Towards Higher-Level Synthesis and Co-design

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. In this paper, the direction towards even higher-level synthesis is promoted. Existing HLS frameworks 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.

CCS Concepts: • Hardware → Hardware accelerators; Hardware description languages and compilation.

Keywords: high-level synthesis, framework, FPGA, Python.

ACM Reference Format:

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.

In this paper, we promote the idea to go further in the concept of high-level synthesis and co-design. 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 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 HLS frameworks
Many frameworks dedicated to high-level synthesis have been designed for more than twenty years. Here below follows a selection of such 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 [13], enabling C and C++;
- **2013** LegUp [5], built in C;
- **2014** PyMTL [11, 12], written 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 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...

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

LATTE ’21, April 15, 2021,
© 2021 Association for Computing Machinery.
dedicated to artificial intelligence, e.g., the interface with TensorFlow. Therefore, the remaining text of this article will focus on Python.

3 Strategies for Python implementation on FPGAs

Some of the possible strategies for executing Python code in the FPGA are direct execution of the Python bytecode [7, 8], writing hot functions in the FPGA, i.e., commonly executed functions [9, 14] and writing a transpiler [1].

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:** 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 doable. The problem arises 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.

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 NumPy 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:** The disadvantage is that 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.

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 challenge 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].

4 Conclusion

As discussed in previous sections, 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. [14] 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


