A Common Backend for Hardware Acceleration on FPGA

A Common Backend for Hardware Acceleration on FPGA

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

***Abstract*—Field Programmable Gate Arrays (FPGAs) are configurable integrated circuits able to provide a good trade-off in terms of performance, power consumption, and flexibility with respect to other architectures, like CPUs, GPUs and ASICs. The main drawback in using FPGAs, however, is their steep learning curve. An emerging solution to this problem is to write algorithms in a Domain Specific Language (DSL) and to let the DSL compiler generate efficient code targeting FPGAs. This work proposes *FROST*, a unified backend that enables different DSL compilers to target FPGA architectures. Differently from other code generation frameworks targeting FPGA, FROST exploits a scheduling co-language that enables users to have full control over which optimizations to apply in order to generate efficient code (e.g. loop pipelining, array partitioning, vectorization). At first, FROST analyzes and manipulates the input Abstract Syntax Tree (AST) in order to apply FPGA-oriented transformations and optimizations, then generates a C/C++ implementation suitable for High-Level Synthesis (HLS) tools. Finally, the output of HLS phase is synthesized and implemented on the target FPGA using Xilinx SDAccel toolchain. The experimental results show a speedup up of $15\times$ with respect to O3-optimized implementations of the same algorithms on CPU.**

I. Introduction

Due to the reaching of the end of Dennard scaling and Moore's law [1], we are experiencing a growing interest towards Heterogeneous System Architectures (HSAs) as a promising solution to boost performance and, at the same time, reduce power consumption. The combination of different hardware accelerators, like Graphic Processing Units (GPUs), Field Programmable Gate Arrays (FPGAs), and Application Specific Integrated Circuits (ASICs), along with Central Processing Units (CPUs), allows to choose the most suitable architecture for a specific task, and, for this reason, many high performance systems (like HPC, image and signal processing) are currently taking advantage of heterogeneity. Among the aforementioned architectures, FPGAs are a good candidate for high performance computations since they provide a good trade-off in terms of performance (higher than CPUs and similar to GPUs) and power consumption (lower than CPUs and GPUs), as well as a high level of flexibility (with respect to ASICs). Moreover, the possibility to configure FPGAs to implement a custom architecture (for instance, in terms of data precision) permits to tailor it to the target computation. However, the major flaw of FPGAs is their hard programmability and steep learning curve. Although High-Level Synthesis (HLS) tools [2] are easing the implementation of algorithms on FPGAs (using C/C++ instead of Verilog or VHDL), the design process is still complex. An emerging solution to this problem is to write algorithms in a Domain Specific Language (DSL) and to let the DSL compiler generate efficient code targeting FPGAs [3]–[6].

This paper presents *FROST*, a unified backend that enables DSL compilers to target FPGA architectures. A DSL compiler can generate the FROST Intermediate Representation (IR) and use FROST to generate efficient HLS code to target FPGA. FROST leverages a scheduling co-language to specify FPGA specific optimizations (loop pipelining, unrolling, vectorization) as well as the type of communication with the off-chip memory. After applying the transformations on the IR, FROST generates a C/C++ code suitable for HLS tools. Finally, the outcome of HLS phase is synthesized and implemented on FPGA using a synthesis toolchain.

Given that many FROST optimizations require higher level loop nest transformations (e.g., vectorization requires the loops to be split), FROST is designed to integrate well with higher level loop nest transformation framework (such as the Halide mid-level compiler and the Tiramisu optimization framework). These frameworks are a layer between DSL languages and the FPGA code generation layer. This separation allows FROST to fully focus on efficient FPGA code generation and to leave loop transformations that are architecture-independent to higher level frameworks.

The paper is organized as follows: Section II describes the FROST IR, optimizations and code generator. Section III presents an experimental evaluation of FROST, while Section IV discusses related work.

II. Frost

This section explains the rationale behind FROST design, and provides a detailed description of its workflow.

A. Common Backend

Recently, the use of DSLs and high level languages such as Halide [7], Tensorflow [8], Julia [9] and Theano [10] has been gaining in popularity for many reasons: (1) they provide portability across multiple hardware architectures; (2) they allow the application of certain optimizations such as fusion,

