High-level Synthesis for FPGAs: Code optimisation strategies for real-time image processing

Chao Li, Yanjing Bi, Yannick Benezeth, D. Ginhac, Fan Yang

Journal of Real-Time Image Processing manuscript No.
(will be inserted by the editor)

Chao LI · Yanjing BI · Yannick BENEZETH ·

Dominique GINHAC · Fan YANG

High-level Synthesis for FPGAs: Code optimisation strategies for real-time image processing

Received: date / Revised: date

Abstract High-Level Synthesis (HLS) is a potential solution to increase the productivity of FPGA based real-time image processing development. It allows designers to reap the benefits of hardware implementation directly from the algorithm behaviors specified using C-like languages with high abstraction level. In order to close the performance gap between the manual and HLS based FPGA designs, various code optimization forms are made available in today's HLS tools. This paper proposes a HLS source code and directive manipulation strategy for real-time image processing by taking into account the applying order of different optimization forms. Experiment results demonstrate that our

Chao LI

[1] Stat Key Laboratory of Acoustics, Institute of Acoustics, Chinese Academy of Sciences, Beijing 100190, China

[2] LE2I FRE2005 CNRS, Arts et Métiers, Univ. Bourgogne Franche-Comté, F-21000 Dijon, France

E-mail: chao.li.1986@ieee.org

Yanjing BI (Corresponding author)

Laboratory of CPTC, Univ. Bourgogne Franche-Comté, F-21000 Dijon, France, E-mail: yanjing.bi@hotmail.com

Yannick BENEZETH, Dominique GINHAC, Fan YANG

LE2I FRE2005 CNRS, Arts et Métiers, Univ. Bourgogne Franche-Comté, F-21000 Dijon, France

E-mail: yannick.benezeth@u-bourgogne.fr, dginhac@u-bourgogne.fr, fanyang@u-bourgogne.fr

approach can improve more effectively the test implementations comparing to the other optimization strategies.

1 Introduction

Real-time image processing is an increasingly widely used computer vision technique in various fields. A signal or image processing system is considered as "real-time" if it is a reactive system able to absorb input data and outputs results within a latency lower than the time between two successive frames acquired by the image sensor. From the hardware point of view, the challenge of real-time image processing is to find the optimal platform satisfying the requirements of the image processing application among a large space of potential solutions. Its solution exploration therefore usually revolves around how to combine the software implementation with the hardware platform [1–5].

FPGA is one of the frequently used computing device for real-time image processing for its advantages of high efficiency-cost ratio. However, the devices of this type necessitate a configuration in the Register-Transfer Level (RTL) with low abstraction level, potentially reducing the research and development productivity. For this issue, an automatical C-to-RTL synthesis technique, known as High-Level Synthesis (HLS), is developed and has made great progress over the past twenty years [6–10]. Recently, some robust and mature HLS frameworks have been made available to engineers, i.e. Vivado_HLS of Xilinx [11] and Catapult C Synthesis Work Flow [12]. These convenient tools allow one to specify targeted hardware behaviour in high abstract levels rather than RTL, and then create the Hardware Description Language (HDL) specification of desired FPGA implementations from its software prototype. This approach can greatly accelerate the developments by freeing the designers from the boring work of hardware implementing [7]. For example, the case of Wakabayashi Kazutoshi [13] shows that a 1M-gate design usually requires about 300K lines RTL code, which cannot be easily manually handled, while the code density can be easily reduced by $7 - 10\times$ when moved to high-level specification in C-like languages, resulting in a much reduced design complexity. Villarreal et al. [14] present a Riverside Optimizing Compiler for Configurable Circuits that achieves considerable improvements in terms of code intensity and programming time over hand-written VHDL in evaluation experiments.

However, although HLS tools can offer high quality designs for small kernels, many studies demonstrate a significant performance gap between HLS-based and manual design for complex applications [15–17]. In the cases of Rupnow et al. [17] and Liang et al. [15], the performance difference between the two is up to 40× for a high-definition stereo matching implementation. Thus, various academics proposed different solutions to this issue. For example, Rodirgues et al. [18] proposes an execution technique to speed-up the overall execution of successive and data-dependent tasks on a reconfigurable architecture. Ziegler et al. [19] develop a set of compiler analyses that can help to automatically map a sequential and un-annotated C program into a pipelined implementation targeted for an FPGA with multiple external memories. Meanwhile, Cong et al. [20] propose a new communication synthesis approach targeting systems with sequential communication media and Li et al. [21] develop a customized affine-ISS (Index-Set Splitting) optimization algorithm that aims at reducing the Initiation Interval of pipelined inner loops to reduce the program latency.

Recently, Cong et al.[9] indicate that quality of the RTLs generated from HLS are influenced by the high-level description of language. Meanwhile, Huang et al. [22] minutely studied the effects of different compiler optimizations on HLS-generated hardware. It demonstrates that the following two factors can help for improving the quality of generated RTL: the optimizing itself and its applying ordering. In their work, six optimizing methods are successively applied into the *jpeg* benchmark for all 6! (= 720) orderings, and it is found that the fastest implementation is nearly 28% more efficient than the slowest one. This finding offers new benefits to further improve the HLS tools and its design flows.

Up to our knowledge, there is not yet a mature source-to-source compilation tool specially designed for HLS, the designers therefore have to borrow from earlier research achievements on optimizing compiler back-end for DSP, VLIW (Very Long Instruction World) or multiple-core processors. This temporary solution partly free the users from the code re-writing, but the transformation strategies used in these unspecified compilers does not solve the problems in nature. Our work focus on the rapid development framework of HLS-based FPGA designs for image processing. Basing on the findings of Huang et al. [22], this paper proposes a novel code and directive manipulation strategy for HLS, which can be used in automatical C-to-C compilers. Unlike the other similar researches, we take into

