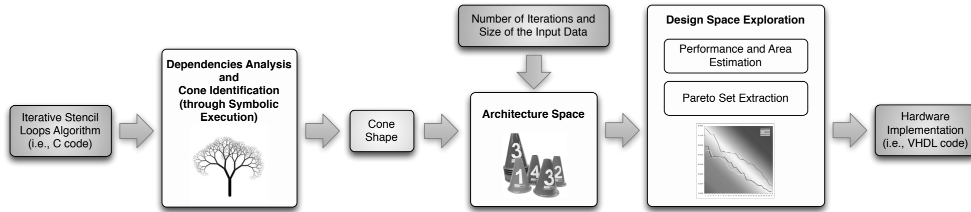

Figure 2: Schema of the proposed synthesis flow

As a consequence, multiple different trade-offs between area usage and achievable throughput (the more cones, the better) need to be evaluated, which is an aspect we address in Section 3.3.

3.2 Dependencies Analysis

In order 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 [19]) but, when different numbers of levels need to be evaluated during the design space exploration, an efficient and automatic way to determine the equations for all the $m = 1,...,N$ is required.

In the solution proposed here, the algorithm analysis is automated by running an optimized *symbolic execution* on a C description of the input algorithm. Symbolic execution is a well-known technique [21] that has traditionally been employed for testing purposes, which consists in executing the algorithm by propagating symbolic expressions rather than the actual values of the variables. Thus, after executing the algorithm from iteration i to $i+m$, the output 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.

The main problem that arises while performing symbolic execution in the general case is the exponential growth 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 2, which enables an efficient symbolic execution for the targeted class of algorithms. Firstly, it is not necessary to find an equation for *all* the elements of f_{i+m}: if *translation invariance* holds, the dependencies of the elements in the frame only differ by a translation, which allows tracking only one element in order to get the desired expressions for the whole f_{i+m}. Secondly, 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} 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 4. As we mentioned above, 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. This generates a slim VHDL code with a high degree of resource reuse, which can later handled by any synthesis tool for FPGAs that also performs area optimization.

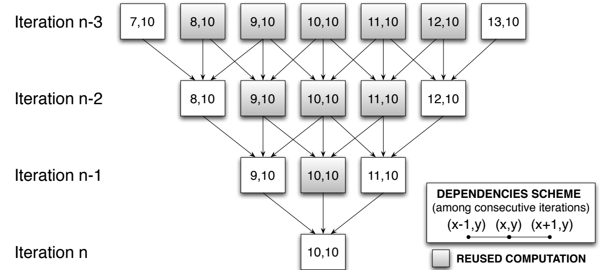

Figure 4: Example of the data-reuse technique

3.3 Performance and Area Estimation

In order to determine the Pareto-optimal architectures to implement a given algorithm, it is necessary to know the cost and the throughput of each architecture of the solution space. Unfortunately, a synthesis is generally needed to know the area required by a cone, which is directly related to the number of cones deployable on the target system, and thus to the performance of the whole architecture. An obvious way to determine area and performance would be to synthesize all the cones of every window size and depth but, for typical problem sizes, the synthesis may take days of CPU time, making a complete design space exploration unfeasible.

As a consequence, we hereby propose a novel methodology to quickly estimate the area requirements of a cone architecture. The proposed evaluation only requires a very small number (as low as two) of circuit syntheses, 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 in an analytical form 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, which is 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:

$$A_i^{est} = A_{i-1}^{est} + (Reg_i - Reg_{i-1}) \cdot Size_{reg} \cdot \alpha \quad (1)$$

Where A_i^{est} is the estimated area requirement for an architecture whose cones have an output window of size i. Reg_i is the number of registers in a cone with an output window of size i, and this quantity is already known when the VHDL description of the algorithm is generated and data reuse is enforced. $Size_{reg}$ represents the average size of a register on the target 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 or more initial syntheses (if a higher accuracy is needed, more initial synthesis need to be performed). The proposed estimation proves to be very effective in practice (see Section 4).

The estimation of the throughput follows the traditional approach, i.e., summing the delays of the operations included in each cone, and counting the number of cones that can run in parallel. This information is immediately available after the VHDL generation, when the kind and the number of operations are analyzed.

The precise estimation of the area of each cone makes it possible in turn to simply evaluate the latency (and thus the throughput) of any solution. Once the architectures space is completely characterized (thus, the area and throughput of each possible implementation has been estimated) the flow is finally able to extract the Pareto set by means of an exhaustive search that typically requires the evaluation of a few hundreds of solutions.

