J Real-Time Image Proc (2019) 16:1795–1812
DOI 10.1007/s11554-017-0688-1

ORIGINAL RESEARCH PAPER

Fast FPGA prototyping for real-time image processing with very high-level synthesis

Chao Li[1] · Yanjing Bi[2] · Franck Marzani[1] · Fan Yang[1]

Received: 8 November 2016 / Accepted: 2 April 2017 / Published online: 11 April 2017

Abstract Programming in high abstraction level can facilitate the development of digital signal processing systems. In the recent 20 years, high-level synthesis (HLS) has made significantly progress. This technique greatly benefits the R&D productivity of the Field Programmable Gate Array (FPGA) developments and helps for adding to the maintainability of the products by automating the C-to-RTL (register transfer language) conversion. However, due to the high complexity and computational intensity, image processing algorithms usually necessitate a higher abstraction environment than C-synthesis, and the current HLS tools do not have the ability of this kind. This paper presents a conception of very high-level synthesis method which allows fast prototyping and verifying the FPGA-based image processing designs in the MATLAB environment. We build a heterogeneous development flow by using currently available tool kits for verifying the proposed approach and evaluated it within two real-life applications. Experiment results demonstrate that it can effectively reduce the complexity of the development by automatically synthesizing the algorithm behavior from the user level into the low register transfer level and give play to the advantages of FPGA related to the other devices.

Keywords High-level synthesis · FPGA · Fast prototyping · Real-time image processing · Computer-aided design

✉ Yanjing Bi
yanjing.bi@hotmail.com

Chao Li
chao.li.1986@ieee.org

Franck Marzani
franck.marzani@u-bourgogne.fr

Fan Yang
fanyang@u-bourgogne.fr

[1] Laboratory Le2I UMR6306, CNRS, Arts et Métiers, Univ. Bourgogne Franche-Comté, 21000 Dijon, France

[2] Univ. Bourgogne Franche-Comté, 21000 Dijon, France

1 Introduction

In real-time image processing area, embedded systems are often involved. A precise definition of embedded system is not easy. Generally speaking, all computing systems other than general-purposed computer (with monitor, keyboard, etc.) are embedded systems. An embedded system inherently needs to embed a software into hardware, which makes it dedicated for a customized application or a specific part of an application.

FPGA is one of the frequently used embedded devices in digital signal processing for its significant advantages in terms of running-cost, power consumption and flexibility [5, 11, 27, 36, 40, 45, 46, 48, 51]. Advanced Digital Sciences Center (ADSC) of the University of Illinois reported that FPGA can achieve a speedup up to 2–2.5× and save 84–92% of the energy consumption comparing to Graphics Processing Units (GPUs) [18]. However, its users have to suffer from the complexity of the development flow due to the fact that the configuration of the devices of this type necessitates the low abstraction register transfer languages, such as VHSIC Hardware Description Language (VHDL) or Verilog. According to the evaluation results of Xilinx [60], RTLs-based FPGA designs consume the highest development period in all the devices available for digital signal processing. ADSC indicates also that a manual FPGA design may consume 6–18 months and even years

for a full custom hardware, while the GPUs (CUDA) based designs only 1–2 weeks [1]. Consequently, today's electronic manufacturers are increasingly pushed to select low development effort–cost devices by the pressure of market competition so that in most cases designers may accept the increased costs of performance, power or price in order to reduce development time. As a result, despite many advantages related to the other devices, i.e., general-purposed processors (GPPs), GPUs or digital signal processors (DSPs) [19, 22, 29, 64], FPGAs are usually applied only when the other devices cannot satisfy the design requirements. The abstraction level gap between the user-convenient and hardware-available languages seriously narrows the applications of FPGA in the image processing field.

Since the 1990s, many efforts have been made to develop a production quality high-level synthesis (HLS) technique for FPGA designs [2, 12, 13, 34, 35, 43, 54, 55, 63]. Figure 1 compares the conventional RTL with the HLS-based design flows by using Gasjki-Kuhn's Y-chart [39]. We can see that its philosophy is to provide an error-free path from abstract specifications to RTL, having significant values to the electronic manufacturers that constantly strive to improve their productivity by raising the abstraction level and easing the development process for their engineers.

Up to present, many high-quality HLS tools have been made available for practical applications [39, 42, 44], i.e., Vivado_HLS of Xilinx [59] and Catapult C-Synthesis Work Flow [57]. However, all the currently available HLS tools are based on C-synthesis techniques oriented to the scalar variables, whereas in the field of image processing a vector-oriented environment with higher abstraction level is usually required due to the high complexity and computationally intensity of the target algorithms. Basing on the achievements of HLS, we intend to conceive an automatical very high-level synthesis (VHLS) framework (briefly presented by Bi et al. [8] for the first time) with the following properties:

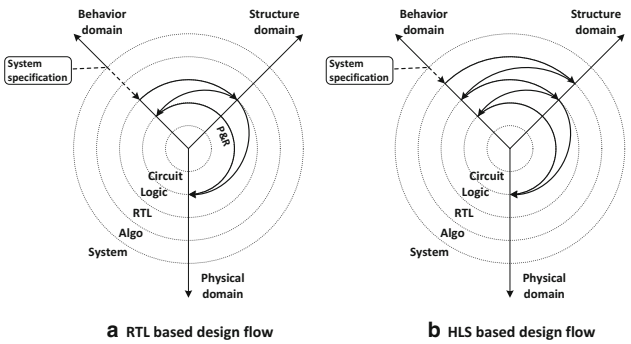


Fig. 1 Comparison of RTL- and HLS-based design flows by using Gasjki-Kuhn's Y-chart: *full lines* indicate the automated cycles, while *dotted lines* the manual cycles

(a) to handle the algorithm behavior described in very high-level languages, such as MATLAB or OpenCL,
(b) to handle the code written without FPGA expertise or even not for FPGA but the platforms of other types,
(c) to optimize the performances of the designs with the hardware constrains such as frequency or area of the target device,
(d) to automatically generate the desired RTLs in a short time rather than hours or even days,
(e) to be capable of being implemented by using the currently available electronic design automation (EDA) tools. This is important for industrial designs, because it can effectively reduce the R&D cost by helping for fast building the desired design space exploration (DSE) framework and avoiding the additional cost for the new tool kits.

We base the conception on MATLAB for its advantages in terms of vector processing and powerful built-in image processing tools. The challenges of MATLAB-to-RTL synthesis include:

(a) Operators in MATLAB perform different operations depending on the type of the operands, whereas the functions of the operators in RTL are fixed.
(b) MATLAB includes very simple and powerful vector operations such as the concatenation "[]"and column operators "x(:)" or "end" construct, which can be quite hard to map to RTL.
(c) MATLAB supports "polymorphism," whereas RTL does not. More precisely, functions in MATLAB are genetic and can process different types of input parameters. In the behaviors of RTL, each parameter has only a single given type, which cannot change.
(d) MATLAB supports dynamic loop bounds or vector size, whereas RTL requires users to initialize explicitly them and cannot do and changes during the synthesis.
(e) The variables in MATLAB can be reused for different contents (different types), whereas RTL does not, as each variable has one unique type.

For the issues above, we innovatively incorporate the source-to-source compile into the proposed VHLS framework in order to turn the nature of the source code from vector- into scalar-oriented programming. Finally, the generated code is classically synthesized via control & data flow extraction and RTL generation processes.

The proposed approach is evaluated by using two complex image processing applications: Kubelka-Munk genetic algorithm (KMGA) for the multispectral image-based skin lesion assessments [28] and level set method (LSM)-based algorithm for very high-resolution (VHR) satellite image segmentation [6]. Experiment results

demonstrate that the proposed approach can effectively reduce the complexities of description of the target algorithms. Meanwhile, the generated implementations of RTL have a higher performance than the reference implementations realized by using the other devices.

The remainder of this paper is organized as follows: Sect. 2 presents the proposed very high-level synthesis method. Section 3 discusses the challenges that we met when prototyping this conception, as well as the solutions. Section 4 evaluate the prototype of the proposed method and analyzes the experiment results. Finally, a conclusion is given in Sect. 5.

2 Proposed very high-level synthesis

This section presents the proposed VHLS method in detail (see Fig. 2). It consists of three steps: source-to-source compile (SSC), control and data extraction (CDE) and RTL generation. For each step, interdependent tasks are executed.

2.1 Source-to-source compile

As discussed before, the nature of vector-based representation of MATLAB and its powerful built-in tools result in many challenges for synthesizing it to RTL. We summed up these issues as the following three problems: dynamic variable problem, operation polymorphism problem and built-in function problem, and solve them by comply the MATLAB source code into a second intermediate code.

2.1.1 Dynamic variable problem

Since the corresponding memories can be re-allocated over and over for the new contents with different lengths (types), MATLAB users do not have to declare the types for variable initializing or allocate the right amount of memory for vector variable before each use. The variables can automatically change their lengths or dimensions depending on the content to be held. But none of the currently available register transfer languages have the dynamic variable ability like this. Therefore, as shown in Fig. 3a, we must allocate explicitly enough storage for every variable. In the proposed approach, a manually specified *define file* is required for memory allocation. Further more, when a variable is dynamically reused in the source code, such as x and X in the example, some additional definitions with other names are necessitated, i.e., x_1 and X_1 in the example.

2.1.2 Operation polymorphism problem

MATLAB allows operator and function polymorphism. That is, its operators or functions may support either a matrix or a scalar as a second operand or argument, and their returns are changed as well depending on the inputs. For the operators, the nature of each invocation has to be determined first. If scalar, they are mapped directly as the corresponding monomorphism ones of the target languages, else they are replaced with a loop construct as shown in the top of Fig. 3b. For the function mapping, the types of all the arguments should be determined as well. Depending on the invocation natures, we must create multiple different versions for the same MATLAB functions to satisfy the operations of different invocations as shown in the bottom of Fig. 3b.