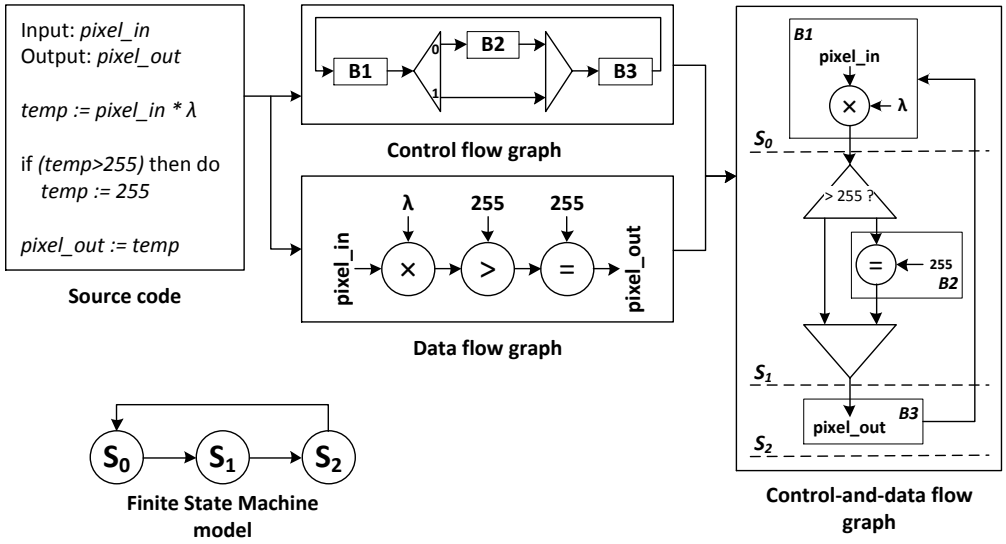


Fig. 1 Control-and-datapath extraction.

account the effects of applying ordering of optimizations. For this purpose, the synthesis constraints of the world-leading HLS tool, Vivado_HLS, and its potential optimization opportunities are first carefully studied. Next, we perform a special compilation procedure, which sequentially applies different optimization forms into the design. The proposed approach can reduce the resource consumption of the final implementation due to the control flow by simplifying the function or loop hierarchies of the designs, and provide the schedule process more pipelining opportunities in the instruction level according to the symbolic expression manipulation.

In the experiments, we evaluate the proposed approach by inserting it into the classical HLS based image processing framework with four benchmarks: 3×3 Filter, Matrix Production, Image Segmentation and Stereo Matching. The evaluation results demonstrated that our approach can effectively improve the efficiency of source code. In additional, we compare as well the HLS design flows improved by our approach with two other ones. The comparative results show that it substantially improves the efficiency of the final implementation.

The remainder of this paper is organized as follows: Section 2 reviews the synthesis process of HLS. Section 3 presents the proposed code optimization strategy. Section 4 analyzes the evaluation experiments. Finally, a conclusion is given in Section 5.

2 Description of High-Level Synthesis

High-Level Synthesis (HLS) is also known as C Synthesis or Electronic System Level synthesis (ESL synthesis). It allows hardware designers to efficiently build and verify their targeted hardware implementations by giving them better control over optimization of their design architecture and allowing one to describe the design in a higher abstract level. Generally, its overall process consists of control and datapath extraction and RTL generation.

HLS formally represents the source code by using a Control and Datapath Flow Graph (CDFG), which is one of the most widely accepted modeling paradigms. A CDFG is a directed graph in which every node and arcs refer to a basic blocks B and control flows respectively [23]. A basic block is defined as a straight-line sequence of statements without branches. As shown at the top of Fig. (1), in this cycle, the data and control flows are firstly represented by using a Data Flow Graph (DFG) and a Control Flow Graph (CFG) respectively. In DFG, every node refers to an operation while arcs the data assignments. Next, the two graphs are fused into together as the desired CDFG by assigning the operations in DFG into the basic blocks of CFG. In this way, the order of execution of the process elements can be determined as well as its architecture.

The extracted CDFG can be represented by using a Finite State Machine (FSM) with datapath as shown at the lower left of Fig. (1). This model is one of the most popular methods for digital system specification at RTL. The generated FSM divides the elements of CDFG into a set of states S and control steps for synthesis. Meanwhile, it should be noted that the overall process of this transformation can be automated or user-driven.

Now we can start to generate the desired RTL. To do this, three interdependent tasks are needed, including scheduling, allocation and binding. Scheduling task schedules the operations represented in CDFG into cycles. More precisely, for every operation, its operands must be read from either storage or functional unit components, and the results must be assigned to its destinations (another operation, storage or functional unit). These operations need to be scheduled within a single clock or over several cycles one by one.

Allocation and binding processes come after scheduling. In allocation, the type and quantity of hardware resources, i.e. functional units, storage or connectivity components, are determined first

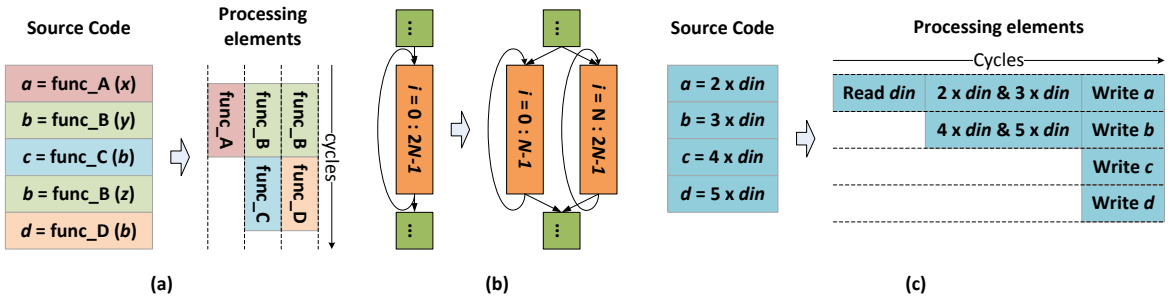


Fig. 2 Forms of parallelism: (a) Function-Level Parallelism, (b) Loop-Level Parallelism and (c) Instruction-Level Manipulation.

depending on the scheduled CDFG. Next, the desired hardware resources are selected from the given Intellectual Property (IP) library that contains all the necessary information for every component, such as area, delay, power and metrics to be used by other synthesis tasks as well. Finally, binding is done to generate the RTL with the following tasks:

a) Functional binding: bind all the arithmetic or logic operations to the functional units allocated from the IP library;
b) Storage binding: each variable that carries values across cycles must be bound to a storage unit like register;
c) Connectivity binding: bind data transfers to the connective units, such as assignments, buses etc.. In additional, if the interconnects are shared by multiple data transfers, a multiplexer will be needed between the sources and destinations.

In parallel computing, variant parallelism forms, i.e. Instruction-Level Parallelism, Data-Level Parallelism and Loop-Level Parallelism, etc [24,25], are usually made in order to achieve high efficiency implementations. For HLS, its source code can be divided into function level, loop level and instruction level, which provide different opportunities to improve the throughout and efficiency performance of RTL implementations:

1. Function-Level Parallelism (FLP). The sub functions of the source code for HLS are specified as sub entities in RTL and interconnected according to the extracted control behaviour. Despite of the sequential nature of C, these entities can be scheduled in parallel as shown in Fig.2-(a). Furthermore,

the generated sub entities can be reused to reduce the area consummation if there is no scheduling conflict.

