A Unified Backend for Targeting FPGAs from DSLs

Emanuele Del Sozzo[1], Riyadh Baghdadi[2], Saman Amarasinghe[2], Marco D. Santambrogio[1]
[1]DEIB - Politecnico di Milano, Italy
{emanuele.delsozzo, marco.santambrogio}@polimi.it
[2]CSAIL - Massachusetts Institute of Technology, USA
{baghdadi, saman}@csail.mit.edu

***Abstract*—The major flaw of Field Programmable Gate Arrays (FPGAs) is their hard programmability and steep learning curve. Even though High-Level Synthesis (HLS) tools may alleviate this task by providing directives to optimize the hardware design, as well as supporting languages like C/C++ and OpenCL, the development of efficient designs for FPGA is still a challenging and time-consuming task. In this context, Domain Specific Languages (DSLs) represent an emerging solution to generate efficient code to target FPGAs. However, the support for these languages towards FPGA is still limited, and only few DSLs provide FPGA backends. This paper describes *FROST*, a unified backend for targeting FPGAs from DSLs. FROST takes as input an algorithm described in one of the supported DSLs and generates an optimized design suitable for HLS tools. To this end, FROST exposes a high-level scheduling co-language to drive many aspects of the optimization process, like the resulting architecture, the level of parallelism, and so on. We evaluated FROST on a set of image processing kernels, developed in Halide and TIRAMISU, and compared the results against a hand-tuned FPGA library. The experimental results demonstrate that FROST designs are able to match the performance of such library (exploiting the same level of parallelism), and surpass it by a factor of 10X when combining FROST and the frontends scheduling commands.**

***Index Terms*—FPGA, DSL, Common Backend, Scheduling Co-Language**

I. Introduction

In the recent years, we have been experiencing an increasing interest in systems composed by multiple heterogeneous architectures. Such systems permit to overcome the limits of homogeneous architectures [1] and thus improve performance while reducing power consumption. For this reason, many high performance systems [2] are currently combining Central Processing Units (CPUs) with architectures like Graphic Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs) to accelerate computations belonging to different fields (like image and signal processing, linear algebra, computational biology, etc.) on the most suitable device for that domain.

Concurrently, many and different tools are emerging in literature to ease and abstract the design of highly parallel applications for such architectures [3], [4]. One of the most interesting solutions in this context is represented by Domain Specific Languages (DSLs). Indeed, current DSLs [5]–[7] allow the user to quickly and easily develop portable designs for multiple architectures (mainly CPUs and GPUs). Thanks to the restriction of the domain, DSL compilers are able rapidly explore the design space and deeply optimize the resulting implementations. As a result, DSL applications often outperform hand-tuned libraries.

Among the aforementioned architectures, FPGAs currently lack a concrete support for DSLs. Historically, hardware design for FPGAs has always been more complex with respect to the design for CPUs and GPUs, in spite of the great design possibilities FPGAs can provide (for instance, in terms of arbitrary data precision and custom architecture tailored to the target application). Even though in the last years there has been a significant improvement in the toolchain for FPGAs, like High-Level Synthesis (HLS) tools [8] that permit to hardware accelerate algorithms using C/C++ and OpenCL instead of Hardware Description Languages like Verilog and VHDL, the design process remains complex and the supported languages are still limited. As a consequence, there is little to no support for DSLs, and, even though there exist some DSLs able to target FPGAs [9]–[13], a common solution to target FPGAs from DSLs is still lacking.

This paper describes *FROST*, a unified backend to efficiently hardware accelerate DSLs on FPGAs. Starting from the an algorithm described in one of the supported DSLs, FROST translates it into its Intermediate Representation (IR), performs a series of FPGA-oriented optimizations steps, and, finally, generates an optimized design suitable of HLS tools. In order to better leverage the features of the FPGA and enhance the performance, FROST provides a high-level scheduling co-language the user can exploit to guide the optimizations to apply, as well as specify the architecture to implement. This allows to easily evaluate different hardware designs and choose the most suitable to the input algorithm.

This paper, starting from the work described in [14], provides the following contributions:

- A common backend capable of supporting multiple DSLs as frontend. In the context of this paper, we show FROST integration with Halide and TIRAMISU.
- Introduction of a new critical scheduling command able to generate a *streaming* dataflow architecture, mainly suitable for applications like image processing kernels.
- The FPGA designs generated by FROST match the performance of a hand-tuned HLS library [15], and, thanks to a combination of both FROST and TIRAMISU scheduling commands, they significant outperform such library (by a factor of 10X).

The paper is organized as follows: Section II describes the related work, while Section III exhaustively analyzes FROST