We can see that either operator or function mapping may necessitate additional loop constructs in the present or deeper function level. In RTL, all the loop boundaries must be initialized in advance instead of using the unknown variables due to the fact that FPGA supports only static compile. In this paper, we use the vector information, which has been explicitly defined in *define file*, to help to compute the boundaries of the loops generated for operation polymorphism problem. Meanwhile, it should be noted

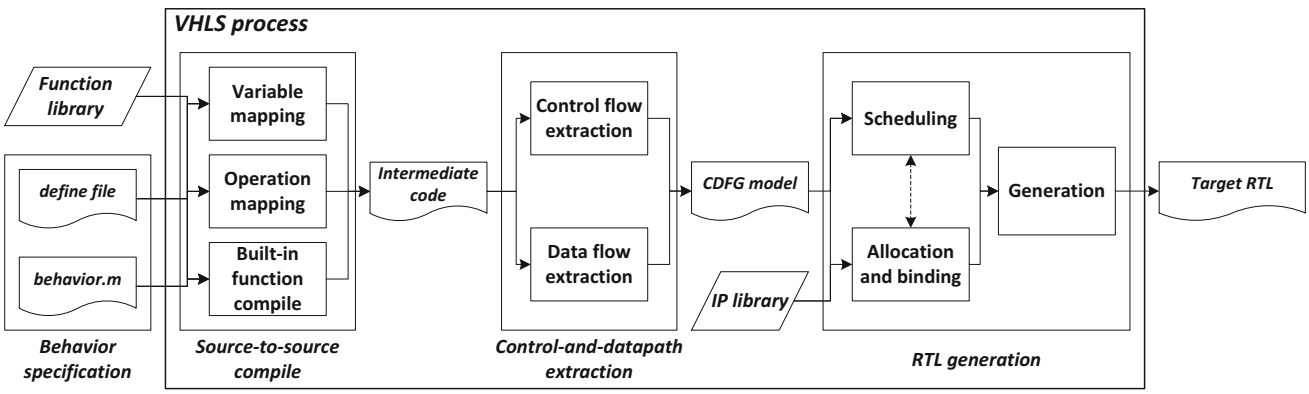


Fig. 2 Flowchart of the proposed VHLS method

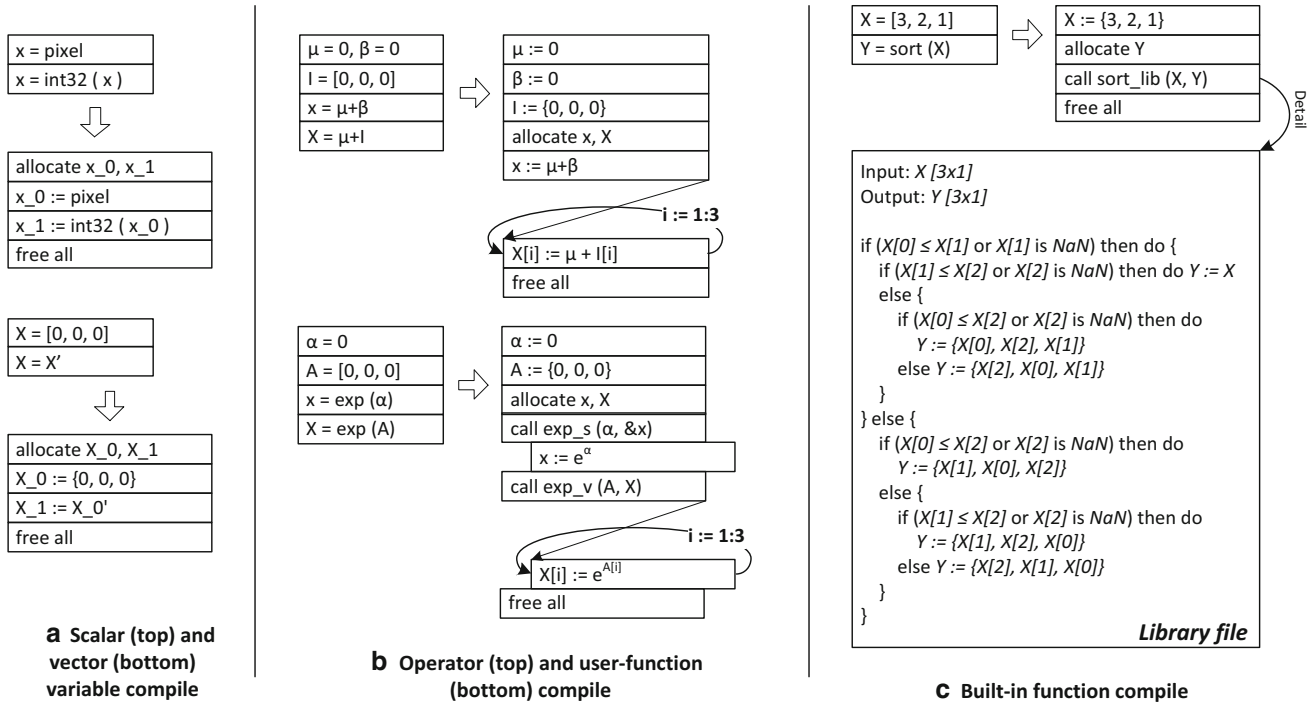


Fig. 3 Source-to-source compilation strategies for VHLS

that different vector dimensions may require more additional function versions even for the invocations that have same nature.

2.1.3 Built-in function problem

The built-in algorithms/functions of MATLAB provide users many benefits of facilitating their algorithm specifying, but they are usually invisible to a third-party compiler and detected as undefined functions when invoked. Therefore, we need to build a new library by using the synthesizable code for these powerful algorithms. As shown in Fig. 3c, the function *sort* of MATLAB is re-specified in the library. When it is invoked in the source code, the corresponding routine can be easily compiled by including its specification into the generated file. Since the built-in functions of MATLAB have also the polymorphism nature, we create different versions each. Depending on the argument types, the right version is re-targeted to the invocation in the source code.

2.2 Code and data extraction

Since there may be still a lot of variations between the semantics of the pre-compiled code and the target architectures, a canonical intermediate representation is needed for data dependency analysis. This issue can be formally solved by using the control and data flow graph (CDFG), which is one of the most widely accepted modeling paradigm for specifications that are processed by HLS tools.

A CDFG is a directed graph in which every node and arcs refer to a basic blocks B and control flows, respectively [16]. A basic block is defined as a straight-line sequence of statements without branches. As shown in the top of Fig. 4, a CDFG has two parts: control flow graph (CFG) and data flow graph (DFG). In the proposed approach, the data and control flows are firstly represented by using a DFG and a 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 such a way, the execution order of the process elements can be determined as well as its architecture.

The CDFGs can be classically created by using the finite-state machines (FSMs) with datapath as shown in the lower left of Fig. 4. This model is one of the most popular methods for digital system specification in register transfer level [60, 61]. 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 [15, 21, 49].

2.3 RTL generation

This subsection presents how the desired RTL is generated. To do this, three interdependent tasks are needed first, including scheduling, allocation and binding.

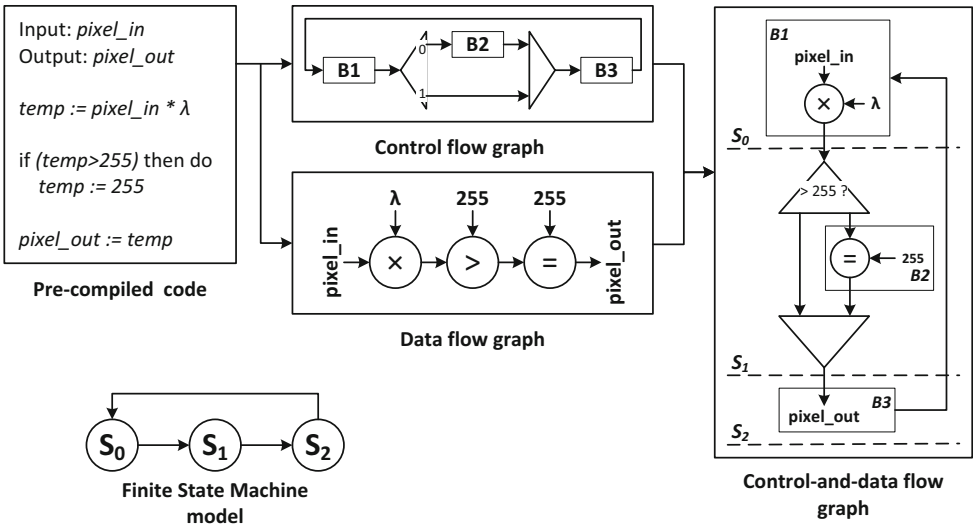


Fig. 4 Control-and-datapath extraction

2.3.1 Scheduling

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.

Up to the present, many efforts have been made in order to achieve more efficient scheduling [20, 24, 25, 38, 41, 52]. Integer linear programming algorithm [30] is the most widely used solution for this problem, its solver is guaranteed to find an optimal schedule from the problem models. In this algorithm, the mobility range of every operation $M = \{m_j | S_i \leqslant j \leqslant L_i\}$ is calculated first by using As Soon As Possible (ASAP) and As Late As Possible (ALAP) algorithms [20, 38, 52], where $[S_i, L_i]$ is the ASAP–ALAP range in which the i-th operation can be scheduled. Next, for the sake of convenience, let us suppose the extracted CDFG contains n operations and need to be scheduled into s steps. Each of the operations is denoted by o_i, where $1 \leqslant i \leqslant n$. A precedence relation between o_i and $o_{i'}$ is denoted by $o_i \rightarrow o_{i'}$, where o_i is the immediate predecessor of $o_{i'}$. The cost of a function unit of type t_k, defined as FU_{t_k}, is c_{t_k}, and there are m types of functions unites are available. Let M_{t_k} be the number of FU_{t_k}. $x_{i,j}$ is a Boolean variable associated with o_i: $x_{i,j} = 1$ if o_i is placed into step j, otherwise 0. Now the scheduling problem can be formulated as follows:

$$\text{Minimize}\left\{\sum_{k=1}^{m} c_{t_k} M_{t_k}\right\} \tag{1}$$

subject to

$$\sum_{\substack{i=1 \\ o_i \in FU_{t_k}}}^{n} x_{i,j} - M_{t_k} \leqslant 0, \quad \text{for} 1 \leqslant \text{j} \leqslant \text{s}, 1 \leqslant \text{k} \leqslant \text{m} \tag{2}$$

$$\sum_{j=S_i}^{L_i} x_{i,j} = 1, \quad \text{for} 1 \leqslant \text{i} \leqslant \text{n} \tag{3}$$

$$\sum_{j=S_i}^{L_i} j \times x_{i,j} - \sum_{j=S_{i'}}^{L_{i'}} j \times x_{i',j} \leqslant -1, \quad \text{for all} \quad \text{o}_\text{i} \rightarrow \text{o}_{\text{i}'} \tag{4}$$

The objective function in Eq. 1 states that we are going to minimize the total function unit cost. Equation 2 is the constraint that there should be no more than M_{t_k} function units of type t_k in a single step. Equation 3 states that the operation o_i can only be scheduled into a step between S_i and L_i. Equation 4 ensures the precedence relations among the operations extracted from the control and data flow graph.

2.3.2 Allocation and binding

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 depending on the scheduled CDFG. Next, the desired hardware resources are selected from the given 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 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. In additional, if the interconnects are shared by multiple data transfers, a multiplexer will be needed between the sources and destinations.

Since an IP library usually has several alternatives for a given functional unit or a storage unit, i.e., ripple-carry adder versus carry-look-ahead adder or reset-only register versus the register with both presets and reset, besides the right function, the selected hardware must be able to optimize the generated RTL or minimize either the total cost of source consumption or interconnection length with the design constraints. This allocation problem can be formulated as a clique-partitioning problem classically in HLS [17, 23, 52].