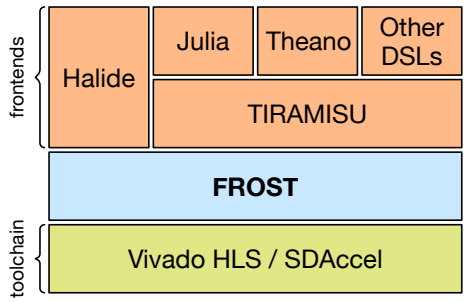

Fig. 1. FROST stack

and data layout transformations that are difficult otherwise, and (3) they provide high productivity. FROST is a unified backend that allows DSL compilers to target FPGA. 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, a state-of-the-art DSL and compiler for image processing pipelines, as well as TIRAMISU, a unified optimization framework for DSL compilers, and which presently is integrated in DSLs such as Julia [9] and Theano [10]. We are working on a full integration of FROST within Halide and Tiramisu; once such integration is done, FROST will be demonstrated on Halide and on all the DSL languages that use Tiramisu as an optimization framework (Theano and Julia). In this paper, we will focus on presenting FROST itself, and on evaluating FROST in isolation from higher level frameworks. A full end-to-end evaluation of FROST with many high level DSLs is left for a future paper.

The output of FROST is a C/C++ optimized implementation of the original algorithm suitable for HLS tools. In particular, FROST relies on two Xilinx toolchains, Vivado HLS [2] and SDAccel [11], [12] to, respectively, evaluate the performance of the current implementation (in terms of latency and resource usage), and generate the FPGA implementation. Thanks to SDAccel, the user does not have to design all the architecture surrounding the IP Core, nor to write all the code necessary to manage the runtime of the application as SDAccel completely takes care of these steps. The user just needs to write an host code and exploit SDAccel APIs to flash the bitstream, and communicate with the FPGA via PCIe. Figure 1 shows the complete FROST stack.

B. Scheduling Co-Language

The main motivation behind the choice of a scheduling co-language relies on the fact that generating efficient code for FPGA is definitely a hard task, especially if we want to support a wide range of applications. A generic approach may be a viable solution, but it would not allow to efficiently leverage the FPGA potentialities. Moreover, also HLS tools provide a set of pragma comments the designer may exploit to specify some optimizations to enforce (e.g. loop pipelining), and define the communication interfaces with the off-chip memory. However, many other optimizations require significant code changes, and are currently up to the designer's skills and expertise with FPGAs. Correspondingly, the designer has to implement an efficient communication with the off-chip memory complaint with the chosen interfaces. Therefore, evaluating different hardware designs quickly turns into a time-consuming and error-prone task. For these reasons, the purpose of FROST scheduling co-language is to guide the optimizations to apply, in order to easily evaluate different solutions and find the most suitable one for the target algorithm. The designer just has to insert a set of scheduling commands, and FROST will generate the corresponding code.

FROST scheduling commands may refer to how the computation has to be carried out (e.g. loop pipelining, vectorization), how the data has to be stored on the FPGA (e.g. array partitioning), and the type of communication with the off-chip DDR memory. Currently, FROST supports a master/slave communication based on AMBA AXI4 interface protocol, where the data are copied on the on-chip BRAM memory before starting the computation, and copied back to the DDR once the computation is over. This approach allows to exploit data locality but, on the other hand, is constrained by the limited number of BRAM on the FPGA. In general, FROST scheduling commands may be split into two categories: the former includes the commands that require FROST IR manipulation, the latter the ones that can be expressed as pragmas for the HLS tools. According to the category the scheduling commands belong to, FROST handles them in different ways.

C. Workflow

FROST receives as input one or more functions, described in FROST IR, as well as a set of scheduling commands to apply. According to FROST IR, each function is a data structure mainly containing the name of the function itself, its arguments (defined as either buffers or scalars), an Abstract Syntax Tree (AST) representing the body of the function, along with other parameters. At first, FROST builds a top function, i.e. the main function to be synthesized on FPGA. This function is in change of invoking all the input functions, instantiating the local buffers, declaring the memory interfaces, and handling the data transfer from/to the off-chip DDR memory. In particular, FROST analyzes each input function argument to identify and separate global arguments, i.e. the ones referring to data to be read/written from/to the off-chip memory, and temporary arguments, the ones existing only in the context of the top function. Therefore, the global arguments are the arguments of the top function. For example, let us consider a blur filter, designed as a chain of `BlurX` and `BlurY` filter. The arguments/buffers of `BlurX` are `InX` and `OutX`, while the arguments/buffers of `BlurY` are `InY` and `OutY`. Actually, since these two filters work as a pipeline, the output of `BlurX` is the input of `BlurY`, hence `InY` is `OutX`. As a result, `InX` and `OutY` are the global buffers, while `OutX`/`InY` is a temporary buffer.