4. EXPERIMENTAL RESULTS

We applied the proposed flow on different case studies, of which we discuss the most significant two for the sake of illustration: an iterative gaussian filter [13] and the Chambolle algorithm [18]. The two algorithms are characterized by data dependencies of different complexities. The aim of the shown experiments is the validation of the proposed area estimation model, as well as the performance of the final architecture on two different ISL algortimhs.

4.1 Iterative Gaussian Filter (IGF)

The first case study considered is the *blur* effect, obtained by convolving an image f with a Gaussian kernel G. A common approach to implement gaussian convolutions with large kernels is to use an iterative gaussian filter (IGF) with a smaller kernel [11]. We exploit this property to formulate the filter as an iterative convolution of the frame f with a small kernel g.

The proposed flow performed the dependencies analysis, and then the area estimation. To verify the precision of our technique for the latter phase, we performed most of the syntheses, and compared them with our estimations: the results are presented in Figure 5 with respect to the output window size and to the number of iterations involved in the optimization. The maximum estimation error is 6.58%, and the average error is 2.93%, hence 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 6 shows the resulting Pareto curve, with respect to performance (in this case, the time to process a single fame) and area requirements (i.e., the number of slices on a FPGA), for the convolution of a 1024x768 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 know in advance, the synthesis tool uses all the available area to maximize the throughput, thus obtaining the results shown in Figure 7. This chart shows the degree of the throughput variation on a Xilinx Virtex-6 XC6VLX760 FPGA when the size of the output window is varied. It can be observed that the cores that lead to best performances are those whose depth is a divider of the number of overall iterations (in the example, 10 iteration are best performed with cores of depth 1, 2 and 5). The reason why cores of depth 3 and 4 achieve worse performance is that they are not dividers of 10, hence it is necessary to allocate an additional specific core (of depth 1 and 2, respectively) in order to implement the remaining iterations, thus making the exploitation of the available area suboptimal. Even by considering a single cone depth, the trend reported in Figure 7 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.

A comparison between our cone-based solutions and the ones presented in the literature show a significant speed-up when the amount of resources is comparable. For instance, [16] presents a

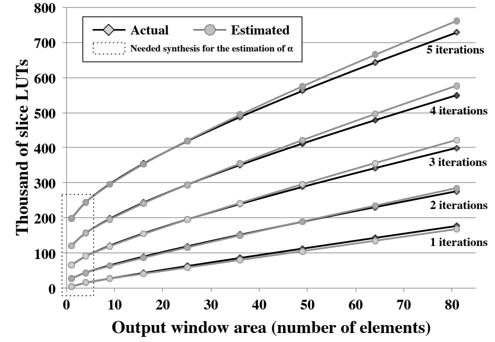

Figure 5: IGF area estimation

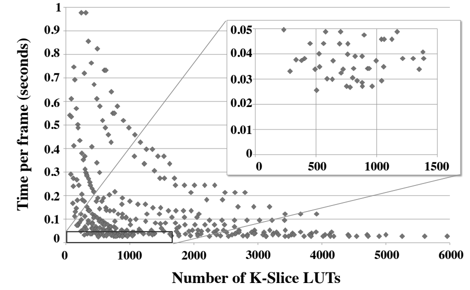

Figure 6: IGF Pareto curve (image size: 1024x768)

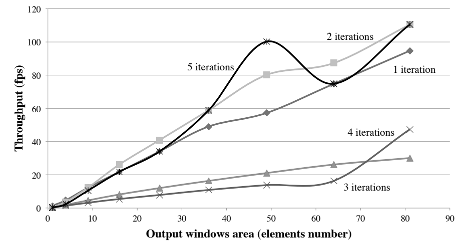

Figure 7: IGF throughput (image size: 1024x768)

20-iterations convolution with a 3x3 kernel working on a Xilinx Virtex-II Pro at 13.5fps with 1024x768 images, and at less than 5fps with Full-HD images, while our architecture achieves, on the same FPGA, up to 35fps on Full-HD images. With a modern FPGA such as a Virtex 6, our architecture reaches 110fps on 1024x768 images.

4.2 Chambolle Algorithm

Chambolle ([18], [19]) is an algorithm for total variation minimization, which is used in such fields as de-noising, zooming, and optical flow computation. As for the gaussian filter, we estimated the areas of each possible cone architecture for Chambolle, and we compared them to the actual synthesis results. Figure 8 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%.