A clique-partitioning problem is to partition an undirected graph $G_c = (V_c, E_c)$ into a minimal number of cliques such that each vertex belongs to exactly one clique, in which V_c is a set of vertices, E_c is a set of edges and a clique is a set of vertices that form a complete subgraph of G_c. Figure 5a displays a typical example of function unit allocation solved by using this method. For functional unit allocation, a vertex represents an operation. The two vertexes are connected by an edge if and only if they are scheduled into different steps and can be carried out by a single functional unit. On the other hand, for the storage unit allocation problem, as shown in Fig. 5b, a vertex represents a value r_* needed to be stored, and the two vertexes are connected if and only if the lifetime

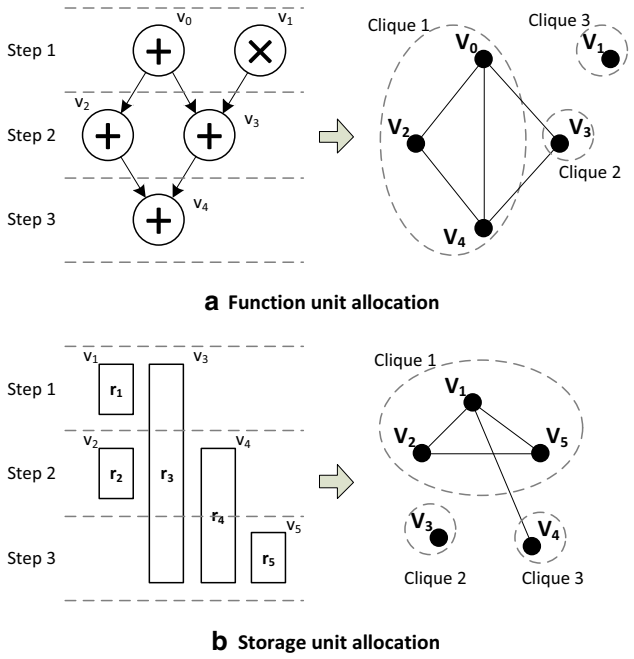


Fig. 5 Examples of clique-partitioning problems and solutions for function and storage unit allocations

of the corresponding values do not intersect. Historically, this problem can be solved by using Tseng and Siewiork's algorithm [52].

2.3.3 Generation

Once all the decisions of the preceding tasks, including scheduling, allocation and binding, have been made, we can start to generate the desired RTL. Figure 6 shows a typical example of the register transfer level architecture generated through the proposed approach. The FSM diagram is mapped as a logic controller to orchestrate the data flow through the control signals, i.e., selecting the right inputs for functional units, registers or multiplexers. This architecture consists of state logic, state register and output logic. The state register stores the present state of the data path, while the state logic computes the next state to be loaded depending on the control inputs from the external world and the state signals from the processor (data path architecture). Finally, the corresponding control signals and control outputs are generated and exports by output logic.

On the other hand, the computing of each state is mapped as a data path architecture that consists of a set of storage components, a set of functional units and connectivity components. The quantities and types of storage components and functional units are allocated according to the decision made in the allocation and binding tasks, then arbitrarily connected through connectivity components. The interconnection behavior of data path architecture is described in register level language. Algorithm 1 shows the data path behavior description of the RGB-to-gray level transformation. In this example, three multipliers (FU_1, FU_2 and FU_3) are allocated to minimize the latency by parallelizing the production operations. The two addition operations are serially scheduled due to the data dependency, and because of this, only a single adder is allocated to minimize the resource consumption.

Algorithm 1 Data path architecture of RGB-to-Gray transformation: $I = 0.2989 \times I_r + 0.5870 \times I_g + 0.1140 \times I_b$

Require: I_r, I_g, I_b, S_{ctrl}
Ensure: I, S_{state}
1: $W_{ctrl} := S_{ctrl}$
2: $\{Register\_1, Register\_2, Register\_3\} := \{I_r, I_g, I_b\}$
3: $Bus\_3 := FU_1(W_{ctrl}, I_r, 0.2989)$
4: $Register\_1 := Bus\_3$
5: $Bus\_3 := FU_2(W_{ctrl}, I_g, 0.5870)$
6: $Register\_2 := Bus\_3$
7: $Bus\_3 := FU_3(W_{ctrl}, I_r, 0.1140)$
8: $Register\_3 := Bus\_3$
9: $Bus\_3 := FU_4(W_{ctrl}, Register\_1, Register\_2)$
10: $Register\_1 := Bus\_3$
11: $Bus\_3 := FU_4(W_{ctrl}, Register\_1, Register\_3)$
12: $Register\_1 := Bus\_3$
13: $I := Register\_1$
14: $S_{state} :=$ go to next state

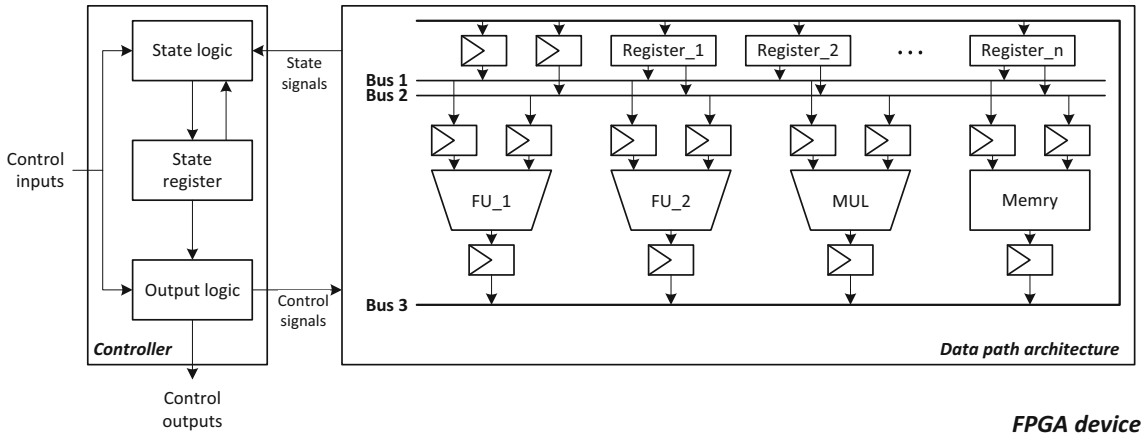


Fig. 6 An example of VHLS-generated FPGA architecture

3 Prototype of VHLS process

According to Sect. 2, we can see that the proposed VHLS method is hard to be implemented. In this section, we will discuss its challenges and solutions.

3.1 Work flow

The work flow for the proposed VHLS method is shown in Fig. 7, within which the following tasks should be done:

(a) Users first specify and functionally verify their algorithm behavior by using classical MATLAB.
(b) Next, the verified MATLAB code is compiled into the intermediate code. Due to the compatibility problem between the tools (discussed later), the code needs to be further transformed for the subsequent steps. Additionally, a second function verification can be done by using the corresponding compilers if necessary.
(c) Thirdly, the verified intermediate code is synthesized into RTL through a synthesis tool with the user-specified and hardware constraints.
(d) Fourthly, the generated RTL is evaluated. If the results satisfy the design requirement, go to the next step, otherwise go back to the first step.
(e) Finally, the evaluated RTL is send back to the MATLAB environment for co-simulation. If its logic verification is satisfied, work end, otherwise go back to the first step.

3.2 Intermediate code

The first step of the proposed approach is to compile the source code from MATLAB into an intermediate code. In this paper, ANSI C is selected for the following reasons:

(a) It is a scalar-based language with high compatibility, which can satisfy most requirements of synthesis process related to language natures.
(b) It can be compiled as either source or destination code by many mature source-to-source compilers, which can facilitate the discovering of the alternative solutions for source-to-source compile task. This is important for DSE because the description methods or code style could influence the performances of the generated RTL [14].
(c) It is supported by most available FPGA synthesis tools. Since the qualities of the generated RTL are essentially decided by the synthesis strategies, i.e., scheduling and binding algorithms, so different tools may result in different design results. Being supported by multiple synthesis tools allows easy experiments for solution improvements.

Basing on this decision, MATLAB Coder is selected for the MATLAB-to-C conversion.

3.3 Intermediate code versus RTL

In the electronic design automation world, many high-quality HLS tools have been made available and used in real-life applications [10, 42, 47, 53]. Meeus et al. [39] evaluates the main characteristics of the most currently available commercial HLS tools and represented the results by using the 5-level spider web diagrams. According to their achievements, we selected AutoPilot as our synthesis tool from five candidates, including AutoPilot [62], Catapult C [57], C-to-Silicon [9], Cyber Workbench [56] and Synphony C [50]. Table 1 compares their characteristics, and Table 2 describes its score levels. We can find that AutoPilot has a near perfect performance related to the others. It can benefit the desired design suite in the following aspects:

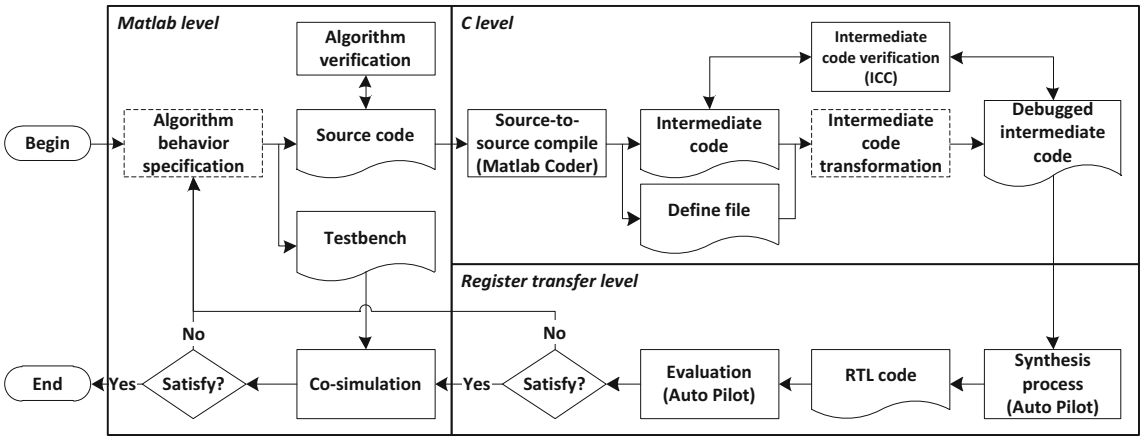


Fig. 7 VHLS-based work flow

Table 1 Evaluation of HLS tool candidates: "*" refers to the selected HLS tool for the proposed approach

	Source	Abstraction level	Data type	Design exploration	Verification	RTL quality	Ease of implementation
AutoPilot*	C/C++, System C	++	+	++	++	++	++
Catapult C	C/C++, System C	++	+	++	++	+	++
C-to-Silicon	C, TLM, System C	+	+	−	+	+−	+
Cyber Workbench	C, System C, BDL	++	+	++	+	++	+
Synphony C	C/C++	++	+	+−	++	+−	++

(a) Scheduling untimed code (operations) without any restrictions. Since MATLAB Coder does not have the ability of code timing, this advantage allows us to move the intermediated code into it without further processing for time control.
(b) Handling floating point variable by mapping them into fixed point. This facilitates the designs of high accuracy applications.
(c) Providing extensive and intuitive exploration options. With AutoPilot, users can discover new solution alternatives by re-configuring the optimization directives instead of modifying the source code.
(d) Ability of estimating the running-cost of the design. This can facilitate the evaluation of the design candidates and making the final decision.
(e) Generating high-quality RTLs. Designers always strive for better design performances within their powers. According to the measuring of Meeus et al. [39], AutoPilot could save up to more than 95% of hardware sources (slices) compared to the other HLS tools.
(f) Easy to be learned even for those who is knowledge less with FPGA. This tool can be quickly mastered by C users and requires less hardware knowledge. Furthermore, its documentation is extensive and easy to understand.