2. Loop-Level Parallelism (LLP). LLP is one of the popular parallelization methods in the scientific computing community. In this form of parallelization, independent iterations of the same loop are executed in parallel (see Fig.2-(b)). Although the acceleration ratio is constrained by memory access conflicts and data dependency existing at the iterations in the same loop, this method offer often significant speedup gains to the designs with DOALL loops[1].
3. Instruction-Level Manipulation (ILM). This form of parallelization consists of Instruction-Level Parallelism (ILP) and Binding Control (BC). The motivation of ILP is to improve the performance of the designs by simultaneously executing multiple independent elements, even in an order different from the programm (out-of-order execution). However, its DOP[2] is constrained by the available IP core numbers. In order to achieve more potential efficiency improvement, the operations can be bound to high efficiency operators. In the example of Fig.2-(c), 2-level pipeline multipliers are used to execute 4 multiplications in parallel with a limitation of 2 cores.

3 The proposed code and directives manipulation strategy

This section describes the proposed code and directive manipulation strategy for HLS. Depending on the characteristics of HLS presented in Section 2, we select four optimization forms from the frequently used candidates in parallel computing field, including function inline, loop fusion, symbolic expression manipulation and loop unwinding. But unlike the existing code optimization methods presented in literature, as shown in Fig. (3), they are applied on the input source code in a proper order in order to enable each optimization method to make a maximum effectiveness.

[1] DOALL loop: the loops with independent iterations

[2] Degree of parallelism of an operation: a variable that indicates how many times an operation can be or are being simultaneously executed in maximum for an implementation.

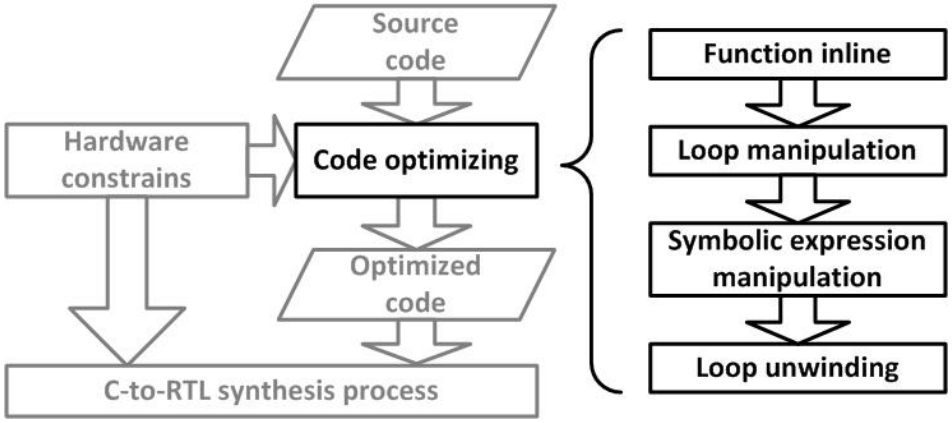


Fig. 3 The proposed code optimization process.

3.1 Function inline

Historically, well using custom function is invariably encouraged as one of the most basic necessary abilities for programmers. However, this coding habits seriously disrupt the optimizations in the instruction level in HLS. Despite of the ability to parallelize the sub functions, HLS has to separately process each function in order to map them to the sub entities, which are interconnected via assignments interfaces. The shortcoming of this approach is that all the operations in deeper-level entities can't be executed until the higher-level entities finish the jobs even if some of the former's operations could be parallelized with the latter's.

In order to offer more optimization opportunities in loop and instruction level, we first flatten the hierarchical algorithm description into a single level. Fig. (4) shows an instance to compare the affects from the code sources before and after function inline. In *Code 1*, *func_A* and *func_B* are mapped into two separate entities, so all the operations of *func_B* are activated after the termination of *func_A*. Meanwhile, function inline assembles all the operations into a single RTL entity, therefore *opt_3* could be scheduled at the beginning of the clock sequence with other operations, which put the execution of the *opt_1* in terminal 1 cycle ahead of schedule. Thus, inlining functions can simplify the hierarchical architecture of designs and enable HLS to effect optimizations on the independent objects isolated by sub function customizations, such as loop fusion and instruction pipeline.

Supposing subfunctions of the top level have the same interval latency, for a top behavior consisted of M_{func} dependent subfunctions with a latency of $L_{interval}$, its latency acceleration ratio, R_{fi}, can

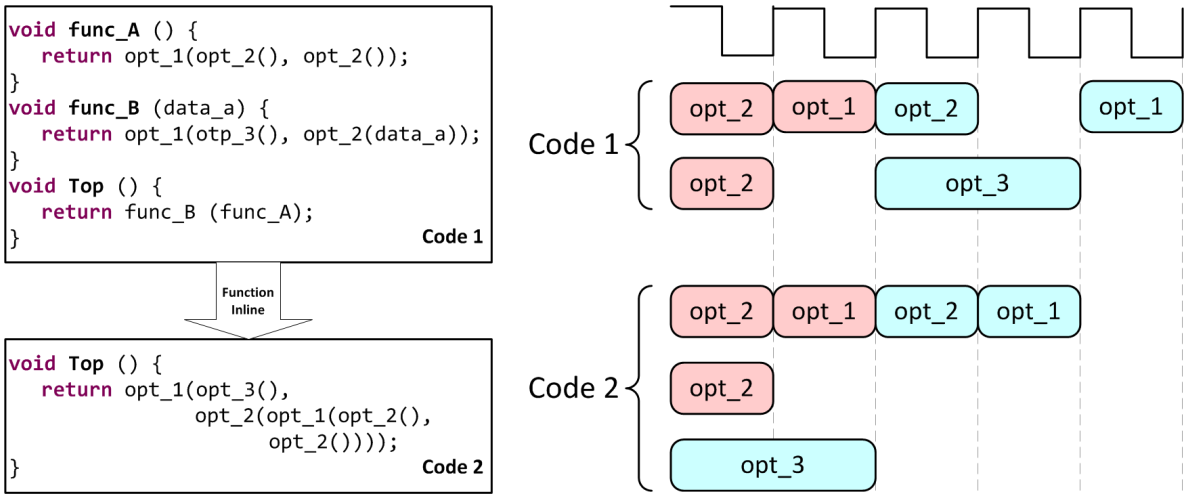


Fig. 4 Comparison between the code sources before and after Function Inline.