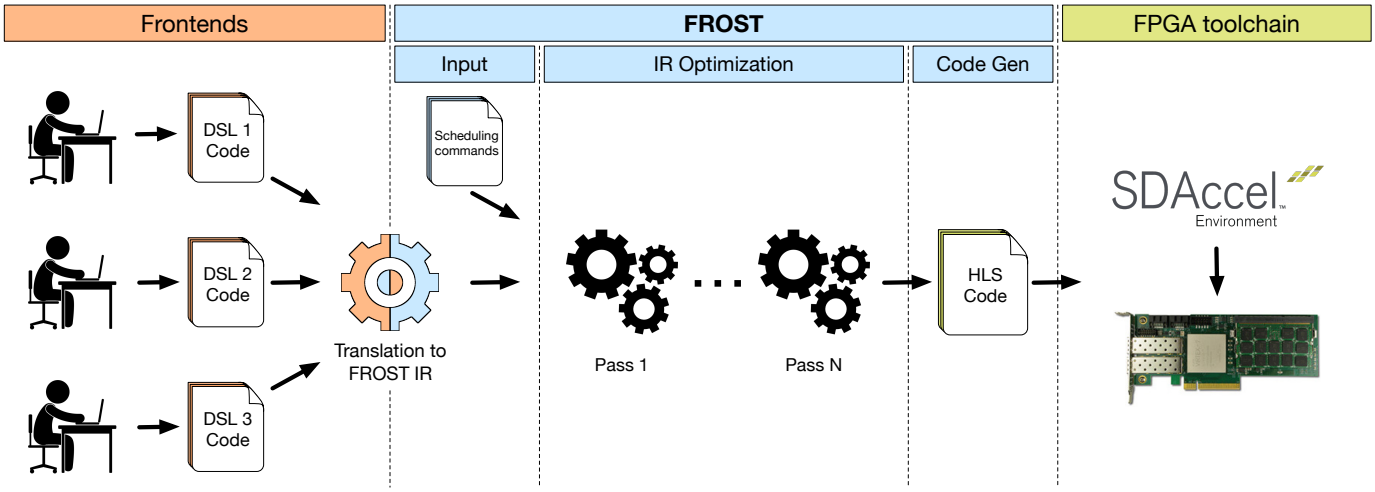

Fig. 1. FROST design flow

architecture, as well as its high-level scheduling co-language. In Section IV we report FROST experimental evaluation. Finally, Section V draws the conclusions and gives some insights on the future work.

II. Related Work

In the state of the art there are many and different tools whose purpose is to facilitate the hardware acceleration on FPGA of algorithms. HLS tools are examples of that. For instance, frameworks like Xilinx Vivado HLS [8], Intel HLS Compiler [16] and SDK for OpenCL [17] allow users to produce a RTL representation of a high level code, usually written in languages like C/C++ and OpenCL. HLS tools, like the aforementioned, support all computational domains, and feature a set of directives to guide the optimizations to apply to the resulting hardware design, as well as exhaustive reports describing the details of the design (e.g., resource usage, pipeline depths, etc.).

On the other hand, many frameworks and compilers focusing on specific contexts to generate efficient hardware implementations are emerging in literature. Darkroom [9] is a language and compiler embedded in Terra language [18] for image processing. Darkroom compiler takes as input a high level description of the application and translates it into line-buffered pipelines, expressed in Verilog HDL. Darkroom then synthesizes such pipelines for ASIC, FPGA, or CPU. The experimental evaluation of Darkroom reports gigapixel/sec performance for ASIC designs, while realtime 1080p/60 video processing for FPGAs ones. In [10], the authors present RIPL, a memory-efficient, declarative FPGA image processing DSL. At first, RIPL compiles the input programs into dataflow graphs, then it relies on an open source dataflow compiler [19] to generate the HDL. The authors evaluated RIPL against five benchmarks, and, without the need of synthesis directives, they showed a comparable memory usage with respect to the Vivado HLS Video Library [15]. The work in [11] presents an FPGA backend for the PolyMage DSL [20]. At first, the proposed backend enforces optimizations in terms of both data parallelism and memory bandwidth, then it leverages Vivado HLS to produce the FPGA design. In the experimental evaluation, the authors compared their backend against both Darkroom, and Vivado HLS Video Library. On average, this work reaches a $1.5\times$ speedup. ExaSlang 4 [12] is a DSL designed for the hardware acceleration on FPGA of numerical solvers based on the multigrid method. ExaSlang 4 takes advantage of Vivado HLS as backend to generate the hardware implementation of the input code. The presented approach outperformed a vectorized, single-threaded execution on an Intel i7 by a 3X factor. In [13], the authors describe an extension to Halide [5] to accelerate image processing kernels on Xilinx Zynq MPSoCs. To this end, the authors provided Halide with additional scheduling commands to control some crucial aspects of the resulting FPGA designs, like the depth of the FIFOs between kernels. Polyhedral compilers such as Rose [21] and PENCIL [6] use fully automatic techniques (such as the Pluto [22] scheduling algorithm) to parallelize and optimize computations and generate an OpenCL or HLS code that targets FPGA architectures.

With respect to the aforementioned work available in literature, FROST is designed as a common backend for multiple DSLs, instead of being specific to one DSL, and thus reduces the effort of developing a new FPGA backend. It is also designed to support data parallel algorithms implemented as loops and operating on dense arrays and tensors. One of the fundamental features of FROST is its high level scheduling co-language, which allows the user to specify exactly how the computation should be optimized and mapped to FPGA. Thanks to this feature, FROST is capable of being generic enough and to support general data parallel algorithms, while still reaching a high level of performance.

III. Frost

The purpose of this Section is to describe the key features and rationale of this work. FROST is a common backend for

the acceleration of FPGA for DSLs. Given a description of an algorithm in one of the supported DSLs, FROST translates it into FROST IR, and then manipulates and optimized the IR. To this end, FROST provides a high level *scheduling co-language* to allow users to specify the optimizations to apply at different levels (e.g., computation, memory interface, and so on). Once the optimization process is done, FROST generates a C++ code suitable for HLS tools. The final step consists in generating the FPGA bitstream from FROST output. Figure 1 gives an overview of FROST design flow.

The following Sections analyze in deep the overall architecture of FROST framework. In particular, in Section III-B we first start describing the DSLs supported as frontends, as well as the rationale behind the choice of designing FROST as a common backend. Section III-C describes the motivation for the scheduling co-language, and analyzes the commands that the user can express with it. Section III-D focuses on the IR manipulation and optimization process. More specifically, we explain how the IR changes according to the scheduling commands. Finally, in Section III-E we report the FPGA bitstream generation step, which starts from FROST output.