After creating the top function, FROST starts manipulating the functions ASTs to apply some of the scheduling commands, i.e. the ones that require FROST IR transformation. One example is the vectorization scheduling command, where

the buffer data are packed in bunches of N bits. For instance, a 512-bit vectorization of a 32-bit integer buffer will pack 16 data 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. First of all, the buffer data type needs to be updated to the proper one. Then, the access to the data bunches has to be changed as well. Finally, and most important, in order to fully take advantage of vectorization, it may be necessary to use support data structures like line buffers or shift-registers to properly store a portion of the data. Indeed, this is necessary when the computation operates on a window of data, like the blur filter, which applies a fixed nearest-neighbor pattern to produce the output pixels. In such a case, FROST analyzes the access pattern to the buffer in order to instantiate a support data structure of the proper dimension. Then, FROST introduces additional IR statements in charge of populating the data structure, accessing to its values, and shifting them. The main drawback of an N-bit vectorization is 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 buffer data. As a consequence, the input may need to be padded, while the output may contain some garbage data. For instance, the output of a 3x3 blur filter should be $(N-2)\cdot(M-2)$, where N and M are, respectively, the height and the width of the input image. In case of vectorization, assuming the M-dimension does not need to be padded, the corresponding output is a $(N-2)xM$ image, where two columns contain garbage data.

Once the IR manipulation is over, FROST analyzes the new ASTs in order to start the code generation. The analysis extracts information related to the libraries to include, and the type of the variables (FROST builds a lookup table). Then, FROST visits the ASTs, generates the C++ code, and enforces the remaining scheduling commands as HLS pragmas. At that point, the user can evaluate the implementation with Vivado HLS, and synthesize it to FPGA with SDAccel.

It is important to notice that FROST is not designed to perform transformation like loop splitting, loop tiling, and so on. Such transformations are surely useful and necessary in some cases (e.g. vectorization), but, since tools like Halide and TIRAMISU already support them, it was useless to implement them again. Therefore, to achieve better performance in terms of FPGA implementation, FROST and its frontends has to work in synergy.

III. Experimental Results

We evaluated FROST against three applications designed in TIRAMISU: brightening filter, blur filter, and matrix multiplication. The blur filter is implemented as a pipeline of two filters: one over the input image width, and one over the height. The input for the two filters is a 3-channel 512x384 image, while the size of the input matrices for the third application is 400x400. We compared an FPGA implementation of each application against an O3-optimized CPU implementation. We relied on Xilinx Vivado HLS and SDAccel 2016.4 to,

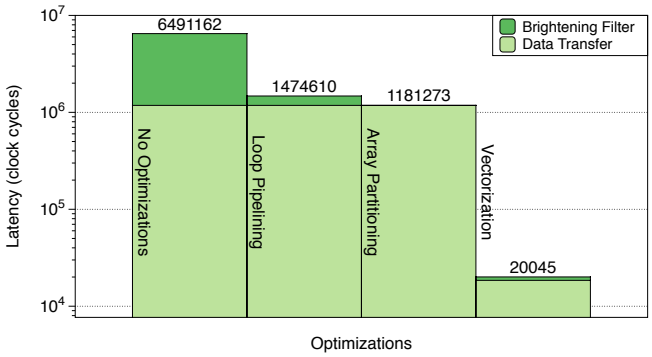

Fig. 2. A summary of the sequence of optimizations applied on the brightening filter. The input is a 3-channel 512x384 image

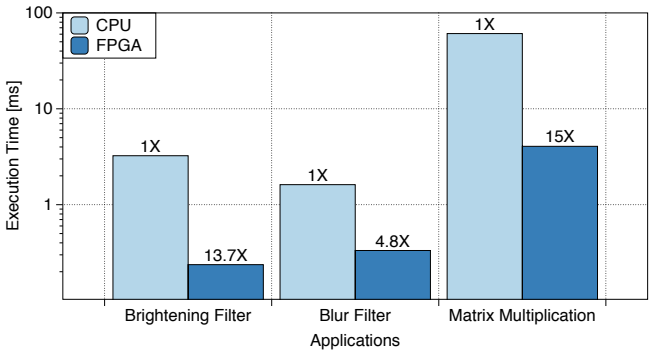

Fig. 3. Performance comparison between FPGA and CPU implementations

respectively, evaluate the different implementations generated by FROST and generate the bitstreams for FPGA. We targeted a ADM-PCIE-7V3 board by Alpha Data, powered by a Xilinx Virtex 7 FPGA. The board was connected to an host CPU, an Intel Core i7-870 at 2.93GHz, via PCIe. The host was in charge of executing the application, managing the FPGA runtime by means of SDAccel APIs, and measuring the execution time of both CPU and FPGA implementations.