Starting from the area estimation, we computed the Pareto curve, which is illustrated in Figure 9. When a specific FPGA is targeted, the behavior of the throughput is similar to the one 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 10). The performance of the cone-based architectures detected by our flow are competitive with respect to state-of-the-art implementations. For example, the architectures in [3], [22] and [23]), are unable to reach the real-time threshold (i.e., 30fps) even on small images because of their intrinsic absence of

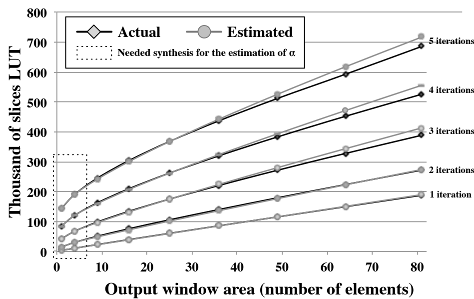

Figure 8: Chambolle area estimation

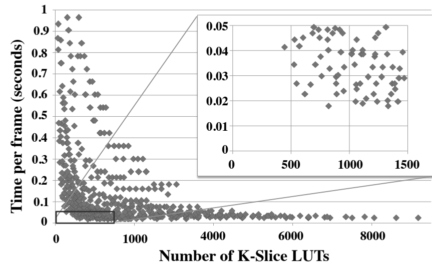

Figure 9: Chambolle Pareto curve (image size: 1024x768)

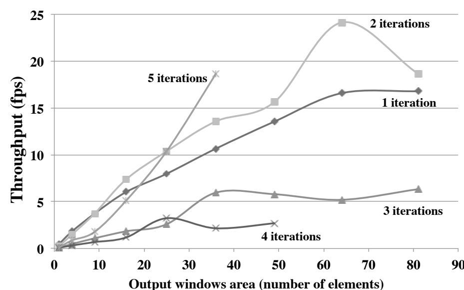

Figure 10: Chambolle throughput (image size: 1024x768)

parallelism. The architecture in [19], based on a similar locality exploitation, was designed by hand in several months of work, and it reaches 38fps on 1024x768 images and 99fps on 512x512 resolutions. With our flow we *automatically* obtained comparable results: 24fps on 1024x768 images and 72fps on 512x512 images.

4.3 Evaluation of Commercial HLS Tools

As a conclusion, we synthesized the two aforementioned case studies with two market-leading HLS tools: *Synphony C Complier* by Synopsis [24], and *Vivado HLS* by Xilinx [25]. Both the tools are able to perform a set of optimizations starting from a C code (even though Vivado HLS offers more choices than Synphony C Compiler), including loop unrolling, merging, flattening, pipelining, array partitioning, and many others. However, even for the simple iterative gaussian filter, the tools show their limitations in finding an efficient solution. For instance, by combining the possible loop manipulations and the array partitioning of Vivado HLS, the best implementation found by the tool has a throughput of only 0.14fps on a 1024x768 image. When loop merging is enabled, a solution cannot be found because of the data dependencies between subsequent iterations (which is peculiar of ISL algorithms). Conversely, when pipelining and loop flattening are employed, the execution cannot be completed because of memory shortage (an out-of-memory exception is generated even on a powerful Intel i7 with 16GB of RAM), thus showing poor scalability on ISL algorithms.

5. CONCLUDING REMARKS

In this paper, we considered the problematic implementation of Iterative Stencil Loop algorithms on custom hardware platform (such as FPGAs) from the high level synthesis perspective. Starting from the characterization of the family of algorithms, we proposed a hardware architecture template that overcomes the problem of storing the intermediate results by exploiting the dependencies between subsequent iterations. We also designed an automatic synthesis flow that produces a set of Pareto-optimal solutions with respect to area and throughput, starting from the C description of the input algorithm. In addition to the advantage of automatic synthesis, experimental results showed that the performance of the solutions found by the proposed flow are comparable to (and, in some cases, significantly better than) state-of-the-art manual implementations.

6. 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", *in Proc. of 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] C. Alias et al., “Automatic generation of FPGA-specific pipelined accelerators,” in Proc. of *ARC*, 2011, pp. 53–66.

[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, Department of Electrical & Electronic Engineering, 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,” *Journal of Mathematical Imaging and Vision*, vol. 20, pp. 89–97, 2004.

[19] A. Akin et al., “A high-performance parallel implementation of the Chambolle algorithm," in Proc. of *DATE*, 2011, pp.1,6.

[20] V. Rana et al., “Design Methods for Parallel Hardware Implementation of Multimedia Iterative Algorithms," *IEEE Design & Test of Computers*, 2012.

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

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

[23] A. Weishaupt et al., “Tracking and Structure from Motion,” Master’s thesis, EPFL, 2010.

[24] Synopsys, “Synphony C Compiler," 2012.

[25] Xilinx Inc., “Vivado Design Suite User Guide, High-Level Synthesis," UG902, 2012.