A. Scope

FROST is designed for expressing data parallel algorithms, in particular algorithms that operate over dense arrays using loop nests. These algorithms are often found in the areas of image processing, dense linear algebra and tensor algebra, stencil computations, and deep neural networks. Moreover, FROST is designed as a common backend for DSLs only. We restrict ourselves to DSLs because DSLs are very effective in producing efficient code for a given target architecture (CPU, GPU, FPGA) because they are restricted to a small set of language features and because their context is limited. Indeed, domain restrictions allow better exploration of the design space and better identification of the computational patterns that are typical in a specific domain. Language restrictions allow better static analysis for the code. For example, many DSLs do not have pointers which allows better static analysis (it is known that static analysis is undecidable if the language has double pointer indirection [23]). As a result, DSLs enable users to easily reach significant performance with a relative small effort with respect to other more general programming languages.

B. Frontends

Recently, the use of DSLs and high level languages has been gaining in popularity for many reasons: (1) they provide portability across multiple hardware architectures; (2) they provide high productivity, and (3) they allow the application of certain optimizations such as fusion, and data layout transformations that are difficult otherwise. The input of the FROST backend is the FROST IR which describes the algorithm and a list of optimizations (scheduling) commands to optimize the algorithm. Currently, the FROST IR is fully compatible with Halide [5], a state-of-the-art DSL and compiler for image processing pipelines, as well as TIRAMISU [7], a unified optimization framework for DSL compilers, and which presently is integrated in DSLs such as Julia [24] and Theano [25]. In this paper, we will focus on presenting FROST itself, and on evaluating FROST with Halide and TIRAMISU as frontends. A full end-to-end evaluation of FROST with many high level DSLs, like the ones that use TIRAMISU as an optimization framework (Theano and Julia), is left for a future paper. Figure 2 shows the complete FROST stack.

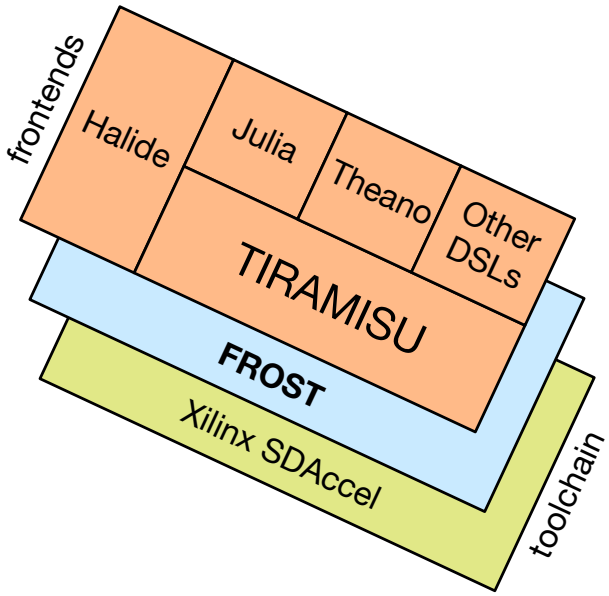

Fig. 2. FROST stack

C. Scheduling Co-Language

We decided to provide FROST with a high scheduling co-language for different reasons. In many performance critical domains, users need code that achieves performance comparable to hand-optimized code. Generating such code requires combinations of non-trivial program transformations that optimization frameworks try to fully automate using cost models, heuristics [26], and machine learning [27]. While automatic optimization techniques provide productivity, they may not always achieve the desired level of performance. A scheduling co-language allows to solve this problem by separating mechanism from policy [1]. This way, FROST allows full control over scheduling while still enabling integration with higher level frameworks for policy-making.

Another reason for the scheduling co-language is the fact that designing efficient architectures for FPGA is definitely a challenging task, and it gets more complex when the goal is to generate them automatically. Again, a set of well defined guidelines for the development of the FPGA design is a viable solution, but it would prevent a fully exploitation of the FPGA features. On the other hand, an exhaustive design space exploration can for sure identify a highly efficient FPGA

[1]Mechanism means the application of optimizations while policy means deciding which optimization to apply

design, but it would require a significant amount of time. Besides, not even HLS tools do that automatically, but rather they provide the designer with a set of directives to enhance the performance of the hardware design, even though most of the work is still up to the designer's expertise and skills with FPGA design. As a result, producing and evaluating different hardware designs may quickly become a time-consuming and error-prone task.

For the aforementioned reasons, the goal of FROST scheduling co-language is to (1) separate mechanism from policy and enable users to fully control the generated code; (2) provide the user with a set of possible optimizations in order to tailor the resulting architecture to the input computation, and (3) simplify optimization space exploration. Given the scheduling commands, FROST will automatically generate an optimized version of the input code.

FROST scheduling commands may refer to different aspects of the resulting hardware design. Indeed, they can specify the scheduling of parts of the computation (e.g., loop scheduling), how the data need to be stored on the FPGA memory (either logic or BRAM), or the design of the overall architecture. To this end, we organized the scheduling commands in three different categories (Table I reports a summary of the scheduling commands currently supported by FROST).

Computation commands: This category contains the scheduling commands that allow to change the scheduling of part of the computations within the overall hardware architecture. In particular, these commands mainly involve the scheduling of loop statements. Indeed, given a function within the design that iterates over the dimensions of the input/output buffers, the user can mark one or more dimensions to be pipelined, unrolled, or (un)flattened. Moreover, the user can choose to vectorize one or more input/output buffers in chunks of `n` bits. This commands has an impact on both the access to the off-chip memory and the computation.

