Towards Higher-Level Synthesis and Co-design with Python

Alexandre Quenon
alexandre.quenon@umons.ac.be
University of Mons
Mons, Belgium

Vitor Ramos Gomes da Silva
vitor.ramosgomesdasilva@umons.ac.be
University of Mons
Mons, Belgium

ABSTRACT

Several methods have arisen to fasten the hardware design process. Among them, the high-level synthesis (HLS), i.e., the use of a higher-level programming language than the usual Verilog or VHDL to create an implementation of a register transfer level (RTL). In this paper, the direction towards even higher-level synthesis is promoted with the use of Python as a high-level language/interface. Existing HLS frameworks and high-level hardware description languages are reviewed, then strategies to use Python code directly on the hardware are proposed. This brings the power of scientific high-level computation libraries of Python to the hardware design, which we believe is the ultimate goal of HLS.

KEYWORDS

FPGA, framework, hardware description language (HDL), high-level synthesis (HLS), Python.

1 INTRODUCTION

Over the past decades, huge efforts have been deployed to increase the speed and throughput, as well as to decrease the latency of communications and data processing. This is particularly critical for real-time applications, such as slow-motion full HD cameras or autonomous vehicles.

One of the technological answers to this issue is the so-called hardware acceleration, which consists of using circuits instead of computations to speed up the processing. A well-known example is the use of a Field Programmable Gate Array (FPGA) together with a CPU or a GPU.

Unfortunately, hardware acceleration is not easily accessible to software developers. Indeed, it requires skills in hardware description languages (HDLs), which have a design flow and design constraints quite different from other programming languages. In addition, the development of digital circuits has a longer time-to-market than pure software. Fortunately, two facts have started to mitigate this issue: (1) the adoption of FPGAs, which have a far shorter time-to-market than Application-Specific Integrated Circuits (ASICs), and (2) the rise of the so-called high-level synthesis (HLS), which consists in coding in a more usual "high-level" programming language that will be used to generate an HDL (Verilog or VHDL, typically) or directly the RTL view. This assessment can be generalized not only to hardware acceleration, but also to any digital hardware development.

In this paper, we promote the idea to go further in the concept of high-level synthesis and co-design thanks to the Python programming language. We believe that making hardware and software designers work closer will fasten production and generate innovation. We start by reviewing shortly the existing frameworks for HLS, as well as higher-level HDLs, in section 2. Then, in section 3, we focus on Python and propose strategies of hardware acceleration to use this high-level language directly on an FPGA. Section 4 summarizes the main ideas and draws the conclusions.

2 A SHORT OVERVIEW OF HIGH-LEVEL SYNTHESIS FRAMEWORKS AND HARDWARE DESCRIPTION LANGUAGES

On the one hand, many high-level hardware description languages have been designed for more than twenty years. On the other hand, a few frameworks dedicated to high-level synthesis have been created since the well-known Vivado HLS from Xilinx was released. Depending on the (self-) definition, the nature of the tool's or framework's output can be either another HDL, such as Verilog or VHDL, or a direct implementation of an RTL view. Nevertheless, in the remaining text of this article, no more distinction between HLS and HDL will be made, as the discussion focuses on the possibility to design the hardware in a high-level language.

Here below follows a selection of such high-level frameworks, sorted by first impacting publication, in chronological order:

1998	Lava [4], created in Haskell;
2008	Kiwi [10], written in C#;
2010	Clash [2], also in Haskell;
2011	FloPoCo [6], made in C++;
2012	Chisel [3], designed in Scala;
2012	Vivado HLS [14], enabling C and C++;
2013	LegUp [5], built in C;
2014	PyMTL [12, 13], written in Python;
2018	LeFlow [15], created in Python;
2021	PyLog [11], also in Python.

It is interesting to note that there is no correlation between the date of the release of the framework and the date of creation of the language used to design it.

All those frameworks have in common that they provide to the user facilities to generate commonly used HDLs, i.e., Verilog and VHDL, or even to create directly an implementation of the RTL view of a digital circuit. Some of them specifically target FPGAs.

However, the most important feature is not the language itself, but what it can enable and provide if used for high-level synthesis. In this way, our opinion is that Python can play a major role in the future because of the powerful scientific computation libraries, such as NumPy, and the tools dedicated to artificial intelligence, e.g., the application programming interface (API) with TensorFlow.

Even though, in most cases, Python is a high-level wrapper to C/C++ optimized libraries. These, and the rapid growth in artificial intelligence applications, might explain that the three more recent frameworks have been designed in Python.