3.4 SSC versus HLS

As shown in Fig. 3, besides of the algorithm behavior, a *define file* is needed to specify the size of the vector variables used in the behavior file *behavior.m*. Moreover, since MATLAB Coder is inherently designed for the MEX function rather than high-level synthesis, the compile process needs to be re-configured to ensure the C code generated by default amenable to the C-synthesis process. For the sake of convenience, let us take the example shown in Fig. 8 as instance to show how we make it. Figure 8a displays the user-specified algorithm behavior, which first transforms the input color image I_{rgb} into gray level then enhances it using *log* transformation. Figure 8b is the *define file* specified manually in the format of MATLAB script (*.*m*), in which the size of the input argument I_{rgb} is defined by using function *zeros*() with an element type of single-precision floating point number. Next, the desired C code can be generated by entering the following commands in the command window of MATLAB:

```
≫ run define.m;
≫ cfg = coder.config('lib');
≫ codegen myfunc_vhls -config cfg -args{Irgb};
```

The first command above is to create a set of vector or matrix variables in the workspace of MATLAB by running

Table 2 Symbol definition of Table 1

	++	+	+−	−	−−
Abstraction level	Untimed code	Untimed code with restrictions	Timed code	Block level design	
Data type	Floating and fixed point number	Fixed point number	Need data conversion	Neither floating nor fixed point number	
Design exploration	Intuitive exploration options	Non-intuitive exploration options	By swapping predefined blocks	Limited exploration capabilities	
Verification	Support either source or generated code	Require data conversion	Support source code only	No verification support	
RTL quality (slices)	Less than 300	300–1200	1200–2100	2100–3000	More than 3000
Ease of implementation	No code modifications	Less code modifications	A number of code modifications	Many code modifications	Rewrite code

The evaluation results of the tool candidates are represented by using five levels, −−, −, +−, + and ++ from low to high, respectively [39]. The abstraction level evaluates the ability of the tools in terms of timing control. Data types refer to the ability of data format supporting, such as floating or fixed point data. Exploration is the design space exploration ability of the tool. Verification evaluates whether the tool can help to generate easy-to-use test benches quickly. RTL quality is estimated by the amount of the hardware consumption of the generated RTL presented in [39]. Ease of implementation means whether the original source code could be used with less modifications or need to rewrite completely

the *define file*. The elements of these variables are initialized as zeros, but contain the desired size information of the corresponding arguments of the algorithm behavior. Next, we modify the transformation constraints of MATLAB Coder from *mex* (for MEX functions) to *lib* (for C/C++ Static Library). Finally, MATLAB-to-C transformation is run with the defined configuration (*-config cfg*) and information of the input vector (*-args {Irgb}*).

In the generated code shown in Fig. 8(c), we can see that the boundary of the loop nest is automatically detected during the code transformation depending on the given size information of the input vector I_{rgb}, as well as the size of the two output vectors I_{gray} and I_{log}. However, it should be noted that compatibility problems still exist potentially due to the source code constraints of AutoPilot, including but not limited to:

(a) The memory manipulation statements frequently used by MATLAB Coder, i.e., *malloc()* and *memcpy()*, are not supported by AutoPilot;
(b) Static variable is not amenable to AutoPilot because FPGA does not have the storage element of this type;
(c) Not all the MATLAB built-in functions are support by MATLAB Coder.

Therefore, before CDFG extraction, once these problems emerge, as shown in Fig. 7, an *"Intermediate code transformation"* task must be done. For example, the memory allocation statements can be mapped into the common array declarations, while the static variable into a variable out of the top entity and stored in the external memory.

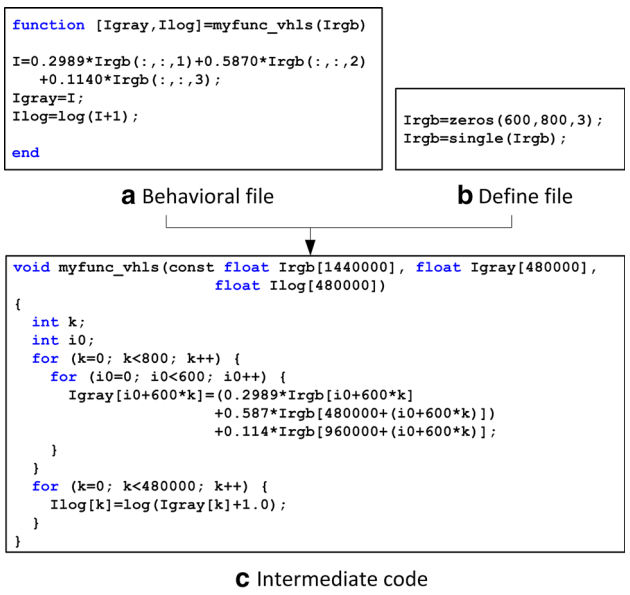


Fig. 8 An example of source-to-source compile

3.5 Verification and evaluation

According to Fig. 7, we can see that three verification and one evaluation tools are required. The source code can be easily verified by using the compiler of MATLAB. Meanwhile, we select Intel C++ Compiler (ICC) to debug the intermediate C code and evaluate the generated RTL by using AutoPilot directly.

Since the algorithm behavior is specified within a MATLAB environment, the final co-simulation should be performed between MATLAB and RTL. To do this, System Generator/Simulink [58] is suggested. This tool provides a visual programming environment which can facilitate the building of the testbench and profiting from the data base already configured in MATLAB.

4 Experiments

The experiments of the proposed approach are conducted by using two real-life applications: KMGA method [28] and LSM-based image segmentation algorithm [6] (defined as KMGA and LSM in this paper). The two algorithms are selected for the following two reasons: a) both of them have complex iterative architectures, which are effort-cost to be implemented by using conventional FPGA design flows, and b) KMGA have a low parallelism while LSM allows a Lattice Boltzmann Method (LBM) solver, which has a high parallelism. This allows us to compare the performance difference of the proposed approach in two different cases.

This section first analyzes the synthesis process of KMGA in detail in order to illustrate how VHLS works in the real-life applications that evaluates the implementations of the two algorithms from different aspects, including accuracy, efficiency, code intensity and development cost. For the purpose of concision, we present only the experiment results of LSM in this paper, but readers can consult the Reference [6] for more details about it if desired.

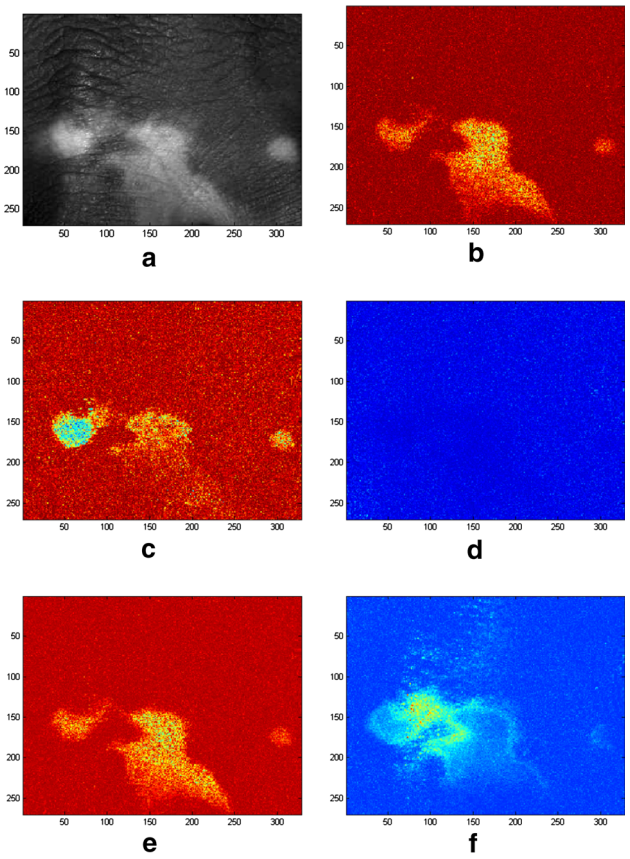


Fig. 9 Skin parameter maps retrieved through KMGA: **a** reflectance image of Vitiligo at 550 nm, **b** volume fraction of melanosome map, **c** volume fraction of hemoglobin map, **d** relative blood oxygenation map, **e** epidermis thickness map and **f** dermis thickness map

4.1 Synthesis process of KMGA

KMGA is a promising technique for the computer-aided skin lesion assessment. As shown in Fig. 9, this method can retrieve the skin parameter maps from a multispectral reflectance cube to perform a second quantified diagnostic opinion in consultation. Its algorithm behavior needs to combine a light-tissue model, Kubelka-Munk (KM) function, with a function optimizer of genetic algorithm (GA). In other words, this algorithm inverses the KM process by using the genetic algorithm. In our experiments we use a reduced 2-layer KM function [33] and a nature evaluation process for function optimizing.