Local memory commands: This category refers to the scheduling commands related to the data storage on the FPGA. Indeed, given a buffer, such commands enable to partition one or more dimensions of the buffer in a `complete`, `cyclic` or `block` way. According to the command, the data are stored in one or multiple BRAMs, or in logic. In this way, it is possible access in the same clock cycle to different elements of a buffer.

Architecture commands: This last category of scheduling commands impacts on the overall architecture to be generated by FROST. In particular, it defines whether to generate a `tiled` or `streaming` dataflow architecture. In Section III-D we will better describe the difference between these two architectures. These commands also take care of the communication with the off-chip memory. Currently, FROST provides support for a master/slave communication based on *AMBA AXI4* interface protocol [28]. This protocol allows to either stream or move a tile of data from the DDR to the FPGA local memory and vice-versa. According to the selected command, FROST employs a different approach to move data from/to the off-chip memory.

Finally, it is fundamental to notice that FROST scheduling co-language only focuses on transformations related to the FPGA implementation. Hence, FROST is not designed to perform transformations like loop splitting, loop tiling, and so on. Such transformations are surely useful and necessary in some cases (e.g. vectorization), but, since the supported DSLs, like Halide and TIRAMISU, already support them, there was no point in re-implement them also in FROST. Therefore, the idea behind this design choice is that FROST and its frontends has to work in synergy to produce an efficient hardware implementation.

TABLE I
SCHEDULING COMMANDS

We assume that `f` is a function, while `m` is the whole computation

Command	Description
Computation scheduling commands	
`f.pipeline(i)`	marks the dimension `i` to be pipelined
`f.unroll(i)`	marks the dimension `i` to be unrolled
`f.flatten(i)`	marks the dimension `i` to be (un)flattened
`f.vectorize(b, n)`	marks the buffer `b` to be vectorized in chunks on `n` bits
Local Memory scheduling commands	
`f.partition(b, d, t)`	marks the buffer `b` to be partitioned on dimension `d` in a `t` way
Architecture scheduling commands	
`m.tiled()`	marks the architecture to be implemented in a `tiled` way
`m.streaming()`	marks the architecture to be implemented in a `streaming` dataflow way

D. FROST workflow

The input of FROST framework is a set of functions described in one of the supported DSLs, as well as the scheduling commands to optimize the output hardware design. First of all, FROST translates the input functions into FROST IR by means of an ad-hoc translator for each of the frontend DSLs. As a result, FROST represents each function as a data structure that mainly contains: the name of the function itself, its arguments (which are either buffers or scalars), an Abstract Syntax Tree (AST) describing the body of the function, along with other minor parameters. FROST requires the dimensions of the input/output buffers to be specified at this stage of the workflow, otherwise it may not be possible to apply some of the optimizations. At this point, FROST starts applying a series of IR manipulation and optimization steps. In particular, the scheduling commands trigger some of the steps to perform. FROST enforces the scheduling commands in two different ways: IR manipulation or directives for HLS tools. Indeed, some of the commands have a direct map to HLS directives (like loop pipelining, unrolling, etc.), hence FROST inserts them during the code generation.

We now focus on the main steps FROST may perform according to the scheduling commands.

Top function generation: The first step of FROST is the definition of the top function, i.e. the main function to be synthesized on FPGA. The purpose of this function is to: (1) invoke the input functions, (2) instantiate the local

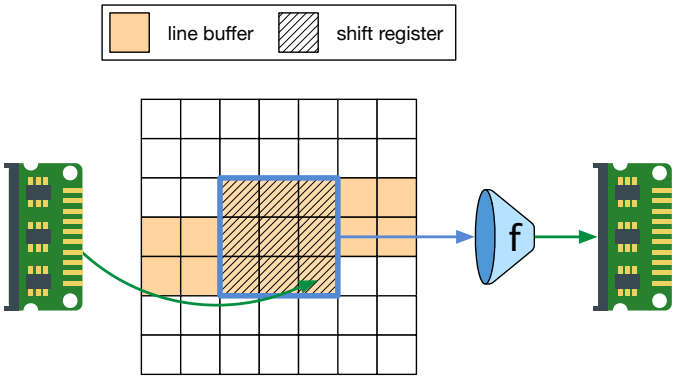

Fig. 3. Streaming architecture

buffers, (3) declare the memory interfaces, (4) manage the data transfer from/to the off-chip DDR memory. To this end, FROST analyzes the arguments of each function to differentiate the global arguments (i.e., the arguments that refer to buffers to be read/written from/to the off-chip memory) from temporary arguments (i.e. the arguments existing only within the top function). For instance, let us consider a pipeline of two image processing filters, namely `FilterA` and `FilterB`. The arguments/buffers of `FilterA` are `InA` and `OutA`, while the arguments/buffers of `FilterB` are `InB` and `OutB`. Since these two filters work as a pipeline, the output of `FilterA` is the input of `FilterB`, hence `InB` is `OutA`. As a result, `InA` and `OutB` are the global buffers, while `OutA`/`InB` is a temporary buffer. After identifying the global buffers, FROST inserts code blocks to read/write buffers from/to the off-chip memory before/after the computation. More technically, according to the chosen architecture command (namely `tiled` or `streaming`), FROST instantiates different read/write blocks, as well as buffer types.

