Reflections on 10 years of FloPoCo

Florent de Dinechin

Reflections on 10 years of FloPoCo

Florent de Dinechin
Univ Lyon, INSA Lyon, Inria, CITI, France
florent.de-dinechin@insa-lyon.fr

The FloPoCo open-source arithmetic core generator project started modestly in 2008 [1], with a few parametric floating point cores. It has since then evolved to become a framework for research on hardware arithmetic cores at large, including among others: LNS arithmetic [2], random number generators [3], elementary functions [4]–[9], specialized operators such as constant multiplication and division [10]–[13], various FPGA-specific optimization techniques [14]–[16], and more recently signal-processing transforms and filters [17], [18] (more references can be found on the project's web site: http://flopoco.gforge.inria.fr/).

I. Philosophy

Along with this evolution, the tool has slowly crystallized a productive hardware arithmetic paradigm: *open-ended* generation of *application-specific* operators that *compute just right* thanks to *last-bit accuracy* at all levels.

One key principle of operator design within FloPoCo is to systematically correlate the accuracy with the precision. A component doesn't need to be more accurate than it can express on its output, and conversely no component should output bits that do not carry information. This is essential for performance, but it also has some nice side-effects. One is that it simplifies the interface: no need to specify the accuracy, it is encoded in the output format. Indeed, for several FloPoCo papers, a central contribution has been a better definition of the core problem thanks to such interface simplifications.

FloPoCo is also a very pragmatic project. For instance we don't care too much if we don't have a nice closed-form formula for the optimal value of some parameter, as long as there exists a fast enough program that can compute it... or get close enough to improve the state of the art.

Nevertheless, during this first decade, FloPoCo has gained more and more abstraction. Some of it is uncontroversial, for instance specification-based testing, or the internal support of fixed-point formats. Let us review a few other technical choices that are more disputable.

II. A critical review of some technical choices

A. Print-based VHDL back-end

A FloPoCo operator is a program that prints VHDL to a file. Most operators have no further abstraction of the architecture in a data structure. This primitive choice was initially motivated by 1/ the will to quickly recycle in FloPoCo various bits and pieces of VHDL written by previous students, 2/ the will to have trainees immediately operational in FloPoCo if they were fluent in VHDL, and 3/ laziness to reinvent a language – if MyHDL [19] or Chisel [20] had been around at the time, FloPoCo could have been based on one of them.

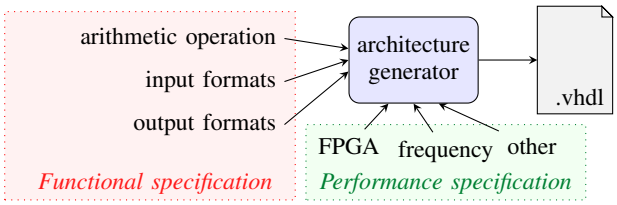

Fig. 1. The interface to an arithmetic generator

VHDL is thus deeply rooted in FloPoCo. We have received numerous requests for a Verilog backend, but none was convincing that this would open new research opportunities. The emergence of HLS tools is another matter. The very purpose of FloPoCo is to build application-specific operators. In a classical design flow, some designer has to manually translate the needs of her application into a set of operator specifications for FloPoCo. In an HLS design flow, there is the opportunity that the compiler, which knows the context of each operation, can perform this translation more automatically. The tie to VHDL is an issue here, which we discuss further in Section III.

B. Automatic pipeline

FloPoCo attempts to pipeline its operators for a user-specified frequency on a user-specified target (Figure 1). This is another disputed choice that is costly to maintain: the current pipeline framework is the third [16], and porting all the existing operators to this framework consumes an insane amount of time that should be dedicated to arithmetic research.

In principle, Leiserson-Saxe retiming should make this effort useless. I have heard this argument for 10 years, but I am still waiting for a democratization of retiming tools. They may indeed never arrive, because HLS scheduling is now providing the same service, with a narrowing performance gap [21].

Another issue with automatic pipelining is that it relies on vendor synthesis tools, which are less and less predictible. In particular, routing delays have become at the same time preponderant and completely random. Arithmetic operators, by definition, should remain small, hence compact with mostly predictable local routing. Alas, it is incresingly difficult to convince vendor tools of this.

Still, automatic pipelining has been an important factor in the success of the project, one that makes the difference between a proof of concept and an useful tool. The challenge here is to find the proper balance between the time spent

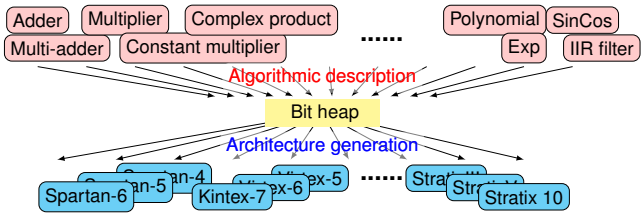

Fig. 2. A bit-heap-centric view on operator generation

in addressing such engineering issues, and the time spent in arithmetic research.