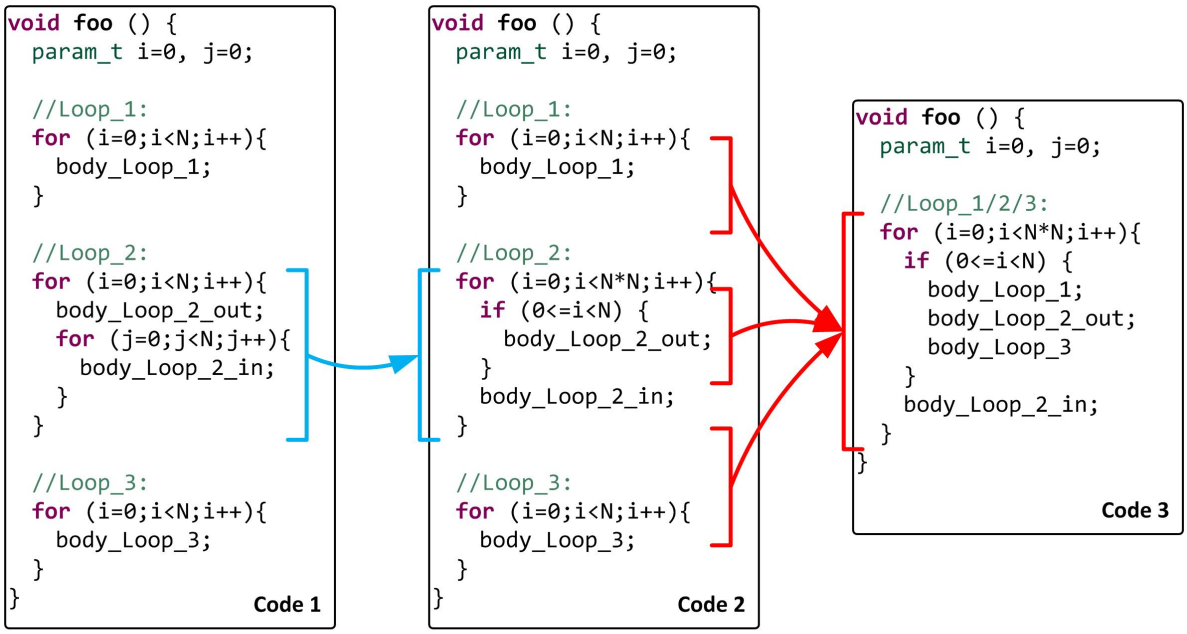
be estimated as follow:

$$R_{fi} = \frac{\sum_{i=1}^{M_{func}} L_i + (M_{func} - 1) \times L_{interval} + L_{other}}{L_{fi}} \tag{1}$$

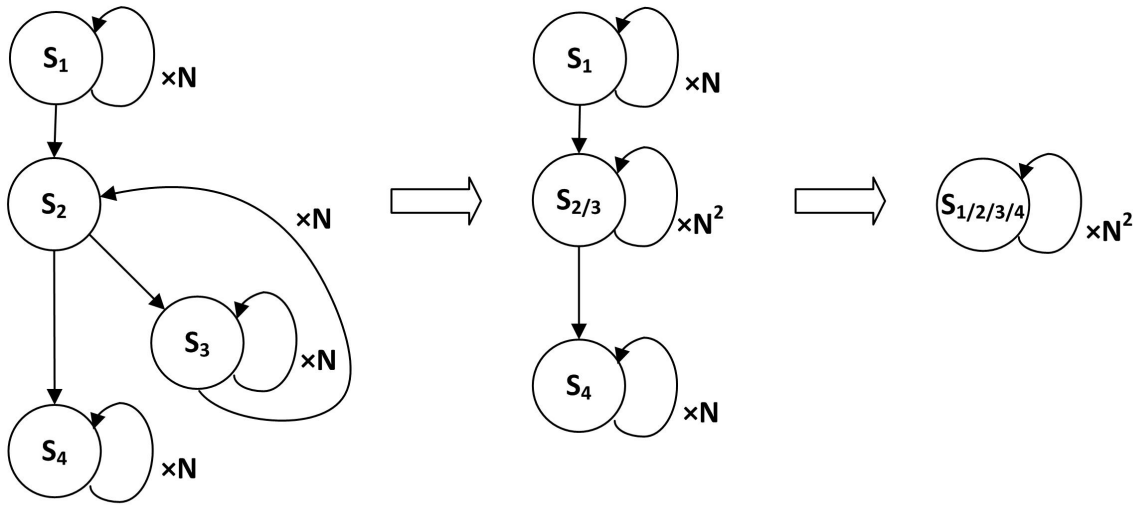
where L_i is the latency of the i-th subfunction, $L_{interval}$ is the interval between the subfunctions, L_{other} is the latency due to other instruction level operations in top function, and L_{fi} is the latency of the top function after function inline. Since different operations scheduling constraints may result in different execution cost, it is hard to give a universal formulation to compute L_i or L_{fi}. In practical applications, the latency cost of a single subfunction/function can be estimated with the help of HLS tools, Vivado_HLS [26] for instance, when the scheduling constraints are defined.

3.2 Loop manipulation

Function Inline puts all the loops in the same function hierarchy. This allows a more comprehensive loop-level optimization. However, HLS processes loop-bodies as separate states during the control-and-datapath extraction, so successive loops have to be sequentially executed rather than in a parallel way, even if they are independent with each other. Nevertheless, separate states prevent the designs from operation pipelining or data sharing. In order to potentially greater optimize the loop-body logic, we manipulate the source code in loop-level using loop flattening and loop merging (see Fig. (5)). Given that only the same-hierarchy loops can be merged together, we first flattens nested loops into



(a) Code transformation of Loop Manipulation.



(b) FSM behaviors for Loop Manipulations.

Fig. 5 Loop manipulation.

simple loops respectively. Next, all the simple loops are merged into a single one. In this cycle, the largest loop bound of all the original loops is selected as the bound of the new generated loop, and their bodies are controlled and optionally executed according to *if* statements. By this means separate loop-bodies are fused into a single state. This simplifies the hierarchical architecture of the design and offers nice opportunities to potentially greater optimize the loop-body logic. In our studies, the total

Table 1 Symbolic expression manipulation strategies.

Strategies	Original expressions	Equivalent expressions
Folding	$1 + 2a + 3 - 4a$	$4 - 2a$
Division	$(2 \times a)/(4 \times b)$	a$/(2 \times b)$
Short-circuit evaluation	a && 0 && b	0
Normalization	if $(1 + a < b - 2)$	if $(a - b < -3)$
Segmentation	return $a \times b \times c \times d$	$tmp1 = a \times b$
		$tmp2 = c \times d$
		return $tmp1 \times tmp2$

state transition number is defined to estimate the complexity of the FSM to be implemented. As shown in Fig. (5), *Code 3* requires only N^2 state transitions while $2N + N^2$ for *Code 1*.