Tiled architecture: In case of a `tiled` architecture, FROST instantiates the buffers as local arrays, and copies all the data from the off-chip memory, leveraging the memory burst, before starting the computation. Thanks to the data locality, each computation can access data within buffers at different offsets, and iterate multiple times on the same data (ideal for linear algebra kernels). In this case, `partition` scheduling commands may help to improve the performance enabling access to data at different offsets in the same clock cycle. Once the computation is over, the output data are copied back to the off-chip memory.

Streaming architecture: On the other hand, a `streaming` dataflow architecture (more suitable for applications like image processing filters) requires a more complex IR manipulation and analysis of the access patterns within each function. Indeed, to enable a dataflow computation and pipelining between the computations, FROST considers global and local buffers as streams of data (i.e., data FIFOs), and inserts the data coming from the off-chip memory inside such streams. According to the access pattern of the computations, FROST may instantiate line buffers and shift registers to store, respectively, lines of the input (typically an image) and the portion of data to be filtered. This allows to store on the FPGA memory only the data necessary to produce the output. At each clock cycle, the function reads a new element from the input stream and inserts it into the line buffers, while removing the oldest element from it. At the same time, the function loads data into the shift registers. In this way, as soon as enough data are available within such data structures, the function starts generating outputs and pushing them into the output streams. As a consequence, the functions overlap their execution, significantly reducing the latency of the hardware design. Figure 3 displays an overview of the `streaming` dataflow architecture.

Vectorization: Another optimization step that requires significant manipulation of FROST IR is vectorization. Using `vectorization` command, the user marks the buffer data to be packed in bunches of N bits. This command is available both for `tiled` and `streaming` architectures. For instance, a 512-bit vectorization of a 32-bit integer buffer packs 16 integers into a single variable. The vectorization allows to significantly reduce latency of both data transfer and computation itself, but, on the other hand, it implies a significant code restyling. At first, FROST update the data types of the buffers to be vectorized. Then, FROST has to update the access to the data bunches as well. Finally, similarly to the `streaming` architecture, FROST may need to insert shift registers to store a portion of data. In particular, this is necessary when the computation applies a fixed nearest-neighbor pattern to produce the output (e.g., an image processing filter). Hence, FROST analyzes the access pattern to the buffer in order to instantiate a proper sequence of shift registers. Like for the `streaming` architecture, FROST introduces additional code blocks to manage the insertion, access and shift of data within the shift registers. The main drawback of an N-bit vectorization relies in the fact that the number of elements in the buffer (in case of a multi-dimensional buffer, the last dimension) has to be multiple of N/K, where K is the bitwidth of the original buffer data. Consequently, the input may need to be padded, while the output may contain some garbage data. For instance, let us consider a 3×3 filter applied on a $N \times M$ single channel input image. The filtered output should be a $(N-2)\times(M-2)$ image. In case of vectorization, assuming the M-dimension does not need to be padded, the corresponding output is a $(N-2) \times M$ image, where two columns contain garbage data. Padding allows to maintain the hardware design simpler avoiding an invasive control flow.

Final steps: Once the IR manipulation is over, FROST analyzes the new ASTs in order to start the code generation. During the analysis, FROST extracts information related to the to the libraries to include (in case of mathematical functions or particular HLS data structure), and the type of the variables. Basically, FROST builds a lookup table for each function. Finally, FROST visits the ASTs one last time, and, during the generation of the C++ code, enforces the remaining scheduling commands as HLS directives.

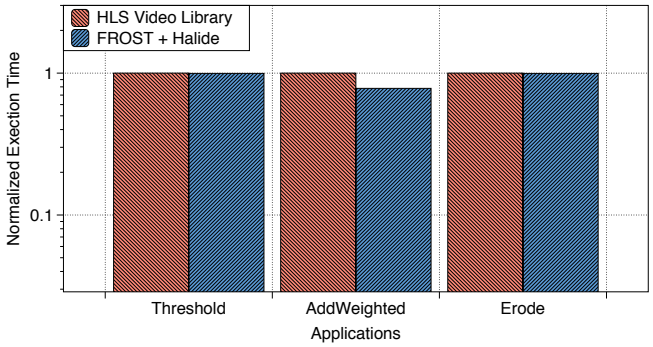

Fig. 4. Performance comparison between FROST with Halide and Vivado HLS Video Library.

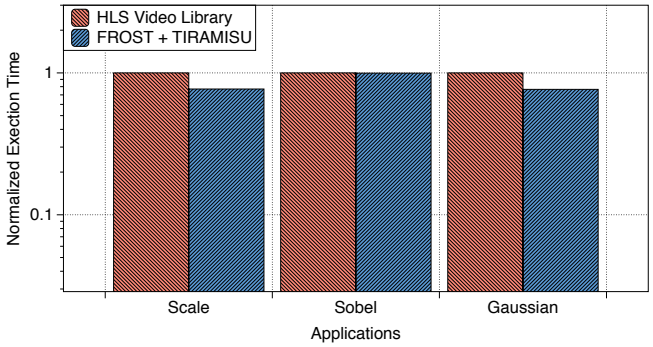

Fig. 5. Performance comparison between FROST with TIRAMISU (channel interleaved designs) and Vivado HLS Video Library.

E. backend

The last step of FROST workflow consist in generating the bitstream file to configure the FPGA. Once the IR optimization step is done, FROST produces a C++ implementation of the input algorithm suitable for HLS tools. Such code can be immediately imported in a HLS tool like Xilinx Vivado HLS [8] or Xilinx SDAccel [29] to have an estimation of the performance of the current design (for instance, in terms of circuit latency, resource usage, and so on). In this way, the user can verify whether the resulting design reaches, at least theoretically, the expected performance or not, and, if necessary, generate a new optimized design using FROST.