To summarize, Python offers high-level programming capabilities, as well as powerful APIs to scientific computation and artificial intelligence libraries. By nature of the language, the software community is already organized in two teams working together: the core developers, who write the low-level, optimized libraries, and the application developers, who write the Python modules and packages for dedicated applications or usages. Consequently, hardware developers could find their legitimate position in the community, in conjunction with the core developers, to offer high-level synthesis and hardware acceleration to "pure" software users. In the end, this would simplify and accelerate co-design of hybrid hardware-software solutions.

3 STRATEGIES FOR PYTHON IMPLEMENTATION ON FPGAS

To restrain the discussion, only the strategies to implement high-level synthesis with Python on an FPGA will be discussed. In other words, we are looking for the possibilities to "execute" a Python code with an FPGA. To do so, three main strategies can be chosen:

(1) the direct execution of the Python bytecode [7, 8],
(2) writing hot functions in the FPGA, i.e., commonly executed functions [9, 16], and
(3) writing a transpiler [1].

3.1 Direct execution of the Python bytecode

The first proposed strategy is the direct execution of the Python bytecode, i.e., creating a CPU architecture capable of decoding Python directly inside the FPGA.

Advantages—This could considerably reduce the execution time even compared to JIT (just-in-time compiler) solutions since the application would run like a standalone binary without an operating system, and there would be no need for an extra layer of translation from Python to native binary code.

Drawbacks—All built-in functions have to be implemented in the architecture. For the pure Python code, this is feasible. The problem arrives when we try to include external libraries that use c-bindings, which are considered built-in. Thus to execute external libraries, the c-binding functions also have to be implemented in the architecture. Finally, catching up with the software development speed is a big challenge.

3.2 In-FPGA implementation of hot functions

Another option can be to implement hot functions in the FPGA. This strategy is commonly used with GPU's libraries that need heavy computations, like TensorFlow and OpenCV, to implement operations like convolution and matrix multiplication in the GPU. The same could be done for the FPGA, picking a widely used library like NumPy and implementing the hot functions inside the FPGA.

Advantages—We can expect a small speedup compared to the GPU due to the specific architecture since GPU also has special hardware for this but with the scalability mindset.

Drawbacks—It would be hard to compete with GPUs in scale since implementing the same hardware would take a lot more space in the FPGA, and generally cannot reach the same clock frequency. Also, GPUs can switch kernels a lot faster than an FPGA can reprogram.

3.3 Transpiler

The last approach would be to write a transpiler for Python code to some hardware description language like Verilog or VHDL to identify patterns in the code that could be written in the FPGA.

Advantages—Reduction in the execution time for some parts of the code.

Drawbacks—Writing a good transpiler is a challenger since we need to identify patterns worth putting in the FPGA, considering time wasted with data transmission. Also, creating the hooks to the original to switch between CPU and FPGA can create some overhead [9]. Another difficulty is to ensure that new bitstreams are not needed to be created on the fly in circumstances that involve recompilation (e.g., due to changing array shapes).

4 CONCLUSION

Python is a powerful high-level language with a rich ecosystem of APIs to scientific computation and artificial intelligence libraries. It offers then a "natural" way to gather software and hardware developers in the design of the core, low-level, optimized libraries, that would be used at high-level in pure Python. This would facilitate co-design of hybrid hardware-software architectures, going to a higher-level synthesis than that is currently available on the market.

There are several possibilities to implement Python function on an FPGA to offer the high-level synthesis. Each strategy has its advantages and disadvantages, and the choice depends on the project's restrictions. For instance, the strategy of implementing a Python processor can yield speedups as high as 200× as shown in the work of Fumero et al. [8] but can be very time-consuming and complex. Hybrid strategies such as implementing only the hot functions can be more straightforward and still have significant speedups; in the work of Skalicky et al. [16] they achieved 39× speedup besides showing that the overheads were minimal. Finally, the transpiler strategy is probably the one that can provide the highest speedups in theory since it is specialized hardware for the entire application. However, writing the transpiler is a big challenge.

REFERENCES

[1] Truls Asheim, Kenneth Skovhede, and Brian Vinter. 2016. VHDL Generation From Python Synchronous Message Exchange Networks. In *Proceedings of Communicating Process Architectures 2016*, Vol. 38. Open Channel Publishing Ltd, Copenhagen, Denmark.