3.3 Symbolic expression manipulation

In the third step, we manipulate symbolic expressions. Table (1) lists the strategies applied in the symbolic manipulation for HLS. Folding, Division, Short-circuit evaluation and Normalization reduce designs' hardware or time consumption by simplifying computations mathematically. Furthermore, Normalization is also an optimization aimed to the comparison operations. It re-performs the irregularly expression by locating the unknown variables and the constant on the different sides of the comparison operator, and then simplifies them by other strategies respectively. Obviously, the equivalent expression consumes less hardware and cycles than the former.

Segmentation is a transformation that enhances HLS's detection ability in terms of Instruction-Level Parallelism. For polymerization, the existing HLS tools can only pipeline different terms while scheduling the operations in a single long term in sequence. This is because "term" is the minimum parallelizable elements in HLS procedure. Thus, we re-perform the long terms that have more than 3 multiplication/division operators via Segmentation to make these independent operations detectable to HLS.

3.4 Loop unwinding

Unwinding the loops in the code can multiply the running speed of implementations by parallelizing the independent iterations in the same loop. Let the unrolling times of this transformation be n, then the theoretical value of the acceleration ratio R should have been $2^n\times$. However, sometimes the efficiency improvement achieved by this optimization is lower than this expected value. This is because loop iterations share always a single top interface and the reading/writing operation of the $(i+1)^{th}$ iteration have to be delayed certain cycles relative to the i^{th} iteration. The value of R can be formulated as follows:

$$R(n) = \frac{L}{L_{lu}(n)} \tag{2}$$

where

$$L_{lu}(n) = 2^{-n} \times L + (2^n - 1) \times D_{ac} \tag{3}$$

$$D_{ac} = max\{D_{rd}, D_{wr}\} \tag{4}$$

In Eqs.2, 3 and 4, L and L_{lu} are the loop latencies before and after unwinding, and D_{ac}, D_{rd} and D_{wr} are the delays due to the access conflicts, reading operations and writing operations. Eq.4 indicates that the maximum of D_{rd} and D_{wr} is selected as D_{ac}, this is because both reading and writing produce access conflicts, but what ever the cause, loop unwinding delays the entire iteration. Thus, D_{ac} only needs to exceed the maximum value between D_{rd} and D_{wr}.

The area of the implementation optimized by loop unwinding, A_{lu}, can be estimated as follow:

$$A_{lu} = \sum_{k=1}^{K}(DOP_k \times a_k) + A_{control} \tag{5}$$

where K is the number of the operation types, DOP_k is the degree of parallelism of the k^{th} operation, a_k is the area of the k^{th} component and $A_{control}$ is the area of control circuit. It should be noted that hardware resources constrain the maximum degree of parallelism of the implementations in practice. According to Eq.5, this constraint can be formulated as:

$$\begin{aligned} & A_{lu} < A \\ \Rightarrow \quad & \sum_{k=1}^{K}(DOP_k(n) \times a_k) < A - A_{control} \end{aligned} \tag{6}$$

where A_{lu} and A are the area of the implementation optimized by loop unwinding and the target device.

4 Experiments and evaluations

In this section, we evaluate the proposed code optimization strategy with multiple different benchmarks using AutoESL, which has been acquired by Xilinx and is now part of Vivado_HLS. These benchmarks are tested using 8×8 arrays with *float*, *int* and *short* data formats. A 2-ports 128-bits memory interface is set as the I/O protocol of the top behaviour. In order to obtain an unbiased conclusion, the versions of the proposed method of these benchmarks are further compared with those optimized using the Polyhedral Framework and Vivado_HLS design suite only. All the experiments are made on the *xc7a200tfbg676-2* of Xilinx.

4.1 Performance improvement evaluation

For the first step of the evaluation, four different algorithms are implemented and functionally verified by using C language, including 3×3 filter for RGB images, matrix product (MatPro), Image Segmentation using Sobel operator (ImSeg) and Stereo Matching using sum of squared difference (StMatch). Next, we transform the source code via our code optimization strategy. During this transformation procedure, the code generated within each step is synthesised with AutoESL to evaluate the performance improvement related to the previous phase.

Table (2) describes the clock cycle and resource consumptions of the four implementations. Compared to the original versions which are transformed from their C versions through HLS without any optimizations, the targeted RTL implementations are greatly improved in terms of cycle consumption within the hardware constraints of the evaluation board. This demonstrates that with the proposed approach, HLS tools can effectively use additional FPGA resources.

As discussed in Section 3, the performance improvement is the integrated result of an optimization procedure with multiple steps. First, the control architectures of the prototypes are well simplified according to function inline and loop manipulation. This allows to avoid the unnecessary cycle and resource consumptions due to the control flow. For example, float_32 FI versions of ImSeg achieves a

1.05× speedup relative to its original version but consumes only 90.12% hardware resources in average, while the float_32 LM versions of MatPro consumes a hardware resource of 97.7% in average for a 1.026× speedup relative to its function inline version. Additionally, simplifying the control architecture can also greatly accelerate the design by reducing the state transition numbers and creating more parallel computation opportunities. This can be proved by the LM versions of 3×3 Filter and ImSeg, which respectively achieve a 2.85× and a 2.83× speedups in the average of the three different data formats.

Next, according to our tests, symbol expression manipulation achieves a 1.447× speedup at most in all the implementations versions. This demonstrates that it can effectively improve the computing efficiency of the design. However, it is also found that this optimization is not effective in some cases. This is mainly caused by two reasons: (a) the input code has been in the simplest form (i.e. the SEM versions of MatPro and StMatch), and (b) the latencies of the operation components are too tiny to effect the performance improvement. The second reason can be observed according to the comparison between the float and int SEM versions of 3×3 Filter and ImSeg. Since the latency of 32-bit floating point number adders is 4 cycles, while the 32-bit integer number adders do not consume any cycles in our experiments, so simplifying these two implementations' computation can only accelerate the execution speed in the case of float_32 versions, but does not effect for the int_32 or int_16 versions, even if the symbolic expressions of the new generated code are simpler than their previous versions.

Finally, as discussed in Section 3.4, the execution speed of all the four implementations are multiplied by unwinding the loops. According to our test, the loop unwinding optimized code respectively achieves 19.84×, 143.34×, 29.94× and 90.39× speedups in average for the four implementations. However, it should be noted that the resource consumption is multiplied as well after this transformation. This results that the surface of the target implementation probably exceeds the hardware constraints. For example, the efficiency of the int_32 LU version of 3×3 Filter could have reached to 50 cycles if its loops are completely unrolled, but due to the limitation of the DSP number (768 required v.s. 740 in maximum), they can be only partly parallelized in order to make the surface of the desired implementation available to the target board.

Table 2 Performance and resource consumption evaluation using the proposed optimization applying order: FI (Function Inline), LM (Loop Manipulation), SEM (Symbolic Expression Manipulation) and LU (Loop Unwinding).