Once satisfied by the produced FPGA design, the user can employ SDAccel to write the host code using SDAccel APIs, and start the synthesis process to, eventually, produce the bitstream file. Indeed, SDAccel environment covers all the steps of the design flow for FPGA (i.e., the HLS and System Level Design steps) and automatizes them. Starting from the kernel generated by FROST, SDAccel first translates it to RTL, and the wraps the resulting IP Core within SDAccel infrastructure. Such infrastructure is in charge of enabling the communication between the board powered by the FPGA and the host machine via PCIe, as well as exploiting partial dynamic reconfiguration enable kernel reconfiguration at runtime. At the end of synthesis process, SDAccel produces the

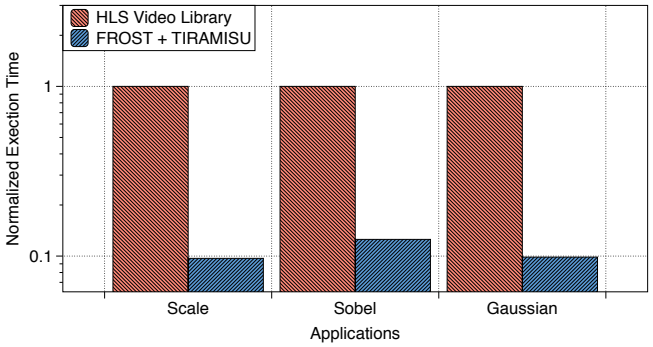

Fig. 6. Performance comparison between FROST with TIRAMISU (plane interleaved designs) and Vivado HLS Video Library.

bitstream file. Thus, the user's task consist only in writing the host code using SDAccel APIs to manage the FPGA configuration, and the communication with it via PCIe.

Currently, these steps (namely, evaluation of performance using a HLS tool, host code generation, and SDAccel invocation) are done manually by the user, but we plan to automatize them from FROST, as an additional scheduling command.

IV. Experimental Evaluation

This Section presents the experimental evaluation of FROST framework. We first describe the experimental setup of our work (Section IV-A), and then compare the designs produced by FROST against a hand-tuned library available in Vivado HLS framework, as well as the designs from [14] (Section IV-B).

A. Experimental Setup

For each design generated by FROST, we first leveraged Xilinx Vivado HLS 2017.2 to evaluate the performance and resource usage of the design, then Xilinx SDAccel 2017.2 to synthesize it and, consequently, produce the bitstream file. The board we targeted is an ADM-PCIE-7V3 by Alpha Data, which mounts a Xilinx Virtex-7 FPGA. We connected such board via PCIe to a host CPU (an Intel Core i7-870 at 2.93GHz), which leverages SDAccel APIs to manage application execution, the communication between the host and the FPGA, and other aspects of the computation.

B. Experimental Results

In this Section, we report the evaluation we performed on the designs produced by FROST. To this end, we used both Halide and TIRAMISU as frontends, and evaluated FROST with 6 image processing kernels, namely `Threshold`, `AddWeighted`, `Erode`, `Scale`, `Sobel`, `Gaussian`. We decided to target such kernels because they are already available within the Vivado HLS Video Library, a library implementing several hand-tuned OpenCV functions for FPGA. We designed the first three kernels using Halide, and the remaining with TIRAMISU. For each kernel, we compared the resulting design (in terms of execution time and resource usage) against the corresponding kernel in the HLS Video Library. The input

TABLE II
RESOURCE USAGE OF HALIDE BENCHMARKS

Application	*BRAM_18K* (2940)	*DSP48E* (3600)	*FF* (866400)	*LUT* (4332009)
FROST				
Threshold	2	0	814	1542
AddWeighted	0	30	5487	7383
Erode	8	0	1023	2012
Vivado HLS Video Library				
Threshold	0	0	307	1300
AddWeighted	0	30	6810	11465
Erode	9	0	2170	2828

of each kernel is a 8-bit FullHD (1920x1080) RGB image. The only exception is the `threshold` kernel, which works on single-channel images.

The Vivado HLS Video Library works in a channel interleaved manner, which means that it expects images arranged by channels (i.e. channels as innermost dimension). The main reason is to maintain consistency with software OpenCV library. This design choice enables to compute each channel in parallel (i.e. one pixel per clock cycle). However, a different data layout, like a plane interleaved one (i.e. image width as innermost dimension), would allow to process more elements in parallel (for instance, N elements belonging to the same channel per clock cycle). In this regard, we demonstrate that FROST is able to design both channel interleaved and plane interleaved designs, thanks to the integration with its frontends. In particular, we implemented all the kernels in a channel interleaved manner, while, for the ones expressed in TIRAMISU, we took advantage of TIRAMISU features to also rearrange the input in a plane interleaved manner. Moreover, for each kernel we leveraged both FROST and the frontends scheduling co-language to evaluated different optimizations, while we chose a `streaming` dataflow architecture to implement such kernels, since the are implemented in the same way within the Video Library. Finally, we synthesized each considered design at 200MHz.

Halide benchmarks: For the Halide benchmarks, we chose the following kernels: `Threshold`, `AddWeighted`, and `Erode`. We designed each kernel in a channel interleaved way, just like the HLS Video Library does. The resulting designs exploit both Halide and FROST scheduling commands to enable a parallel computation of the channels within each pixel (except the `Threshold` kernel, as it works on single-channel images). In Figure 4, we show the comparison, in terms of normalized execution time, between the FROST (with Halide) designs and the ones from the Video Library. FROST designs are able to match the performance of the HLS Video Library, and, in case of the `AddWeighted` kernel, outperform it (a speedup of 1.28X). Table II describes the resource usage of the considered kernels. We cna notice that the resource usage of FROST designs is in line with the one of the Video Library.