Algorithm 2 KMGA method

Require: Reflectance array: R
Ensure: Skin parameter array: P_{skin}
Initialize *seed* of *Random_Generator*
$Pop :=$ initialize population
$P_{fitness} :=$ positions of fitness genes
$Pop(P_{fitness}) :=$ get fitness value via KM model
for all iterations **do**
  $P_{best} :=$ positions of best individuals
  if *iter* = the last iteration **then**
    $P_{skin} :=$ the best individual
  else
    for $i \in$ positions of all the best individuals **do**
      $NewPop(i) :=$ get best individuals
    end for
    $P_{random} :=$ positions of random individuals
    for $i \in$ positions of all the random individuals **do**
      $NewPop(i) :=$ get random individuals
    end for
    $P_{cross} :=$ positions of cross individuals
    for $i \in$ positions of all the cross pairs **do**
      $p :=$ position of cross gene
      Swap cross genes
    end for
    for $i \in$ positions of all the cross individuals **do**
      $NewPop(i) :=$ get cross individuals
    end for
    $P_{mutation} :=$ position of mutation individuals
    for $i \in$ positions of all the mutation individuals **do**
      $p :=$ position of mutation gene
      $NewPop(i \times 6 + p) :=$ renew the mutation gene
    end for
    $NewPop(P_{fitness}) :=$ get fitness value via KM
    $Pop := NewPop$
  end if
end for

The overall flowchart of KMGA is described in Algorithm 2. Firstly, the individuals are randomly generated within reasonable range to form an initial population. Next, in an iterative framework, population evolves using techniques inspired by natural evolution. The evolution of the population is composed of three steps: best-individual selection, crossover-mutation and random selection. During the selection process, only the best individuals are kept

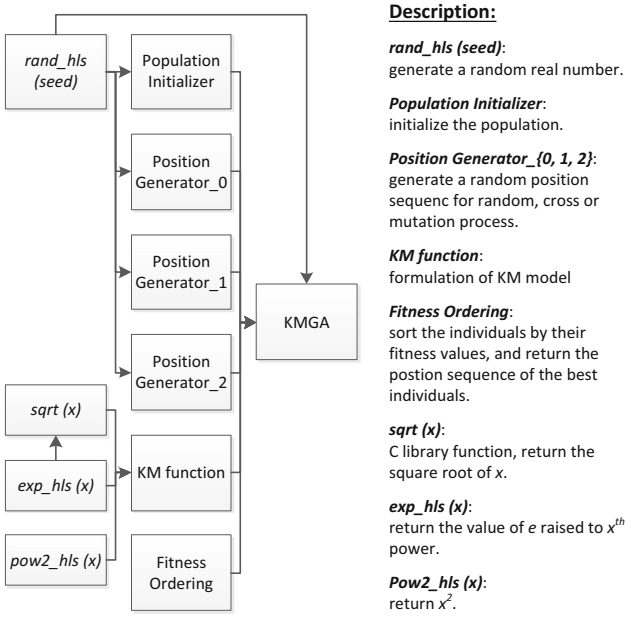


Fig. 10 Function hierarchy of KMGA within transformed intermediate C code

for the next iteration. Then, crossover process selects multiple couples of individuals from the population and one crossover parameter from the five skin parameters to create two new individuals (offsprings) by swapping the parents' selected parameter values. Finally, mutation process randomly generates some new skin parameter values to replace certain individuals' old ones. These processes are repeated until a predefined number of iterations. Finally, the best candidate is selected.

Figure 10 displays the function hierarchy of the transformed intermediate code. Firstly, in the source code, we specify a single random position sequence generator for the selection of random, cross and mutation individuals. This method handles the generations of the three size-different random sequences by taking advantage of the polymorphism of MATLAB functions. In the intermediate code, this function is mapped to three similar functions, *Position Generator_{0, 1, 2}*, depending on the invocation natures. Next, we can see that four C library functions, *rand ()*, *sqrt ()*, *exp ()* and *pow ()* are invoked in the MATLAB Coder-generated code. However, *rand ()*, *exp ()* and *pow ()* are not supported by the selected HLS tool because they necessitate static variables. We therefore map them to the design library functions, *rand_hls (seed)*, *exp_hls ()* and *pow2_hls ()*. It should be noted that in *rand ()*, a seed variable must be defined to initialize it. In ANSI C it is declared as static variable, and we transform it into global one and allocate in the external memory in our case.

The left schematic diagram of Fig. 11 displays the CDFG of the top behavior of KMGA which is scheduled as a FSM with four states labeled S_0, S_1, S_2 and S_3. In state S_0, the *Population Initialization* block waits for the control signal of *ap_start*. When the population is initialized, the state transits to S_1 to compute the fitness values of the randomly generated individuals by using KM function. Next, the population starts to evolve. In the evolution process, the population is first sorted depending on their fitness, and then the present iteration count is verified to see whether reach to the defined maximum iteration number. If yes, the information of the best individual is exported, otherwise the evolution continues. When the present iteration finishes, the FSM controller goes back to the entrance of S_2.

It should be noted that the desired KMGA is complex, so we map the functions of intermediate code to the behavioral instances to enforce the robustness of the design. In Fig. 11, we illustrate also the details of CFGs of the basic blocks of KMGA. We can see that these sub-blocks are implemented by using FSM architectures as well as the top behavior.

Table 3 lists the hardware elements required by the generated RTL with the device of *xc7vc1140tflg1928-1* of Virtex7. During the generation process, two user-specified basic blocks, KM function and Fitness Ordering, are generated, while the others are inlined into the top behavior. Next, nine types of different library functional units are allocated. Despite high complexity, the algorithm behavior does not result in a quantity skyrocket for each unit. This is because the selected HLS tool can effectively handle the scheduling and allocation problem to benefits the area cost performance. For storage, 101 registers and 7 BRAM are allocated respectively. Finally, 39 multiplexers and 117 other units are allocated for logic control.

4.2 Evaluations of KMGA and LSM implementations

Table 4 describes the implementations used for this evaluation experiments. In order to obtain an unbias comparison, the different implementations of MATLAB and VHLS versions are developed from the same algorithm behavior described in MATLAB, while the two C versions are manually implemented additionally.

4.2.1 Function verification

We first functionally verified the generated RTLs, *kmga_fpga_vhls* and *lsm_fpga_vhls*. Figure 12 statistically analyzes the final fitness values of the evolution processes of all the KMGA implementations using box-whisker plot, in which a box with whisker is produced for each data set of the corresponding measure results, the red central mark is the median value, the edges of the blue boxes are 25th

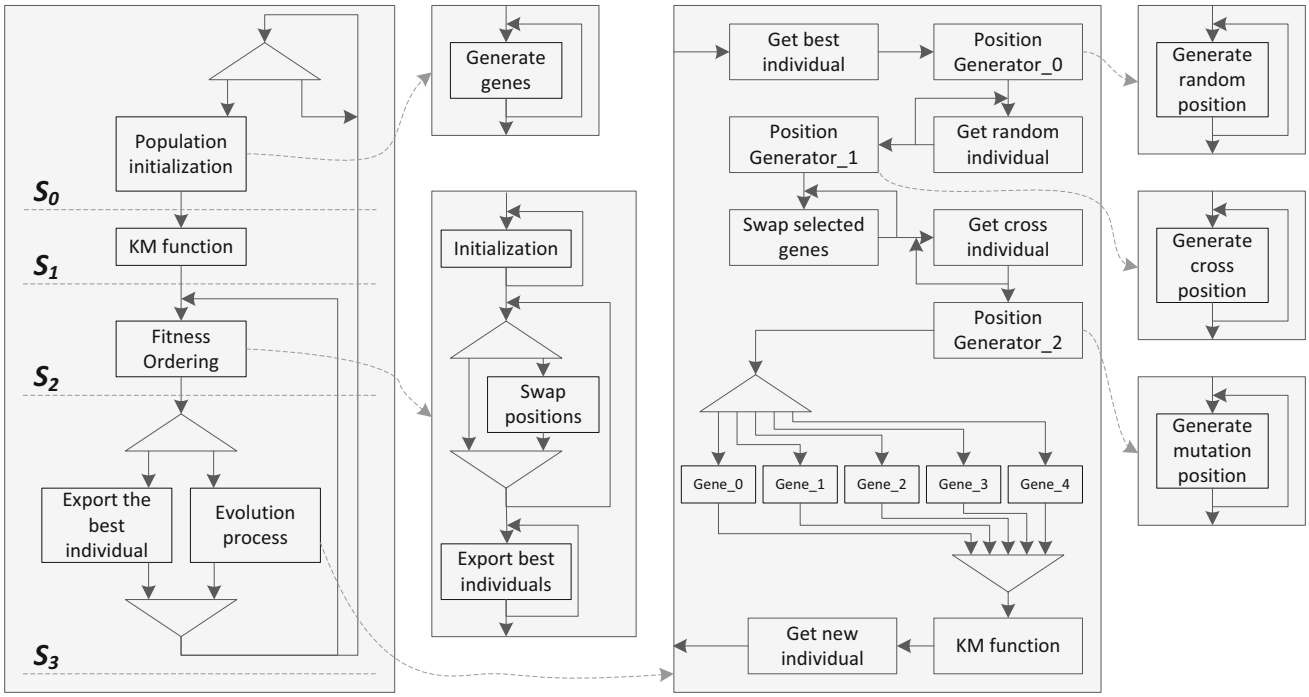


Fig. 11 CDFG of KMGA

and 75th percentiles, the black whiskers extend to the most extreme data points. The measurements are conducted by using 100 skin spectral samples, and lower fitness value is better. We can see that *kmga_cpu_m* has the lowest fitness value in a whole, whereas the *kmga_fpga_vhls* the highest. That is, the accuracy performance of the VHLS implementation is lower than either MATLAB or C implementation.

On the other hand, the function of LSM is verified by observing the similarities between the segmentation results of the implementations and the ground truth images. Table 5 shows the image similarities tested by using the three images taken by IKONOS and GeoEye-1 shown in Fig. 13. The image similarities are figured out by using the built-in function *corr2()* of MATLAB. We can see as well that *lsm_fpga_vhls* has different segmentation results with the other three implementations.

The differences of the obtained experiment results are caused by two reasons: (a) the evolution process of KMGA is random, so the results are hardly identical for each

Table 3 Hardware element estimation of KMGA:"*" refers to the user-specified behaviors

Element	Description	Quan.
KM function*	Tissue-light interaction model of skin depending on the fitness	1
Fitness Ordering*	Sort the individuals	1
dadd	Double-precision floating point addition	1
ddiv	Double-precision floating point division	2
dmul	Double-precision floating point multiplication	2
fadd	Single-precision floating point addition	1
fpext	Single-precision floating point sign-extension	2
fptrunc	Single-precision floating point truncation	2
sitodp	Signed-integer to double-precision floating point conversion	1
sitofp	Signed-integer to single-precision floating point conversion	2
urem	Unsigned Remainder	11
Register	–	101
Memory	Internal storage space (BRAM)	7
Multiplexer	–	39
Others	Operators/components for logic control	117

running, and (b) we convert the data type from *double* to *float* to save the area consumption of the design, which results in a lower accuracy. Further more, in C-synthesis process, the single-precision floating point numbers are further mapped to the fixed point ones, which results in a second change of experiment result. However, all these differences are tiny, so it will not seriously influence the performances of the final implementations. Despite the reduction of accuracy, these experiments demonstrate that the proposed VHLS can generate the desired implementations with the correct functions.