Implementations	Procedures	32-bit floating point numbers (float_32)					32-bit signed integer numbers (int_32)					16-bit signed integer numbers (int_16)				
		Cycles	BRAM	DSP	FF	LUT	Cycles	BRAM	DSP	FF	LUT	Cycles	BRAM	DSP	FF	LUT
3 × 3 Filter	Original	10611	0	2	1237	1241	2355	0	12	912	1446	1779	0	6	497	882
	FI	10611	0	2	1237	1241	2355	0	12	912	1446	1779	0	6	497	882
	LM	3521	0	6	2718	3284	961	0	12	1111	1382	577	0	3	376	644
	SEM	2433	0	6	3582	3284	961	0	12	1111	1382	577	0	3	376	644
	LU	74	0	98	28540	34409	71	0	384	24724	39801	44	0	192	18394	35920
MatPro	Original	5777	0	5	534	387	4241	0	4	118	108	1681	0	1	56	55
	FI	5777	0	5	534	387	4241	0	4	118	108	1681	0	1	56	55
	LM	5633	0	5	523	368	4097	0	4	107	89	1537	0	1	41	36
	SEM	5633	0	5	523	368	4097	0	4	107	89	1537	0	1	41	36
	LU	328	0	7	1379	811	15	0	8	1021	1784	11	0	5	355	450
ImSeg	Original	16644	3	12	4900	4786	10844	3	12	3981	4175	9604	3	3	3356	3677
	FI	15876	3	10	4267	4295	10500	3	28	4065	4771	9220	3	7	3330	3933
	LM	6273	0	30	7945	8033	3841	0	28	9127	10046	3201	0	7	8357	9208
	SEM	5633	0	42	9463	9635	3841	0	28	9127	10046	3201	0	7	8357	9208
	LU	131	0	213	66540	86889	301	0	448	35943	41117	94	0	448	72486	81162
StMatch	Original	37713	0	15	5350	5247	20561	0	36	4065	4071					
	FI	36177	0	10	4500	4810	20561	0	36	4065	4071					
	LM	34705	0	10	4517	4813	20497	0	36	3846	3849					
	SEM	34705	0	10	4517	4813	20497	0	36	3846	3849					
	LU	325	0	197	83748	105937	277	0	516	48844	53820					

4.2 Comparison experiment

In this subsection, we compare the proposed approach with other two functionally similar design flows using the source code of the four benchmarks mentioned in the preceding subsection (see Fig.6). We base the first reference design flow on two improved conventional source-to-source C/C++ compilers: an improved PoCC polyhedral framework [27,28,21] and the Generic Compiler Suite (GeCoS) [29–31] (defined as PolyComp), while the other one on the Vivado_HLS Design Suite [32] (defined as Vivado_HLS). In order to obtain an unbiased conclusion, all the source codes are synthesized using AutoESL and their data formats are normalized to 32-bit integer numbers. Table (3) lists the

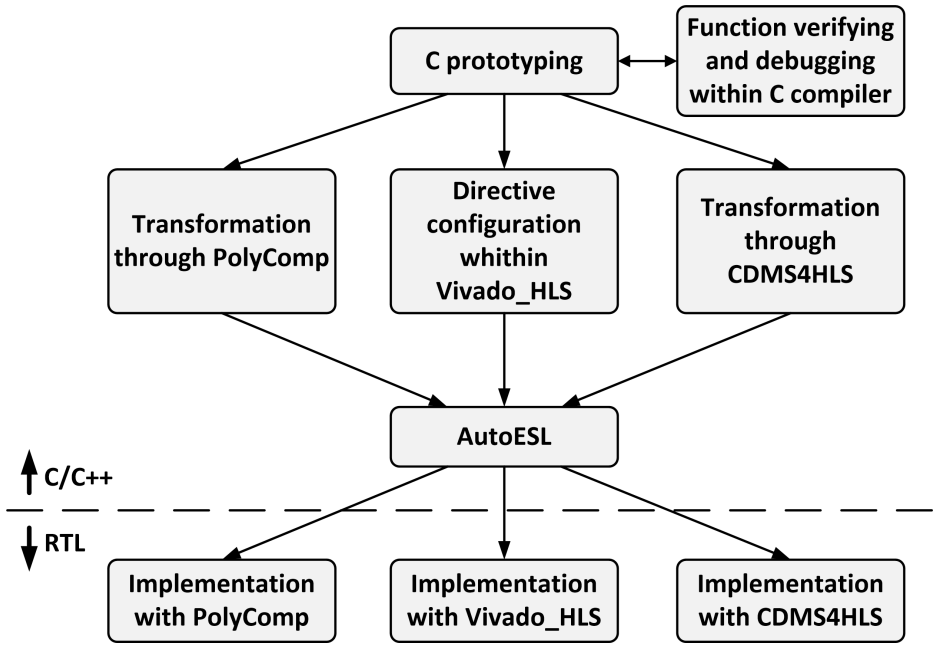


Fig. 6 Implementing flow with different code optimization methods, including PolyComp, manual directive configuration within Vivado_HLS and the proposed code optimization strategy.

Table 3 Optimization forms of reference design flows (PT: Polyhedral Transformation, FI: Function Inline, LF: Loop Flatten, LU: Loop Unroll).

Implementations	PolyComp	Vivado_HLS
3×3 Filter	FI, PT	FI, LF, LU
MatPro	PT	LU
ImgSeg	FI, PT	FI, LU
StMatch	FI, PT	FI, LU

optimizations made by reference design flows. In additional, the instructions are pipelined by default during the synthesis process by the AutoESL to all the three flows. Considering that PolyComp does not have the ability of I/O interface manipulation, we set the I/O protocol of the target implementations as the default of the HLS tool used.

The latency speedups of the three design flows with different algorithms are compared in Fig.7. The int_32 original versions of the related algorithms are set as the reference standard. These three approaches respectively achieve an average of 19.01×, 22.19× and 106.54× speedups. This demonstrates that out method can gain more performance improvements in terms of latency consumption. Compared