TABLE III
RESOURCE USAGE OF THE TIRAMISU BENCHMARKS

Application	*BRAM_18K* (2940)	*DSP48E* (3600)	*FF* (866400)	*LUT* (4332009)
FROST (channel interleaved)				
Scale	2	15	3550	4803
Sobel	8	0	1047	1787
Gaussian	8	0	997	1907
FROST (plane interleaved)				
Scale	30	320	59642	73933
Sobel	144	0	3287	7543
Gaussian	60	0	2757	7008
Vivado HLS Video Library				
Scale	0	15	4828	8792
Sobel	9	0	2220	3255
Gaussian	9	12	2560	3355

TIRAMISU benchmarks: The TIRAMISU benchmarks consist in the following image processing kernels: `Scale`, `Sobel`, and `Gaussian`. For each kernel, we implemented both a channel interleaved and plane interleaved design. We relied on both FROST and TIRAMISU scheduling commands to optimize the hardware designs. First of all, TIRAMISU allowed us to prepare the computation for vectorization (applying loop splitting), and, when necessary, rearrange it in a plane interleaved manner. Then, FROST applied vectorization to the hardware designs and built a `streaming` dataflow architecture. On one hand, for the channel interleaved designs, we packed the 3 channels into a single variable, just like HLS Video Library does. On the other hand, the plane interleaved designs leveraged a higher level of parallelism, as we packed the input in chunks of 512-bit (i.e., 64 elements per chunk). This value represents the maximum bit-width of the memory ports of the DDR mounted on the target board.

Figure 5 displays a comparison in terms of normalized execution time between FROST channel interleaved designs and the HLS Video Library designs, while, in Figure 6, the comparison is between FROST plane interleaved designs and the Video Library. In Figure 5, we notice that FROST designs match the performance of HLS Video Library, and, in two cases, it reaches a better execution time (a speedup up to 1.30X). In Figure 6, thanks to a higher level of parallelism, FROST designs significantly outperform the Video Library, reaching a speedup of 10X. Table III reports the resource usage of both FROST designs (channel and plane interleaved) and Vivado HLS Video Library designs. The channel interleaved designs have a resource usage similar to the Video Library, while the plane interleaved designs require a higher number of resources due to the higher level of parallelism.

Architectures comparison: Finally, we compared the performance of `Scale` and `Gaussian` (plane interleaved) kernels reported here against the ones described in [14]. The main difference in the two works mainly is the designed

architecture; indeed, even though both works operate on plane interleaved images and leverage the same level of parallelism, [14] implements a `tiled` architecture, while we rely on a `streaming` dataflow one. Besides, since dimensions of the input images are different, we compare the two works in terms of $MPixel/s$. The version of `Scale` and `Gaussian` implemented in [14] reach, respectively, $2488MPixel/s$ and $1771MPixel/s$. On the other hand, our designs, thanks to the `streaming` dataflow, achieve on the `Scale` and `Gaussian` kernels $4200MPixel/s$ and $4323MPixel/s$, respectively.

V. Conclusions and Future Work

This paper described FROST, a common backend for targeting FPGAs from DSLs. The design of FROST allows to support multiple DSLs as frontends, and leverages Xilinx SDAccel to generate the bitstream file. A crucial feature of FROST is a high-level scheduling co-language, which permits to optimize the resulting FPGA design according to the input application characteristics. In the experimental evaluation of FROST, we employed Halide and TIRAMISU as frontends, and reached the same level of both performance and resource usage of a hand-tuned HLS library for image processing kernels (when we used the same level of parallelism), and outperformed it up to 10X thanks a combination of TIRAMISU and FROST scheduling commands.

As future work, first of all, we plan to add support for additional DSLs that are not currently supported. Then, we intend to introduce other high-level scheduling commands to further improve the productivity and efficiency of FROST. Finally, we would like to better integrate FROST with Xilinx SDAccel. This would allow to invoke SDAccel directly from FROST, and enable users to easily evaluate the performance of a design and, consequently, optimize it according to the HLS phase results.

Acknowledgment

This work was supported by the European Commission in the context of the H2020 FETHPC EXTRA project (#671653).

References

[1] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *ACM SIGARCH Computer Architecture News*, vol. 39, no. 3. ACM, 2011, pp. 365–376.

[2] "TOP500 List," June 2017. [Online]. Available: https://www.top500.org/lists/2017/06/

[3] NVIDIA, "CUDA." [Online]. Available: https://developer.nvidia.com/about-cuda

[4] "Open MPI Project." [Online]. Available: https://www.open-mpi.org

[5] J. Ragan-Kelley, C. Barnes, A. Adams, S. Paris, F. Durand, and S. Amarasinghe, "Halide: a language and compiler for optimizing parallelism, locality, and recomputation in image processing pipelines," *ACM SIGPLAN Notices*, vol. 48, no. 6, pp. 519–530, 2013.

[6] R. Baghdadi, U. Beaugnon, A. Cohen, T. Grosser, M. Kruse, C. Reddy, S. Verdoolaege, A. Betts, A. F. Donaldson, J. Ketema *et al.*, "Pencil: A platform-neutral compute intermediate language for accelerator programming," in *Parallel Architecture and Compilation (PACT), 2015 International Conference on*. IEEE, 2015, pp. 138–149.