C. The bit heap grand plan

Bit heaps are a generalization of the bit arrays classically used in multiplier design [22]. In FloPoCo, they have been used since 2013 [15] to capture summations in a surprisingly large number of operators, including sums of products and more generally polynomials of several variables, but also all sorts of sums of tabulated values, for instance in KCM-based FIR and IIR filters [17] or in multipartite table methods for function evaluation [23]. They are in principle an elegant way of decoupling the description of a problem with the generation of target-optimized hardware (Figure 2). It is not uncommon for a complex operator such as an elementary function to involve several bit heaps. Having an open-source bit-heap framework in FloPoCo has also renewed interest in bit array compression, improving the state of the art with an ILP-based heuristic [24] that, in return, improves the performance of a large number of FloPoCo operators.

However, some ideas of the initial grand plan [15] failed to materialize so far. One was to view FPGA DSP blocks as (particularly) large compressors. Another was to deploy binary-level algebraic optimizations inside the bit-heap framework. There was also the idea that bit heaps could be used as a measure of the bit-level complexity of an operator (see Figure 3). On FPGAs, whose architecture encourage tabulation and DSP-centric operations, this measure is not that relevant.

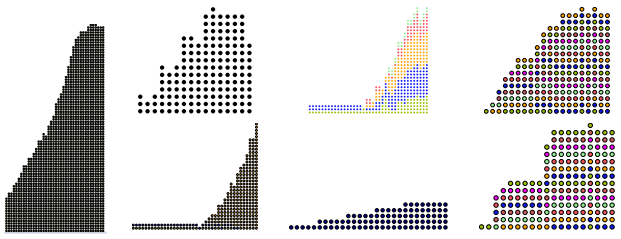

Fig. 3. A gallery of bit heaps from various FloPoCo operators

III. Should we drop FloPoCo for HLS?

Early experiments to embed the FloPoCo spirit in a source-to-source compiler [25] yielded context-specific operators that are simply out of reach of FloPoCo. The years when FloPoCo-like generation was the best way to produce floating-point operators tuned for their context may well be over.

However, an HLS compiler may optimize operators, it doesn't invent them so far: this remains FloPoCo's task. Indeed, the current FloPoCo flagships are no longer the basic floating-point operators that HLS projects are catching up [21], but open-ended generators, e.g. for arbitrary elementary function approximators in fixed point [4] or floating-point [9], or IIR filters [17]. In the former, the external library Sollya [26] is invoked to compute large numbers of approximation polynomials (a process that itself involves numerous multiple-precision interval computations, an optimization based on Euclidean lattice basis reduction [27], etc). Sollya also provides safe bounds of the approximation errors of these polynomials. FloPoCo itself heuristically explores the architecture space, attempting to size multipliers, evaluating the corresponding rounding error bounds, combining them with the approximation error bounds, and looking for the best solution that offers the required final accuracy. The result of this complex and costly process is a comparatively small table of fixed-point values, and the specifications of a handful of multipliers: generating the corresponding hardware is quite simple compared to the generation of the parameters.

It is probably safe to assume that such explorations are out of reach of HLS tools for at least one more decade. However, it would be useful to modularize FloPoCo so that the heavy computations can be easily retargeted to other back-ends or invoked in other contexts, including HLS.

IV. Conclusion: the next ten years

The first ten years of FloPoCo have been quite successful. Its operators are used in many other projects, mostly FPGA-based, but also a few ASICs. Maintaining this service is a lot of work, but it has been well rewarded, for instance with the FPL community award in 2017, or with hundreds of citations by strangers for the publication labelled "how to cite FloPoCo".

The existence of this open-source code base has allowed for increasingly sophisticated arithmetic core generation. FloPoCo has become over the years dependent on more and more external libraries and tools: MPFR for arbitrary precision, then Sollya for polynomial approximations, then ScaLP for integer linear programming, among others. The modern core generator routinely invokes such complex optimization techniques, very far from the simple parameterization of the first releases. Meanwhile, what FloPoCo was best at ten years ago is arguably better done in HLS these days.

There are many more research directions where FloPoCo could help, for instance finite-field arithmetic, or formal proofs of arithmetic hardware.

To conclude, FloPoCo is not only a generator, it is first and foremost a research tool for the arithmetic community. If this project has proven one thing, it is that the open-source software model can be a booster for research.

References

[1] F. de Dinechin, C. Klein, and B. Pasca, "Generating high-performance custom floating-point pipelines," in *Field Programmable Logic and Applications*. IEEE, Aug. 2009, pp. 59–64.

[2] P. D. Vouzis, S. Collange, and M. G. Arnold, "A novel cotransformation for LNS subtraction," *Journal of Signal Processing Systems*, vol. 58, no. 1, pp. 29–40, 2010.

[3] D. B. Thomas, "Parallel generation of gaussian random numbers using the table-hadamard transform," in *FPGAs for Custom Computing Machines*, 2013.