[2] Christiaan Baaij, Matthijs Kooijman, Jan Kuper, Arjan Boeijink, and Marco Gerards. 2010. ClaSH: Structural descriptions of synchronous hardware using Haskell. In *Proceedings of the 13th Euromicro Conference on Digital System Design: Architectures, Methods and Tools*. IEEE, Lille, France, 714–721. https://doi.org/10.1109/DSD.2010.21

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: Constructing hardware in a Scala embedded language. In *Proceedings of the 49th Design Automation Conference, 49th DAC*. IEEE, San Francisco, CA, USA, 1216–1225. https://doi.org/10.1145/2228360.2228584

[4] Per Bjesse, Koen Claessen, Mary Sheeran, and Satnam Singh. 1998. Lava: hardware design in Haskell. In *Proceedings of the third ACM SIGPLAN international conference on Functional programming, ICFP '98*. Association for

Computing Machinery (ACM), New York, New York, USA, 174–184. https://doi.org/10.1145/289423.289440

[5] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen D. Brown, and Jason H. Anderson. 2013. LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *Transactions on Embedded Computing Systems* 13 (9 2013), 1–27. Issue 2. https://doi.org/10.1145/2514740

[6] Florent De Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design and Test of Computers* 28 (7 2011), 18–27. Issue 4. https://doi.org/10.1109/MDT.2011.44

[7] Norbert Feurle. 2012. Python Hardware Processor [MyHDL]. http://old.myhdl.org/doku.php/projects:python_hardware_processor

[8] Juan Fumero, Athanasios Stratikopoulos, and Christos Kotselidis. 2020. Running Parallel Bytecode Interpreters on Heterogeneous Hardware. In *Conference Companion of the 4th International Conference on Art, Science, and Engineering of Programming* (Porto, Portugal) *('20)*. Association for Computing Machinery, New York, NY, USA, 31–35. https://doi.org/10.1145/3397537.3397563

[9] Daniel Granhão and João Canas Ferreira. 2021. Transparent Control Flow Transfer between CPU and Accelerators for HPC. *Electronics* 10, 4 (2021), 406. https://doi.org/10.3390/electronics10040406

[10] David Greaves and Satnam Singh. 2008. Kiwi: Synthesis of FPGA circuits from parallel programs. In *Proceedings of the 16th IEEE Symposium on Field-Programmable Custom Computing Machines, FCCM'08*. IEEE, Stanford, CA, USA, 3–12. https://doi.org/10.1109/FCCM.2008.46

[11] Sitao Huang, Kun Wu, Hyunmin Jeong, Chengyue Wang, Deming Chen, and Wen mei Hwu. 2021. PyLog: An Algorithm-Centric Python-Based FPGA Programming and Synthesis Flow. In *Proceedings of FPGA '21, the 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. ACM, New York, NY, USA, 227–228. https://doi.org/10.1145/3431920.3439478

[12] Shunning Jiang, Peitian Pan, Yanghui Ou, and Christopher Batten. 2020. PyMTL3: A Python Framework for Open-Source Hardware Modeling, Generation, Simulation, and Verification. *IEEE Micro* 40 (7 2020), 58–66. Issue 4. https://doi.org/10.1109/MM.2020.2997638

[13] Derek Lockhart, Gary Zibrat, and Christopher Batten. 2014. PyMTL: A Unified Framework for Vertically Integrated Computer Architecture Research. In *Proceedings of the 47th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO-47*. IEEE, Cambridge, UK, 280–292. https://doi.org/10.1109/MICRO.2014.50

[14] Denis Navarro, Oscar Lucia, Luis A. Barragan, Isidro Urriza, and Oscar Jimenez. 2013. High-level synthesis for accelerating the fpga implementation of computationally demanding control algorithms for power converters. *IEEE Transactions on Industrial Informatics* 9 (2013), 1371–1379. Issue 3. https://doi.org/10.1109/TII.2013.2239302

[15] Daniel H. Noronha, Bahar Salehpour, and Steven J. E. Wilton. 2018. LeFlow: Enabling Flexible FPGA High-Level Synthesis of Tensorflow Deep Neural Networks. *ArXiv e-prints* (7 2018), 46–53. http://arxiv.org/abs/1807.05317

[16] S. Skalicky, J. Monson, A. Schmidt, and M. French. 2018. Hot Spicy: Improving Productivity with Python and HLS for FPGAs. In *2018 IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)*. IEEE, Boulder, CO, USA, 85–92. https://doi.org/10.1109/FCCM.2018.00022