[7] R. Baghdadi, J. Ray, M. B. Romdhane, E. D. Sozzo, P. Suriana, S. Kamil, and S. Amarasinghe, "Tiramisu: A code optimization framework for high performance systems," *ArXiv e-prints*, February 2018.

[8] Xilinx Inc., "Vivado HLS." [Online]. Available: https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html

[9] J. Hegarty, J. Brunhaver, Z. DeVito, J. Ragan-Kelley, N. Cohen, S. Bell, A. Vasilyev, M. Horowitz, and P. Hanrahan, "Darkroom: compiling high-level image processing code into hardware pipelines." *ACM Trans. Graph.*, vol. 33, no. 4, pp. 144–1, 2014.

[10] R. Stewart, G. Michaelson, D. Bhowmik, P. Garcia, and A. Wallace, "A dataflow IR for memory efficient RIPL compilation to FPGAs," in *International Conference on Algorithms and Architectures for Parallel Processing*. Springer, 2016, pp. 174–188.

[11] N. Chugh, V. Vasista, S. Purini, and U. Bondhugula, "A DSL compiler for accelerating image processing pipelines on FPGAs," in *Parallel Architecture and Compilation Techniques (PACT), 2016 International Conference on*. IEEE, 2016, pp. 327–338.

[12] C. Schmitt, M. Schmid, F. Hannig, J. Teich, S. Kuckuk, and H. Köstler, "Generation of multigrid-based numerical solvers for FPGA accelerators," in *Proceedings of the 2nd International Workshop on High-Performance Stencil Computations (HiStencils)*, 2015, pp. 9–15.

[13] J. Pu, S. Bell, X. Yang, J. Setter, S. Richardson, J. Ragan-Kelley, and M. Horowitz, "Programming heterogeneous systems from an image processing dsl," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 3, p. 26, 2017.

[14] E. Del Sozzo, R. Baghdadi, S. Amarasinghe, and M. D. Santambrogio, "A common backend for hardware acceleration on fpga," in *Computer Design (ICCD), 2017 IEEE International Conference on*. IEEE, 2017, pp. 427–430.

[15] Xilinx Inc., "Xilinx Vivado HLS Video Library." [Online]. Available: http://www.wiki.xilinx.com/HLS+Video+Library

[16] Intel Inc., "Intel HLS Compiler." [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01274-intel-hls-compiler-fast-design-coding-and-hardware.pdf

[17] ——, "Intel FPGA SDK for OpenCL." [Online]. Available: https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/hb/opencl-sdk/aocl_getting_started.pdf

[18] Z. DeVito, J. Hegarty, A. Aiken, P. Hanrahan, and J. Vitek, "Terra: a multi-stage language for high-performance computing," in *ACM SIGPLAN Notices*, vol. 48, no. 6. ACM, 2013, pp. 105–116.

[19] E. Bezati, "High-level synthesis of dataflow programs for heterogeneous platforms," Ph.D. dissertation, EPFL, 2015.

[20] R. T. Mullapudi, V. Vasista, and U. Bondhugula, "Polymage: Automatic optimization for image processing pipelines," *SIGARCH Comput. Archit. News*, vol. 43, no. 1, pp. 429–443, Mar. 2015. [Online]. Available: http://doi.acm.org/10.1145/2786763.2694364

[21] L.-N. Pouchet, P. Zhang, P. Sadayappan, and J. Cong, "Polyhedral-based data reuse optimization for configurable computing," in *Proceedings of the ACM/SIGDA international symposium on Field programmable gate arrays*. ACM, 2013, pp. 29–38.

[22] U. Bondhugula, A. Hartono, J. Ramanujam, and P. Sadayappan, "Pluto: A practical and fully automatic polyhedral program optimization system," in *Proceedings of the ACM SIGPLAN 2008 Conference on Programming Language Design and Implementation (PLDI 08), Tucson, AZ (June 2008)*. Citeseer, 2008.

[23] W. Landi, "Undecidability of static analysis," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 4, pp. 323–337, Dec. 1992. [Online]. Available: http://doi.acm.org/10.1145/161494.161501

[24] J. Bezanson, S. Karpinski, V. B. Shah, and A. Edelman, "Julia: A fast dynamic language for technical computing," *arXiv preprint arXiv:1209.5145*, 2012.

[25] J. Bergstra, O. Breuleux, P. Lamblin, R. Pascanu, O. Delalleau, G. Desjardins, I. Goodfellow, A. Bergeron, Y. Bengio, and P. Kaelbling, "Theano: Deep learning on gpus with python," 2011.

[26] M. W. Hall, S. P. Amarasinghe, B. R. Murphy, S.-W. Liao, and M. S. Lam, "Detecting coarse-grain parallelism using an interprocedural parallelizing compiler," in *Supercomputing, 1995. Proceedings of the IEEE/ACM SC95 Conference*. IEEE, 1995, pp. 49–49.

[27] G. Tournavitis, Z. Wang, B. Franke, and M. F. O'Boyle, "Towards a holistic approach to auto-parallelization: integrating profile-driven parallelism detection and machine-learning based mapping," *ACM Sigplan Notices*, vol. 44, no. 6, pp. 177–187, 2009.

[28] Xilinx Inc., "AMBA AXI4 Interface Protocol." [Online]. Available: https://www.xilinx.com/products/intellectual-property/axi.html

[29] ——, "SDAccel Development Environment." [Online]. Available: https://www.xilinx.com/products/design-tools/software-zone/sdaccel.html