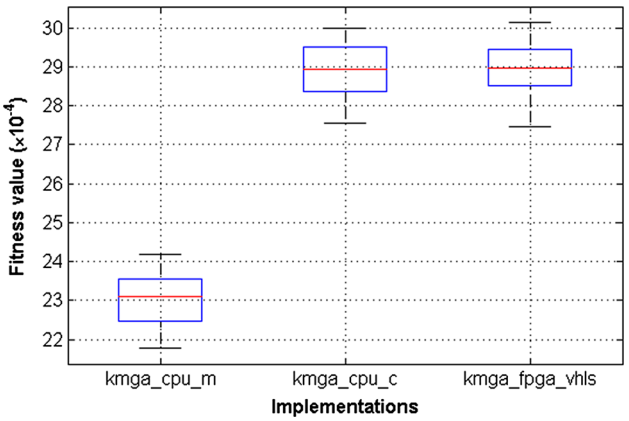


Fig. 12 Fitness values of KMGA implementations

Table 5 Segmentation result similarities of the LSM implementations

	Uxmal	Volcano	Ice sheet
lsm_cpu_m	0.9131	0.9722	0.9514
lsm_cpu_c	0.9452	0.9977	0.9812

4.2.2 Hardware estimate

We evaluate the hardware occupancy of VHLS by comparing the hardware consumption of *kmga_fpga_vhls* and *lsm_fpga_vhls* with two reference implementations. The reference implementation of KMGA, FPGA-KMGA [32] is realized through a C-synthesis based design flow and optimized by using the directives of Vivado_HLS, while the one of LSM through a manual MATLAB-to-RTL development framework proposed by Chao et al. [31] without any optimizations.

As shown in Table 6, it is first found that neither of the two implementations consumes a large number of resources. *kmga_fpga_vhls* takes up only 4% of look-up tables (LUTs) of the target device, while *lsm_fpga_vhls* only 55.5%. This is because the selected HLS tool, AutoPilot, provides a high-quality scheduling, allocation and binding. Secondly, since *FPGA-KMGA* is accelerated by using a series of optimization methods, requiring more components to parallelize the operations, it costs much more hardware resources than *kmga_fpga_vhls*. Meanwhile, because neither of *lsm_fpga_vhls* and its reference implementation is optimized, their hardware consumptions are very close.

4.2.3 Efficiency

Thirdly, we show the running speed ratio in Fig. 14. This experiment is made by using a standard skin lesion reflectance cube retrieved through a neural network designed by Mansouri et al. [37] for KMGA, and the photos taken by the satellite IKONOS for LSM. The MATLAB CPU implementations of each algorithm are set as the reference. We can see that the proposed VHLS method achieves a speedup of around 6× and 1.42× for *kmga_fpga_vhls* versus *kmga_cpu_m* and *lsm_fpga_vhls* versus *lsm_cpu_m*. This demonstrates that our approach can effectively give play to the advantages of Field Programmable Gate Arrays (FPGAs) in terms of running speed when the design is made in the same high abstraction level.

Related to the C implementations, our approach achieves a speedup of 2.2× for KMGA but reduces the running speed of LSM by 46.3%. This is because the selected C compiler, ICC, allows optimizations of different forms [3, 4, 7, 26]. In this experiment, "Maximize speed" mode is used to schedule the algorithm automatically. Since the evolution process of KMGA is an iteration-dependent loop, the parallel optimizations, such as vectorization or Streaming SIMD Extensions (SSE), cannot

Table 4 Implementation description

Algorithm	Implementation	Tool	Hardware
KMGA [28]	kmga_cpu_m	MATLAB R2012a	Intel Q6600 processor
	kmga_cpu_c	ICC 13.1.1	Intel Q6600 processor
	kmga_fpga_vhls	proposed VHLS	Xilinx Virtex-7
LSM [6]	lsm_cpu_m	MATLAB R2012a	Intel Q6600 processor
	lsm_cpu_c	ICC 13.1.1	Intel Q6600 processor
	lsm_fpga_vhls	proposed VHLS	Xilinx Kintex-7

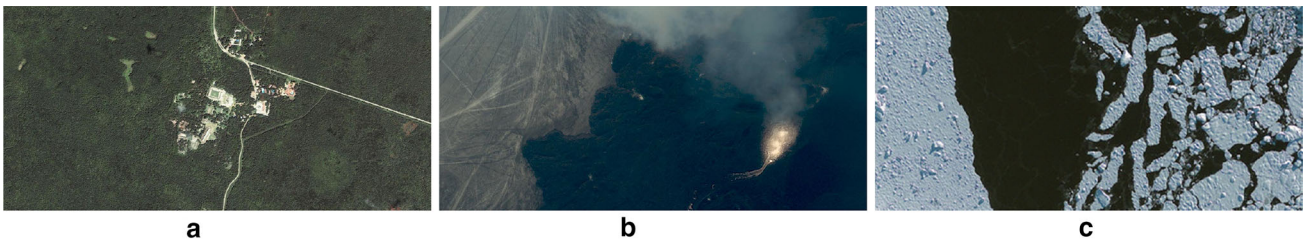


Fig. 13 Photos taken by IKONOS and GeoEye-1 for LSM evaluation: **a** Uxmal, **b** Volcano and **c** Ice sheet

benefit the running-cost performance of the design. In contrast, LSM has a high parallelism, so this method produces a very high-quality implementation. But it should be noted that the comparisons between the C- and VHLS-based implementations are not fair, because the former must be developed in the lower C abstraction level, which usually requires more design efforts than the latter. Furthermore, this performance gap is not impassable, because we do not make any parallel optimizations in the VHLS-based implementations yet.

4.2.4 *Productivity*

The code intensities of different implementations are first evaluated using the line quantities of their routines as metric. As shown in Fig. 15, we can see that the VHLS-based implementations have the same code intensities with the MATLAB ones. This demonstrates that the proposed approach has a high compatibility with MATLAB. Meanwhile, the line quantities of *kmga_fpga_vhls* and *lsm_fpga_vhls* are only around 50% of their C code, so it is concluded that the proposed approach can effectively reduce the complexity of the design related the reference methods.

Next, the effort consumption of the manual and VHLS-based MATLAB-to-C transplantations is estimated using the metric $time \times persons$. In this case, both of the MATLAB implementations of the test algorithms are manually made by the image processing experts with little FPGA knowledge, then transplanted into to the C code amenable to HLS tools.

During the implementation work, it is found that the manual design flow requires an effort costly algorithm analysis cycle for the hardware experts to understand the fundamentals of the algorithm, and this cycle takes up around half the work efforts. During the code conversion, hardware experts must re-specify the algorithm behavior manually line-by-line in C level, whereas it can be done

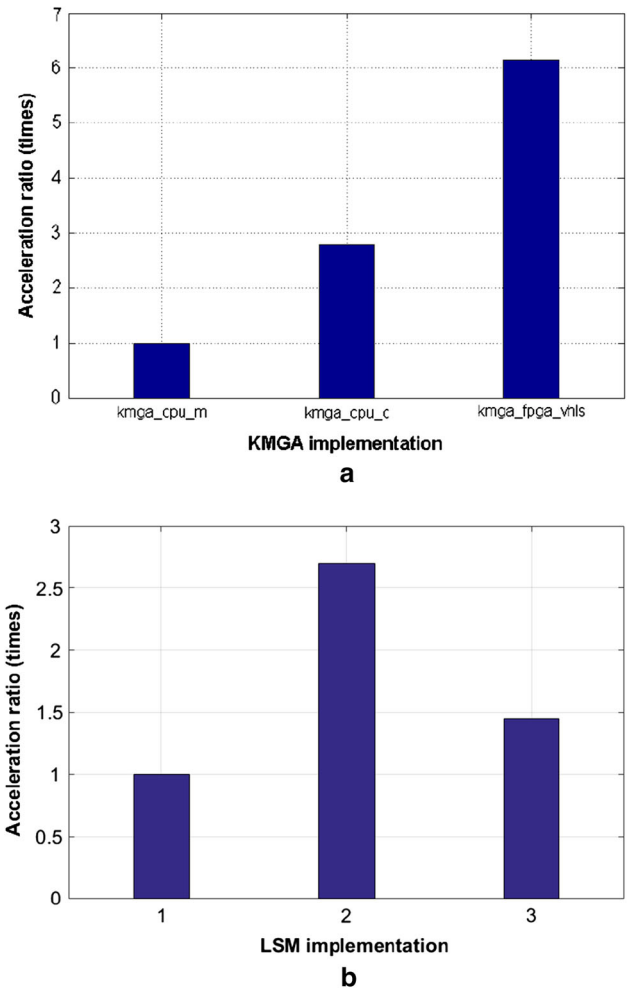


Fig. 14 Acceleration ratio: **a** KMGA, **b** LSM

Table 6 Comparison of hardware consumption

Algorithm	Implementation	Device	BRAM_18K	DSP48E	FF	LUT
KMGA	kmga_fpga_vhls	Virtex-7	14	116	19475	32245
	FPGA-KMGA [32]	Virtex7	192	2352	467264	668784
LSM	lsm_fpga_vhls	Kintex-7	3	74	14321	23854
	Chao et al. [31]	Kintex-7	3	74	13462	23403

automatically in theory within VHLS. But in practical, due to the built-in function problem of MATLAB, we still spend several days to establish function libraries. Meanwhile, since the machine-generated code is usually much harder to read than the manually written one, VHLS costs more time than manual design flow on the code debug and verification. In the two cases of this paper, the proposed VHLS process raises the development productivity by around 77% and 84.6%, respectively.

Meanwhile, it should be noted that for the VHLS prototype of this paper, the practical productivity gain changes with the algorithm complexity. For some classical and simple benchmarks, such as matrix production, Sobel operator, RGB-to-gray level transformation or log transformation, since these algorithms are very simple, the generated code, such as the one shown in Fig. 8, can be directly used into the C-synthesis process without building additional function library or debugging the intermediate code. Under these circumstances, VHLS design flow can get the highest productivity with a completely automatized MATLAB-to-RTL conversion.

5 Discussion and conclusion

This paper proposes a VHLS method for image processing designs by combining the recent source-to-source compilation and HLS techniques. It provides a high abstraction level environment to the users, which can greatly benefit the development productivity by automating the MATLAB-to-RTL synthesis process. We prototyped the proposed approach by using currently available EDA tools, and then verified it within two real-life applications. Experiments demonstrate that it can effectively give play to the advantages of FPGA related to the devices of other types in the same abstraction level. Furthermore, it will not increase the complexity of the algorithm behaviors described using MATLAB in routine level, even if they are not specially developed for FPGA.