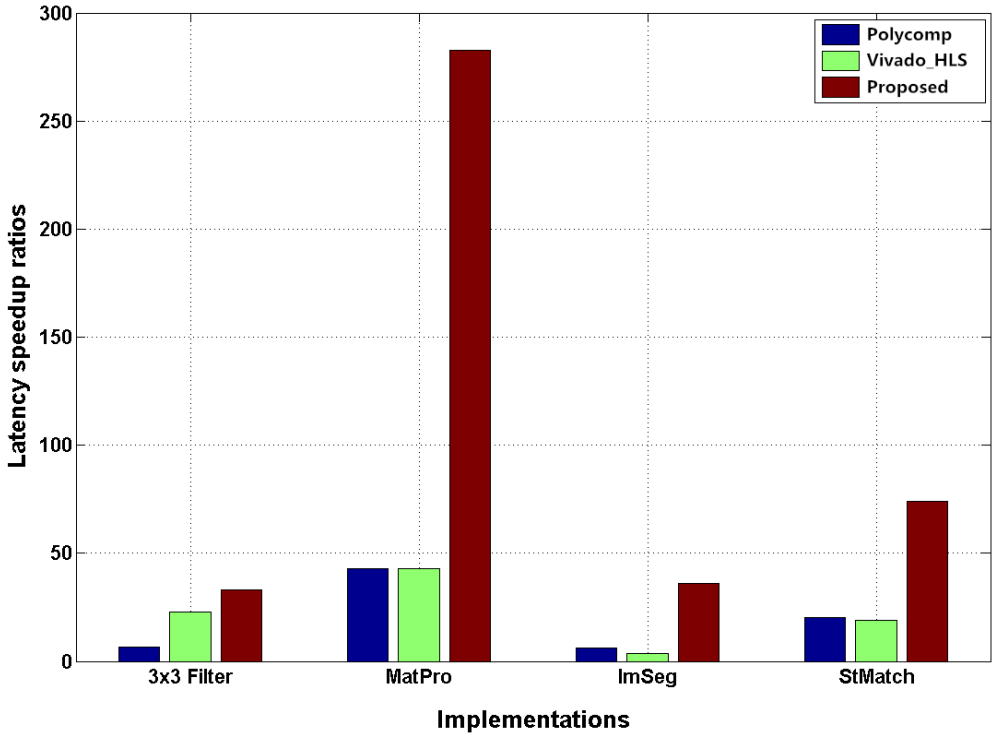


Fig. 7 Latency speedup comparison.

with the other designs, the proposed approach has the ability to manipulate the source code in a lower instructions level, which provide more optimization opportunities to HLS tools. Furthermore, our method can effectively reduce the transition number of the FSM behaviours of the target implementations. For example, the transition number of the MatPro optimized by it is only half as many as PolyComp and Vivado_HLS respectively. Therefore, the method of this paper and Vivado_HLS may speedup the implementations more effectively than PolyComp. However, it is noted that the hardware resource consumptions of the optimized implementations using the reference design frameworks are lower than the proposed approach. Theoretically, the resource consumptions increase with the degree of parallelism of operations. Our method can parallelize more efficiently the implementation, so it is bound to result in more consumption. In the experiments of this paper, a hardware device large enough is selected, allowing completely unwinding/unrolling the loops. If the area of hardware is not enough for full loop unwinding, users can reduce the unrolling times in order to satisfy the resource constraint. The advantage of this strategy is to make good use of the area of FPGA.

5 Discussion and conclusion

This paper presents a study of code optimization forms for HLS based real-time image processing designs. We first explore the HLS process and the optimization forms available for it, then a code optimization strategy is proposed. In the experiments, the proposed approach is evaluated using four basic image processing test benches and compared with two other similar design flows: PolyComp and Vivado_HLS.

The experiment results demonstrate that the optimization method of this paper can more effectively speedup the design than the reference design flows. Since PolyComp is actually a transcompiler for general-purposed processor, its key technique is to improve the loop-level parallelism of the algorithm behaviors by reducing the data dependency between the loop iterations through polyhedral transformation. However, today's HLS tools are usually capable of parallelizing the design in the instruction level during the scheduling process depending on the given CDFGs [33], so polyhedral transformation will not bring additional efficient improvement comparing to the optimization form of loop unroll. Meanwhile, for Vivado_HLS, we can see that the built-in optimization directives have many use conditions. For example, *loop_merge* directive can fuse the consecutive loops to reduce the overall latency, increase sharing and improve logic optimization, but the code between the loops to be merged cannot have side effects. In our case, these use conditions seriously prevent the implementations from benefiting from the optimization forms that are available in either architecture or data dependency apspect.

Within the proposed optimization strategy, the applying ordering of optimization forms is taken into account. Table (2) illustrates that the first three optimization steps do not effectively improve the design, and the main contributor of the efficiency improvement is loop unroll. However, it should be noted that function inline and loop manipulation improve the loop nest and operation sharing by simplifying the function and control architecture, and symbol expression manipulation creates a more efficient operation scheduling. These transformations make much more potential optimization opportunities for the final loop unroll step. According to Table (3), we can see that although the loop nests are unrolled, Vivado_HLS does not provide a higher efficiency to the design than the proposed method.

Finally, we note that many code transformations mentioned in this paper are manually made, which may potentially increase the effort-cost of the development. Fortunately, our research demonstrates also that a source-to-source transcompiler, PoCC for example, is capable of handling this problem if the right compilation strategies are provided. In the future work, we will focus on the automatization of the code optimization process for HLS with the challenges of adopting the proposed optimization strategies into a c-to-c transcompiler. Meanwhile, some more complex real-time image processing algorithms will be used to further evaluate our method. Especially, the implementations optimized by our method and the FPGA experts will be compared in order to estimate the performance gap between the machine- and manually-made optimizations.

Acknowledgements The authors would like to thank the China Scholarship Council and the Conseil Régional de Bourgogne Franche-Comté for their funding of our studies.

References

1. H. Wang, N. Zhang, J. C. Crput, J. Moreau, and Y. Ruichek, "Parallel structured mesh generation with disparity maps by gpu implementation," *IEEE Transactions on Visualization and Computer Graphics*, vol. 21, no. 9, pp. 1045–1057, Sept 2015.
2. H. Wang, "Cellular Matrix for Parallel K-means and Local Search to Euclidean Grid Matching," Theses, Université de Technologie de Belfort-Montbéliard, Dec. 2015. [Online]. Available: https://tel.archives-ouvertes.fr/tel-01265951
3. C. Li, V. Brost, Y. Benezeth, F. Marzani, and F. Yang, "Design and evaluation of a parallel and optimized light-tissue interaction-based method for fast skin lesion assessment," *Journal of Real-Time Image Processing*, pp. 1–14, 2015. [Online]. Available: http://dx.doi.org/10.1007/s11554-015-0494-6
4. C. Li, S. Balla-Arabé, and F. Yang, "Embedded multi-spectral image processing for real-time medical application," *Journal of Systems Architecture*, vol. 64, pp. 26–36, March 2015. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S1383762115001526
5. C. Li, S. Balla-Arabé, D. Ginhac, and F. Yang, "Embedded implementation of vhr satellite image segmentation," *Sensors*, vol. 16, no. 6, p. 771, 2016. [Online]. Available: http://www.mdpi.com/1424-8220/16/6/771
6. K. Wakabayashi, "Use of high-level synthesis to generate hardware from software," *IEICE ESS Fundamentals Review*, vol. 6, no. 1, pp. 37–50, 2012.