For each application, we generated a non-optimized implementation with FROST, and we evaluated it in terms of performance (i.e. circuit latency), resource usage by looking at Vivado HLS log. Then, we started applying different types of optimizations by means of FROST and TIRAMISU scheduling co-languages, like loop pipelining, loop splitting, array partitioning, and vectorization. The optimizations were guided by the information (e.g. circuit latency, resource usage, warnings) produced by Vivado HLS for each implementation. For instance, in order to reduce the circuit latency, we applied loop pipelining/unrolling. In case of warnings about limited memory ports with respect to a certain array access, we enforced array partitioning. In order to reduce the latency of both data transfer and computation, we leveraged vectorization. Finally, once the optimization process was over, we synthesized the circuit using SDAccel. Figure 2 shows a summary of the optimizations applied to the brightening filter.

The final FPGA implementations allowed to achieve a $13.7\times$ speedup for the brightening filter, $4.8\times$ speedup for the

TABLE I
RESOURCE USAGE OF THE EVALUATED APPLICATIONS

Application	*BRAM_18K* (2940)	*DSP48E* (3600)	*FF* (866400)	*LUT* (4332009)
Brightening Filter	24.2%	32%	29.5%	78.8%
Blur Filter	47.5%	0.1%	0.4%	2%
Matrix Multiplication	44.7%	44.5%	4.6%	3.9%

blur filter, and a $15\times$ speedup for the matrix multiplication (Figure 3). Table I reports the resource usage of the final implementation of each application. Finally, the blur filter implementation outperformed the one available in the Vivado HLS OpenCV library by a $6\times$ factor.

IV. RELATED WORK

Many frameworks and compilers in literature focus on a specific contexts to generate an efficient hardware implementation. Darkroom [3] is a language and compiler for image processing. Darkroom compiles a high level description of the application into line-buffered pipelines, described in Verilog, which are then synthesized for ASIC, FPGA, or CPU. The experimental results showed gigapixel/sec performance when targeting ASIC, while realtime 1080p/60 video processing using FPGAs. RIPL [4] is a memory-efficient, declarative FPGA image processing DSL. RIPL programs are first compiled to dataflow graphs, and then to HDL by means of an open source dataflow compiler [13]. The authors evaluated RIPL against five benchmarks and reported a memory use comparable to Vivado HLS OpenCV library, without the need of pragmas to guide the synthesis. In [5], the authors developed an FPGA backend for the PolyMage DSL [14]. The backend first applies optimizations to exploit data parallelism and memory bandwidth, then relies on Vivado HLS to generate the FPGA implementation. The authors compared their results against Vivado HLS OpenCV library, as well as Darkroom, and achieved, on average, a $1.5\times$ speedup. The work in [6] presents ExaSlang 4, a DSL to accelerate numerical solvers based on the multigrid method on FPGA. The authors leveraged Vivado HLS as backend to produce the FPGA design.

Differently from the aforementioned work, we designed FROST to deal with generic computations, as well as support high level languages, instead of constraining the user to a specific context and language. On the other hand, FROST provides a high level scheduling co-language to specify the optimizations to enforce. This feature is necessary in order to support generic computations, and, at the same time, generate efficient FPGA implementations.

V. CONCLUSIONS AND FUTURE WORK

We presented FROST, a common backend for the hardware acceleration on FPGA. FROST is designed to work with any frontend that produces an IR compatible with FROST IR, like Halide and TIRAMISU. On the other hand, FROST scheduling co-language allows the user to enforce high level optimizations as well as specify the communication with the memory. The experimental evaluation showed that the final FPGA implementations significantly outperformed O3-optimized CPU implementations.

As future work, we plan to fully integrate FROST within Halide and TIRAMISU, and evaluate FROST end-to-end on Halide, Theano and Julia. Then, we intend to introduce support for a streaming/dataflow communication, which will definitely benefit computations like filters. Finally, we aim at directly connecting the output of FROST to Xilinx toolchain. In this way, the user can analyze the performance of the current implementation, as well as the warnings generated by Vivado HLS, tune the implementation according to that, and, once satisfied, start the FPGA synthesis process via SDAccel.

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

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] 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.

[8] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard, M. Kudlur, J. Levenberg, R. Monga, S. Moore, D. G. Murray, B. Steiner, P. Tucker, V. Vasudevan, P. Warden, M. Wicke, Y. Yu, and X. Zheng, "Tensorflow: A system for large-scale machine learning," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, 2016, pp. 265–283. [Online]. Available: https://www.usenix.org/system/files/conference/osdi16/osdi16-abadi.pdf

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

[10] 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.

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

[12] G. Guidi, E. Reggiani, L. Di Tucci, G. Durelli, M. Blott, and M. D. Santambrogio, "On how to improve FPGA-based systems design productivity via SDAccel," in *Parallel and Distributed Processing Symposium Workshops, 2016 IEEE International*. IEEE, 2016, pp. 247–252.

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

[14] 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