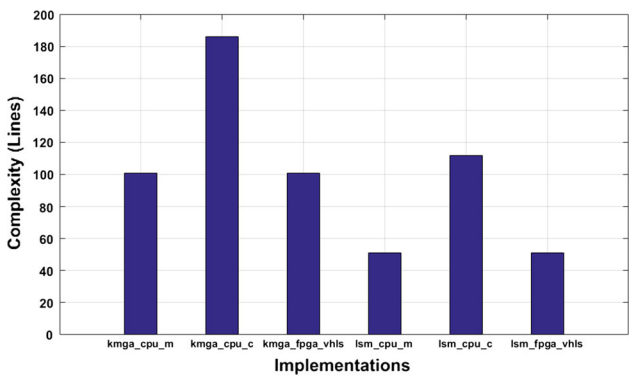


Fig. 15 Complexity comparison

So this method can effectively facilitate the transplantation between the different platforms, such as CPU versus FPGA.

Meanwhile, it should be noted that VHLS constitutionally provides a vector-oriented programming environment. Up to our knowledge, it does not yet exist a mature MATLAB-to-RTL convertor with the capacity of this type. For example, matrix variables are not supported by MATLAB HDL Coder. Further more, since the VHLS-based design flow is a heterogeneous development framework, it can be potentially optimized by replacing the EDA tools proposed in this paper by using other candidates, such as Catapult C or Cyber Workbench. This characteristic enables VHLS to benefit from the future progress of either MATLAB-to-C transcompiler or C-synthesis techniques easily.

On the other hand, some new issues and challenges are found as well. The first is the significant efficiency performance gap between the manual implementations and the proposed method. According to the practices in the both of software and hardware aspects, despite high development productivity and maintainability, higher abstraction level is bound to result in lower efficiency performance. For example, in the cases of Rupnow et al. [44] and Liang et al. [35], the performance difference between the HLS-generated and manual RTL designs is up to 40× for a high-definition stereo matching implementation. In the case of KMGA, the gap of running time performance due to this reason can be estimated as around 4× by comparing to its AutoPilot (Vivado_HLS)-based implementation [33] (118.04 versus 29.48 ms/pixel).

Secondly, many image processing algorithms have a lot of parallel processing code, e.g., the code with loops optimized by using *parfor* in MATLAB or *OpenMP* in C, whose fundamentals are also applicable to the proposed method. Making VHLS to be capable of performing variant optimization forms into the generated implementations can greatly improve the quality of the design and provide a much larger design exploration space.

Fortunately, currently available HLS tools have made a large number of optimization forms available to the C-synthesis, such as function inline, loop unroll, loop pipeline. Moreover, these optimizations can be conveniently made according to the directive configuration [59, 60]. Consequently, a potential solution for the two challenges above is to map the optimization forms applied in the source code into the corresponding optimization directives during the MATLAB-to-C conversion.

Finally, it is also found that manual code transformation is still needed sometimes due to the incompatibility between the selected source-to-source compiler and HLS tool. But in our opinion, this issue should not be considered hard to fix, because MATLAB-to-C conversion has been a

quite mature technique in the software programming engineering, taking the constraints of HLS-available C into the compile process can be easily realized through the currently available techniques.

For the future work, we plan to focus it on the improvement of the proposed VHLS method. Some source-to-source compile-based optimization strategies may be developed to improve the quality of the generated RTL. Meanwhile, we will further verify it within real FPGA-based synthesis process.

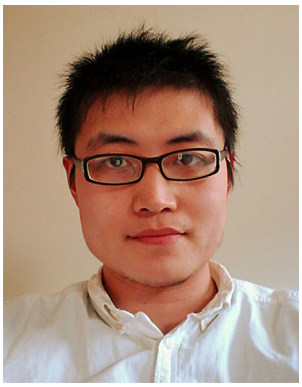
Acknowledgements The authors would like to thank the China Scholarship Council and the Conseil Régional de Bourgogne Franche-Comté for their funding of our studies.

References

1. Adsc research highlights: synthesize hardware, without hardware expertise. Online (2016). https://adsc.illinois.edu/research/adsc-research-highlights/adsc-research-highlights-synthesize-hardware-without-hardware-expe
2. Alle, M., Morvan, A., Derrien, S.: Runtime dependency analysis for loop pipelining in high-level synthesis. In: Design Automation Conference (DAC), 2013 50th ACM/EDAC/IEEE, pp. 1–10 (2013)
3. Andin, J.M., Arenaz, M., Rodrguez, G., Tourio, J.: A novel compiler support for automatic parallelization on multicore systems. Parallel Comput. **39**(9), 442–460 (2013). doi:10.1016/j.parco.2013.04.003
4. Armstrong, B., Kim, S., Park, I., Voss, M., Eigenmann, R.: Compiler-based tools for analyzing parallel programs. Parallel Comput. **24**(3–4), 401–420 (1998)
5. Baklouti, M., Aydi, Y., Marquet, P., Dekeyser, J., Abid, M.: Scalable mpnoc for massively parallel systems—design and implementation on FPGA. J. Syst. Archit. **56**(7), 278 – 292 (2010). doi:10.1016/j.sysarc.2010.04.001. Special Issue on HW/SW Co-Design: Systems and Networks on Chip
6. Balla-Arabe, S., Gao, X., Wang, B., Yang, F., Brost, V.: Multi-kernel implicit curve evolution for selected texture region segmentation in VHR satellite images. Geosci. Remote Sens. IEEE Trans. **52**(8), 5183–5192 (2014). doi:10.1109/TGRS.2013.2287239
7. Barney, B.: Introduction to parallel computing. Article published online. https://computing.llnl.gov/tutorials/parallel_comp/
8. Bi, Y., Li, C., Yang, F.: Very high level synthesis for image processing applications. In: 10th International Conference on Distributed Smart Cameras (ICDSC 2016), Paris France (2016)
9. Cadence Design Systems, Inc: C-to-Silicon Compiler High-Level Synthesis (2011). https://www.cadence.com/rl/Resources/datasheets/C2Silicon_ds.pdf
10. Colodro-Conde, C., Toledo-Moreo, F., Toledo-Moreo, R., Martnez-lvarez, J., Garrigs-Guerrero, J., Ferrndez-Vicente, J.: A practical evaluation of the performance of the impulse codeveloper hls tool for implementing large-kernel 2-d filters. J. Real-Time Image Proc. **9**(1), 263–279 (2014). doi:10.1007/s11554-013-0374-x
11. Colodro-Conde, C., Toledo-Moreo, F.J., Toledo-Moreo, R., Martínez-Álvarez, J.J., Guerrero, J.G., Ferrández-Vicente, J.M.: Evaluation of stereo correspondence algorithms and their implementation on FPGA. J. Syst. Archit. **60**(1), 22–31 (2014). doi:10.1016/j.sysarc.2013.11.006
12. Cong, J., Fan, Y., Han, G., Jiang, W., Zhang, Z.: Behavior and communication co-optimization for systems with sequential communication media. In: Design Automation Conference, 2006 43rd ACM/IEEE, pp. 675–678 (2006). doi:10.1109/DAC.2006.229314
13. Cong, J., Liu, B., Neuendorffer, S., Noguera, J., Vissers, K., Zhang, Z.: High-level synthesis for FPGAs: from prototyping to deployment. Comput.-Aid. Design of Integr. Circuits Syst. IEEE Trans. **30**(4), 473–491 (2011). doi:10.1109/TCAD.2011.2110592
14. Cong, J., Liu, B., Prabhakar, R., Zhang, P.: A study on the impact of compiler optimizations on high-level synthesis. In: Kasahara, H., Kimura, K. (eds.) Languages and Compilers for Parallel Computing, Lecture Notes in Computer Science, vol. 7760, pp. 143–157. Springer, Berlin, Heidelberg (2013). doi:10.1007/978-3-642-37658-0_10
15. Coussy, P., Morawiec, A.: High-Level Synthesis: From Algorithm to Digital Circuit, 1st edn. Springer, Berlin, Incorporated (2008)
16. Daniel D., G., Nikil D., D., Allen C-H, W., Steve Y-L, L.: High-Level Synthesis: Introduction to Chip and System Design, 1st edn. Springer, New York (1992). doi:10.1007/978-1-4615-3636-9.
17. Davoodi, A., Srivastava, A.: Effective techniques for the generalized low-power binding problem. ACM Trans. Des. Autom. Electron. Syst. **11**(1), 52–69 (2006). doi:10.1145/1124713.1124718
18. Deming, C., Eric, L., Kyle, R., Zheng, C.: Hardware synthesis without hardware expertise. Tech. rep., Advanced Digital Sciences Center (ADSC) of the University of Illinois at Urbana-Champaign (2011)
19. Fowers, J., Brown, G., Cooke, P., Stitt, G.: A performance and energy comparison of FPGAs, GPUs, and multicores for sliding-window applications. In: Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays, FPGA '12, pp. 47–56. ACM, New York, NY, USA (2012). doi:10.1145/2145694.2145704
20. Gebotys, C., Elmasry, M.: Vlsi design synthesis with testability. In: Design Automation Conference, 1988. Proceedings, 25th ACM/IEEE, pp. 16–21 (1988). doi:10.1109/DAC.1988.14728
21. Girkar, M., Polychronopoulos, C.: Automatic extraction of functional parallelism from ordinary programs. Parallel Distrib. Syst. IEEE Trans. **3**(2), 166–178 (1992). doi:10.1109/71.127258
22. González, C., Sánchez, S., Paz, A., Resano, J., Mozos, D., Plaza, A.: Use of FPGA or GPU-based architectures for remotely sensed hyperspectral image processing. Integr. VLSI J. **46**(2), 89–103 (2013). doi:10.1016/j.vlsi.2012.04.002
23. Hafer, L., Parker, A.: Register-transfer level digital design automation: The allocation process. In: Design Automation, 1978. 15th Conference on, pp. 213–219 (1978). doi:10.1109/DAC.1978.1585172
24. Heijligers, M., Cluitmans, L., Jess, J.: High-level synthesis scheduling and allocation using genetic algorithms. In: Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages. IFIP International Conference on Very Large Scal, pp. 61–66 (1995). doi:10.1109/ASPDAC.1995.486203
25. Heijligers, M., Jess, J.: High-level synthesis scheduling and allocation using genetic algorithms based on constructive topological scheduling techniques. In: Evolutionary Computation, 1995. IEEE International Conference on, vol. 1, p. 56 (1995). doi:10.1109/ICEC.1995.489119
26. Intel Corporation: Intel® C++ Compiler User and Reference Guides, 304968-022us edn. (2008).http://www.physics.udel.edu/bnikolic/QTTG/shared/docs/intel_c_user_and_reference_guide.pdf