7. J. Cong, B. Liu, S. Neuendorffer, J. Noguera, K. Vissers, and Z. Zhang, "High-level synthesis for fpgas: From prototyping to deployment," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 4, pp. 473–491, 2011.
8. F. Koichi, K. Kazushi, A. Shin-ya, Y. Masao, and N. Togawa, "A floorplan-driven high-level synthesis algorithm for multiplexer reduction targeting fpga designs," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E98.A, no. 7, pp. 1392–1405, 2015.
9. J. Cong, B. Liu, R. Prabhakar, and P. Zhang, "A study on the impact of compiler optimizations on high-level synthesis," in *Languages and Compilers for Parallel Computing*, ser. Lecture Notes in Computer Science, H. Kasahara and K. Kimura, Eds. Springer Berlin Heidelberg, 2013, vol. 7760, pp. 143–157. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-37658-0_10
10. I. Keisuke and K. Mineo, "Dual-edge-triggered flip-flop-based high-level synthesis with programmable duty cycle," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E96.A, no. 12, pp. 2689–2697, 2013.
11. *Vivado Design Suite User Guide*, Ug902(2012.2) ed., XILINX, July 2012.
12. G. Wang, *Catapult C Synthesis Work Flow Tutorial*, version 1.3 ed., ECE Department, Rice University, October 2010.
13. K. Wakabayashi, "C-based behavioral synthesis and verification analysis on industrial design examples," in *Proceedings of the 2004 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '04. Piscataway, NJ, USA: IEEE Press, 2004, pp. 344–348. [Online]. Available: http://dl.acm.org/citation.cfm?id=1015090.1015177
14. J. Villarreal, A. Park, W. Najjar, and R. Halstead, "Designing modular hardware accelerators in c with roccc 2.0," in *Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on*, May 2010, pp. 127–134.
15. Y. Liang, K. Rupnow, Y. Li, D. Min, M. N. Do, and D. Chen, "High-level synthesis: Productivity, performance, and software constraints," *Journal of Electrical and Computer Engineering*, vol. 2012, p. 14, 2012, article ID 649057. [Online]. Available: http://dx.doi.org/10.1155/2012/649057
16. J. Cong, M. Huang, and Y. Zou, "Accelerating fluid registration algorithm on multi-fpga platforms," in *Field Programmable Logic and Applications (FPL), 2011 International Conference on*, Sept 2011, pp. 50–57.
17. K. Rupnow, Y. Liang, Y. Li, D. Min, M. Do, and D. Chen, "High level synthesis of stereo matching: Productivity, performance, and software constraints," in *Field-Programmable Technology (FPT), 2011 International Conference on*. IEEE, 2011.
18. R. Rodrigues, J. Cardoso, and P. Diniz, "A data-driven approach for pipelining sequences of data-dependent loops," in *Field-Programmable Custom Computing Machines, 2007. FCCM 2007. 15th Annual IEEE Symposium on*, April 2007, pp. 219–228.

19. H. Ziegler, M. W. Hall, and P. Diniz, "Compiler-generated communication for pipelined fpga applications," in *Design Automation Conference, 2003. Proceedings*, June 2003, pp. 610–615.

20. J. Cong, Y. Fan, G. Han, W. Jiang, and Z. Zhang, "Behavior and communication co-optimization for systems with sequential communication media," in *Design Automation Conference, 2006 43rd ACM/IEEE*, 2006, pp. 675–678.

21. P. Li, L.-N. Pouchet, and J. Cong, "Throughput optimization for high-level synthesis using resource constraints," in *IMPACT 2014. Fourth International Workshop on Polyhedral Compilation Techniques. In conjunction with HiPEAC 2014*, Vienna, Austria, Jan 20, 2014.

22. Q. Huang, R. Lian, A. Canis, J. Choi, R. Xi, N. Calagar, S. Brown, and J. Anderson, "The effect of compiler optimizations on high-level synthesis-generated hardware," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 8, no. 3, pp. 14:1–14:26, May 2015. [Online]. Available: http://doi.acm.org/10.1145/2629547

23. G. Daniel D., D. Nikil D., W. Allen C-H, and L. Steve Y-L, *High-Level Synthesis: Introduction to Chip and System Design*, 1st ed. Springer US, 1992.

24. M. Mehrara, T. Jablin, D. Upton, D. August, K. Hazelwood, and S. Mahlke, "Multicore compilation strategies and challenges," *Signal Processing Magazine, IEEE*, vol. 26, no. 6, pp. 55–63, November 2009.

25. J. H. Ahn, M. Erez, and W. J. Dally, "Tradeoff between data-, instruction-, and thread-level parallelism in stream processors," in *Proceedings of the 21st Annual International Conference on Supercomputing*, ser. ICS '07. New York, NY, USA: ACM, 2007, pp. 126–137. [Online]. Available: http://doi.acm.org/10.1145/1274971.1274991

26. Xilinx, "Introduction to fpga design with vivado high-level synthesis," Xilinx, Tech. Rep. UG998 (v1.0), July 2013.

27. W. Zuo, Y. Liang, P. Li, K. Rupnow, D. Chen, and J. Cong, "Improving high level synthesis optimization opportunity through polyhedral transformations," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, ser. FPGA '13. New York, NY, USA: ACM, 2013, pp. 9–18. [Online]. Available: http://doi.acm.org/10.1145/2435264.2435271

28. L.-N. Pouchet, *PoCC. The Polyhedral Compiler Collection.*, version 1.2 ed., on line, Computer Science Department, University of California Los Angeles, 4731L Boelter Hall, Los Angeles, CA 90095. [Online]. Available: http://www.cs.ucla.edu/ pouchet/software/pocc/

29. A. M. Steven Derrien and A. Kumar, "S2s4hls-sp1 progress report," INRIA - University of Rennes 1, INRIA - ENS Cachan and INRIA - LIP, Tech. Rep., 2008.

30. A. Morvan, S. Derrien, and P. Quinton, "Efficient nested loop pipelining in high level synthesis using polyhedral bubble insertion," in *Field-Programmable Technology (FPT), 2011 International Conference on*, Dec 2011, pp. 1–10.

31. M. Alle, A. Morvan, and S. Derrien, "Runtime dependency analysis for loop pipelining in high-level synthesis," in *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, May 2013, pp.

1–10.

32. *Vivado Design Suite Tutorial*, Ug871(v2012.2) ed., XILINX, February 2012.
33. J.-H. Lee, Y.-C. Hsu, and Y.-L. Lin, "A new integer linear programming formulation for the scheduling problem in data path synthesis," in *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, Nov 1989, pp. 20–23.