[4] F. de Dinechin, M. Joldes, and B. Pasca, "Automatic generation of polynomial-based hardware architectures for function evaluation," in *Application-specific Systems, Architectures and Processors*. IEEE, 2010.

[5] F. de Dinechin and B. Pasca, "Floating-point exponential functions for DSP-enabled FPGAs," in *Field Programmable Technologies*, Dec. 2010, pp. 110–117.

[6] F. de Dinechin, P. Echeverría, M. López-Vallejo, and B. Pasca, "Floating-point exponentiation units for reconfigurable computing," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 6, no. 1, 2013.

[7] F. de Dinechin, M. Istoan, and G. Sergent, "Fixed-point trigonometric functions on FPGAs," *SIGARCH Computer Architecture News*, vol. 41, no. 5, pp. 83–88, 2013.

[8] F. de Dinechin and M. Istoan, "Hardware implementations of fixed-point Atan2," in *22nd IEEE Symposium of Computer Arithmetic (ARITH-22)*, Jun. 2015, pp. 34–41.

[9] D. B. Thomas, "A general-purpose method for faithfully rounded floating-point function approximation in FPGAs," in *22d Symposium on Computer Arithmetic*. IEEE, 2015.

[10] M. Kumm, K. Liebisch, and P. Zipf, "Reduced Complexity Single and Multiple Constant Multiplication in Floating Point Precision," in *IEEE International Conference on Field Programmable Logic and Application (FPL)*, 2012, pp. 255–261.

[11] F. de Dinechin, "Multiplication by rational constants," *IEEE Transactions on Circuits and Systems, II*, vol. 52, no. 2, pp. 98–102, Feb. 2012.

[12] M. Kumm, O. Gustafsson, M. Garrido, and P. Zipf, "Optimal single constant multiplication using ternary adders," *IEEE Transactions on Circuits and Systems II: Express Briefs*, 2016.

[13] H. F. Ugurdag, F. de Dinechin, Y. S. Gener, S. Gren, and L.-S. Didier, "Hardware division by small integer constants," *IEEE Transactions on Computers*, vol. 66, no. 12, pp. 2097–2110, 2017.

[14] H. D. Nguyen, B. Pasca, and T. Preusser, "FPGA-specific arithmetic optimizations of short-latency adders," in *Field Programmable Logic and Applications*, 2011, pp. 232–237.

[15] N. Brunie, F. de Dinechin, M. Istoan, G. Sergent, K. Illyes, and B. Popa, "Arithmetic core generation using bit heaps," in *Field-Programmable Logic and Applications*, Sep. 2013.

[16] M. Istoan and F. de Dinechin, "Automating the pipeline of arithmetic datapaths," in *DATE 2017*, Lausanne, Switzerland, Mar. 2017.

[17] A. Volkova, M. Istoan, F. de Dinechin, and T. Hilaire, "Towards hardware IIR filters computing just right: Direct form I case study," *IEEE Transactions on Computers*, vol. 68, no. 4, Apr. 2019.

[18] M. Garrido, K. Möller, and M. Kumm, "A floating-point processor for fast and accurate sine/cosine evaluation," *Transactions on Circuits and Systems I: Regular Papers*, vol. 66, no. 4, pp. 1507–1516, 2019.

[19] J. Decaluwe, "MyHDL," http://www.myhdl.org/.

[20] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: constructing hardware in a scala embedded language," in *DAC Design Automation Conference 2012*. IEEE, 2012, pp. 1212–1221.

[21] S. Bansal, H. Hsiao, T. Czajkowski, and J. H. Anderson, "High-level synthesis of software-customizable floating-point cores," in *2018 Design, Automation & Test in Europe*. IEEE, 2018, pp. 37–42.

[22] M. D. Ercegovac and T. Lang, *Digital Arithmetic*. Morgan Kaufmann, 2003.

[23] S.-F. Hsiao, P.-H. Wu, C.-S. Wen, and P. K. Meher, "Table size reduction methods for faithfully rounded lookup-table-based multiplierless function evaluation," *Transactions on Circuits and Systems II*, vol. 62, no. 5, pp. 466–470, 2015.

[24] M. Kumm and J. Kappauf, "Advanced compressor tree synthesis for FPGAs," *IEEE Transactions on Computers*, vol. 67, no. 8, pp. 1078–1091, 2018.

[25] Y. Uguen, F. de Dinechin, and S. Derrien, "Bridging High-Level Synthesis and Application-Specific Arithmetic: The Case Study of Floating-Point Summations," in *27th International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, Sep. 2017. [Online]. Available: https://hal.inria.fr/hal-01373954

[26] S. Chevillard, M. Joldeş, and C. Lauter, "Sollya: An environment for the development of numerical codes," in *Mathematical Software - ICMS 2010*, ser. Lecture Notes in Computer Science, vol. 6327. Springer, Sep. 2010.

[27] N. Brisebarre and S. Chevillard, "Efficient polynomial L^∞- approximations," in *18th Symposium on Computer Arithmetic*. IEEE, 2007, pp. 169–176.