27. Jiang, J., Liu, C., Ling, S.: An FPGA implementation for real-time edge detection. J. Real-Time Image Process. (2015). doi:10.1007/s11554-015-0521-7
28. Jolivot, R., Benezeth, Y., Marzani, F.: Skin parameter map retrieval from a dedicated multispectral imaging system applied to dermatology/cosmetology. Int. J. Biomed. Imaging **2013**, 15 (2013). doi:10.1155/2013/978289
29. Kestur, S., Davis, J., Williams, O.: Blas comparison on FPGA, CPU and GPU. In: VLSI (ISVLSI), 2010 IEEE Computer Society Annual Symposium on, pp. 288–293 (2010). doi:10.1109/ISVLSI.2010.84
30. Lee, J.H., Hsu, Y.C., Lin, Y.L.: 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, pp. 20–23 (1989). doi:10.1109/ICCAD.1989.76896
31. Li, C., Balla-Arabé, S., Ginhac, D., Yang, F.: Embedded implementation of VHR satellite image segmentation. Sensors **16**(6), 771 (2016). doi:10.3390/s16060771. http://www.mdpi.com/1424-8220/16/6/771
32. Li, C., Balla-Arabé, S., Yang, F.: Embedded multispectral image processing for real-time medical application. J. Syst. Archit. (2015). doi:10.1016/j.sysarc.2015.12.002. http://www.sciencedirect.com/science/article/pii/S1383762115001526
33. Li, C., Brost, V., Benezeth, Y., Marzani, F., Yang, F.: Design and evaluation of a parallel and optimized light-tissue interaction-based method for fast skin lesion assessment. J. Real-Time Image Process. (2015). doi:10.1007/s11554-015-0494-6
34. Li, P., Pouchet, L.N., Cong, J.: 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)
35. Liang, Y., Rupnow, K., Li, Y., Min, D., Do, M.N., Chen, D.: High-level synthesis: productivity, performance, and software constraints. J. Electr. Comput. Eng. (2012). Article ID 649057. doi:10.1155/2012/649057
36. Lyberis, S., Kalokerinos, G., Lygerakis, M., Papaefstathiou, V., Mavroidis, I., Katevenis, M., Pnevmatikatos, D., Nikolopoulos, D.S.: Fpga prototyping of emerging manycore architectures for parallel programming research using formic boards. J. Syst. Archit. **60**(6), 481–493 (2014). doi:10.1016/j.sysarc.2014.03.002
37. Mansouri, A., Marzani, F., Gouton, P.: Neural networks in two cascade algorithms for spectral reflectance reconstruction. In: ICIP (2), pp. 718–721. IEEE (2005)
38. Marwedel, P.: A new synthesis algorithm for the mimola software system. In: Design Automation, 1986. 23rd Conference on, pp. 271–277 (1986). doi:10.1109/DAC.1986.1586100
39. Meeus, W., Van Beeck, K., Goedemé, T., Meel, J., Stroobandt, D.: An overview of today's high-level synthesis tools. Des. Autom. Embed. Syst. **16**(3), 31–51 (2012). doi:10.1007/s10617-012-9096-8
40. Musavi, S., Chowdhry, B., Kumar, T., Pandey, B., Kumar, W.: Iots enable active contour modeling based energy efficient and thermal aware object tracking on fpga. Wirel. Pers. Commun. **85**(2), 529–543 (2015). doi:10.1007/s11277-015-2753-z
41. Paulin, P., Knight, J.: Scheduling and binding algorithms for high-level synthesis. In: Design Automation, 1989. 26th Conference on, pp. 1–6 (1989). doi:10.1109/DAC.1989.203360
42. Prost-Boucle, A., Muller, O., Rousseau, F.: Fast and standalone design space exploration for high-level synthesis under resource constraints. J. Syst. Archit. **60**(1), 79–93 (2014). doi:10.1016/j.sysarc.2013.10.002
43. Rodrigues, R., Cardoso, J., Diniz, P.: 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, pp. 219–228 (2007). doi:10.1109/FCCM.2007.16
44. Rupnow, K., Liang, Y., Li, Y., Min, D., Do, M., Chen, D.: High level synthesis of stereo matching: productivity, performance, and software constraints. In: Field-Programmable Technology (FPT), 2011 International Conference on IEEE (2011)
45. Senturk, A., Gok, M.: Sequential large multipliers on FPGAs. J. Signal Process. Syst. **81**(2), 137–152 (2015). doi:10.1007/s11265-014-0912-1
46. Sidiropoulos, H., Siozios, K., Soudris, D.: A novel 3-d FPGA architecture targeting communication intensive applications. J. Syst. Archit. **60**(1), 32–39 (2014). doi:10.1016/j.sysarc.2013.09.012
47. Sugimoto, N., Miyajima, T., Kuhara, T., Katuta, Y., Mitsuichi, T., Amano, H.: Artificial intelligence of blokus duo on FPGA using cyber work bench. In: Field-Programmable Technology (FPT), 2013 International Conference on, pp. 498–501 (2013). doi:10.1109/FPT.2013.6718427
48. Sukhwani, B., Thoennes, M., Min, H., Dube, P., Brezzo, B., Asaad, S., Dillenberger, D.: A hardware/software approach for database query acceleration with FPGAs. Int. J. Parallel Prog. **43**(6), 1129–1159 (2015). doi:10.1007/s10766-014-0327-4
49. Sumit, G., Rajesh, G., Nikil D., D., Alexandru, N.: SPARK: A Parallelizing Approach to the High-Level Synthesis of Digital Circuits. Springer, New York (2004). doi:10.1007/b117058
50. Synopsys, Inc.: Synphony C Compiler (2010). http://www.scanru.ru/file_link.php?fid=831
51. Toledo-Moreo, F.J., Martínez-Álvarez, J.J., Garrigós-Guerrero, J., Ferrández-Vicente, J.M.: FPGA-based architecture for the real-time computation of 2-d convolution with large kernel size. J. Syst. Archit. **58**(8), 277–285 (2012). doi:10.1016/j.sysarc.2012.06.002
52. Tseng, C.J., Siewiorek, D.: Automated synthesis of data paths in digital systems. Comput.-Aid. Des. Integr. Circuits Syst. IEEE Trans. **5**(3), 379–395 (1986). doi:10.1109/TCAD.1986.1270207
53. Vega-Rodríguez, M.A.: Methodologies and tools for the design space exploration of embedded systems. J. Syst. Archit. **60**(1), 53–54 (2014). doi:10.1016/j.sysarc.2013.12.001
54. Villarreal, J., Park, A., Najjar, W., Halstead, R.: Designing modular hardware accelerators in c with roccc 2.0. In: Field-Programmable Custom Computing Machines (FCCM), 2010 18th IEEE Annual International Symposium on, pp. 127–134 (2010). doi:10.1109/FCCM.2010.28
55. Wakabayashi, K.: C-based behavioral synthesis and verification analysis on industrial design examples. In: Proceedings of the 2004 Asia and South Pacific Design Automation Conference. ASP-DAC '04, pp. 344–348. IEEE Press, Piscataway, NJ, USA (2004)
56. Wakabayashi, K.: Cyberworkbench: integrated design environment based on c-based behavior synthesis and verification. In: VLSI Design, Automation and Test, 2005 (VLSI-TSA-DAT). 2005 IEEE VLSI-TSA International Symposium on, pp. 173–176 (2005). doi:10.1109/VDAT.2005.1500048
57. Wang, G.: Catapult C Synthesis Work Flow Tutorial. ECE Department, Rice University, version 1.3 edn. (2010)
58. Xilinx: System Generator for DSP—Getting Started Guide. Xilinx, ug639 (v 14.3) edn. (2012). http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/sysgen_gs.pdf
59. XILINX: Vivado Design Suite User Guide, ug902(2012.2) edn. (2012)
60. Xilinx: Introduction to FPGA design with vivado high-level synthesis. Tech. Rep. UG998 (v1.0), Xilinx (2013)
61. Yuki, T., Morvan, A., Derrien, S.: Derivation of efficient fsm from loop nests. In: Field-Programmable Technology (FPT), 2013 International Conference on, pp. 286–293 (2013). doi:10.1109/FPT.2013.6718367

62. Zhang, Z., Fan, Y., Jiang, W., Han, G., Yang, C., Cong, J.: Autopilot: a platform-based esl synthesis system. In: Coussy, P., Morawiec, A. (eds.) High-Level Synthesis, pp. 99–112. Springer, Dordrecht (2008). doi:10.1007/978-1-4020-8588-8_6
63. Ziegler, H., Hall, M.W., Diniz, P.: Compiler-generated communication for pipelined fpga applications. In: Design Automation Conference, 2003. Proceedings, pp. 610–615 (2003). doi:10.1109/DAC.2003.1219091
64. Zou, D., Dou, Y., Xia, F.: Optimization schemes and performance evaluation of smith-waterman algorithm on cpu, gpu and fpga. Concurr. Comput. Pract. Exper. **24**(14), 1625–1644 (2012). doi:10.1002/cpe.1913



Chao Li received the M.S. in Electronic Engineering from the University of Burgundy in 2012, the M.S. in Computer Science from the University of Paris Sud in 2013 and the Ph.D. in Computer Science from the University of Burgundy in 2016. He is currently a research engineer of LE2I CNRS-UMR, Laboratory of Electronic, Computing and Imaging Sciences at the University of Burgundy. His research interests are in the areas of SW/HW Co-design, High-Performance Computer and Digital Signal Processing.

Yanjing Bi received the M.S. in Language Sciences from the University of Burgundy, France, in 2013. She is currently a Ph.D. candidate with the CPTC (Centre Pluridisciplinaire Textes et Cultures) at the University of Burgundy. Her research interests are in the areas of computational and programming linguistics, comparative linguistics, translation, language pedagogy and the phraseologies.

Franck Marzani received his M.Sc. in computer science from the University of Rennes, France, in 1989. He obtained his Ph.D. in computer vision and image processing from the University of Burgundy, Dijon, France, in 1998. His PhD dissertation dealt with 3D acquisition and processing of human motion. Since then he has worked in the LE2I laboratory (Laboratoire d'Electronique, d'Informatique et Image), which is a CNRS affiliated institute of research (UMR CNRS 6306) at the University of Burgundy. He received his "Habilitation à Diriger les Recherches" in 2007, and he is a full professor since 2009.

Fan Yang received the B.S. degree in electrical engineering from the University of Lanzhou, China, in 1982 and the M.S. (D.E.A.) (computer science) and Ph.D. degrees (image processing) from the University of Burgundy, France, in 1994 and 1998, respectively. She is currently a full professor and member of LE2I CNRS-UMR, Laboratory of Electronic, Computing and Imaging Sciences at the University of Burgundy, France. Her research interests are in the areas of patterns recognition, neural network, motion estimation based on spatiotemporal Gabor filters, parallelism and real-time implementation and, more specifically, automatic face image processing algorithms and architectures. Pr. Yang is member of the French research group ISIS (Information, Signal, Images and Vision), she livens up the theme C: Algorithm Architecture Adequation.