

High-level Synthesis using the Julia Language

Benjamin Biggs
Imperial College London
UK

Eric C. Kerrigan
Imperial College London
UK

Ian McInerney
The University of Manchester
UK

George A. Constantinides
Imperial College London
UK

ABSTRACT

The growing proliferation of FPGAs and High-level Synthesis (HLS) tools has led to a large interest in designing hardware accelerators for complex operations and algorithms. However, existing HLS toolflows typically require a significant amount of user knowledge or training to be effective in both industrial and research applications. In this paper, we propose using the Julia language as the basis for an HLS tool. The Julia HLS tool aims to decrease the barrier to entry for hardware acceleration by taking advantage of the readability of the Julia language and by allowing the use of the existing large library of standard mathematical functions written in Julia. We present a prototype Julia HLS tool, written in Julia, that transforms Julia code to VHDL. We highlight how features of Julia and its compiler simplified the creation of this tool, and we discuss potential directions for future work.

KEYWORDS

FPGA, high-level synthesis (HLS), Julia language

1 INTRODUCTION

There is a growing need for high throughput, low latency, and energy efficient compute platforms, such as Field Programmable Gate Arrays (FPGAs), by a wide selection of tasks, including machine learning and real-time control. These platforms often have a high barrier to entry for software developers and researchers who are unfamiliar with digital hardware development, creating what is known as the “two-language” problem. The “two-language” problem presents the typical hardware accelerator development workflow as first having a researcher develop and prototype the application algorithms in a high-level language, such as Python or MATLAB, that is simpler to code and is mathematics-friendly. Then the algorithm is passed to an engineer who translates it into a lower-level language, such as C/C++/VHDL/Verilog, to achieve the desired performance targets.

The Julia language [5, 6] aims to solve this “two-language” problem by providing a higher-level and human-readable language that compiles into performant machine code for CPUs. Extensions to Julia have also been developed to compile normal Julia code to

accelerator hardware such as GPUs [9], distributed processors [10], and TPUs [4]

The goal of High-level Synthesis (HLS) tools for FPGAs is to solve the “two-language” problem by allowing FPGAs to be programmed using higher-level languages instead of VHDL/Verilog. However, the more mature toolflows like LegUp [1] or Vivado only move the accelerator implementation from VHDL/Verilog into languages like C and C++. Prototype toolflows have been made using higher-level languages, such as Python [2, 8], but these usually require the creation of separate compilers/transpilers to translate the chosen language into an FPGA design.

In this paper, we propose to use Julia as a high-level language for FPGA accelerators, and that the HLS toolflow should be written in Julia as well. We start in Section 2 by describing the features Julia has that makes it an ideal language for FPGA HLS, and present a brief report on a prototype Julia HLS tool in Sections 3 and 4.

2 ADVANTAGES OF THE JULIA LANGUAGE

Julia was developed as a language for scientific computing that sought to provide both ease of programming and performant low-level machine code. The Julia ecosystem contains many mathematical packages, and unlike other higher-level languages, many of these packages are written in pure Julia instead of being a thin wrapper over an underlying C/C++ library. This means a Julia HLS toolflow could recurse into the packages and generate hardware for them using the HLS compiler instead of having to provide and substitute in hand-written function blocks.

The Julia compiler is also mainly written in Julia, and provides several places to hook into the compilation flow to modify/observe the compilation. This means that an HLS toolflow for Julia that is written in Julia can benefit by reusing existing compiler stages and optimisation passes used for the CPU compiler. This also allows for more easier CPU simulation of the desired algorithm, since the initial stages of the compiler flow can be shared between the CPU and HLS tools. Three other design features of Julia that provide benefits for an HLS toolflow are *built-in data-flow type inference*, *multiple dispatch for method specialisation*, and *extensive metaprogramming features*.

2.1 Data-flow type inference

In Julia, explicit type declarations of variables are not required for the code to successfully run because Julia executes a data-flow type inference pass in its compiler to infer these types. The type inference pass tries to produce “type-stable” code, where all the types in the code are known explicitly at compile time, by combining any known type information on a variable alongside a type lattice to propagate

Permission to make digital or hard copies of part or all 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 third-party components of this work must be honored. For all other uses, contact the owner/author(s).

LATTE '22, March 1, 2022, Lausanne, Switzerland

© 2022 Copyright held by the owner/author(s).

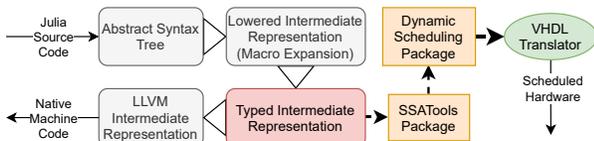


Figure 1: Overview of the Julia compiler & HLS flow.

types to every variable. An HLS toolflow can then utilize the typed Intermediate Representation (IR) after the type inference compiler pass when generating the hardware.

2.2 Multiple dispatch

Julia utilizes multiple dispatch, where the compiler and/or run-time environment examines the types of all function inputs, and then chooses a version of the function that has been written for the specified input types. Using multiple dispatch allows the creation of more efficient and readable code by allowing developers to write functions that contain the generic operations, while also providing functions optimised for the provided input types. Multiple dispatch can also be used for built-in operators, like “+”, allowing the HLS toolflow to provide FPGA-specific methods for the basic operators that can be automatically chosen by the Julia compiler when using the HLS toolflow, eliminating the need to modify the actual application source code to reference the HLS-specific basic operators instead of the normal CPU operators.

2.3 Meta-programming

Julia has an extensive metaprogramming capability provided in the language, allowing Julia code to modify and transform other Julia code before it is compiled. This is accomplished by writing Julia macros, which can be attached to variables, functions and control flow statements. Macros have direct access to the Abstract Syntax Tree (AST) for the associated code blocks and are evaluated before the lowered IR is generated. The Julia AST includes information such as function calls, loops and conditional statements, allowing the macros to operate directly on the control flow operators. HLS analysis and optimization passes can be implemented using Julia macros to modify the AST and interact with the control flow directly, instead of having to infer the original control flow from the LLVM basic blocks and phi-nodes.

3 TRANSFORMING JULIA INTO HARDWARE

The Julia HLS toolflow takes a subset of the Julia source code and transforms it into synthesizable HDL that can be tested in simulation. The flow is contained inside two Julia packages, SSATools.jl and DynamicScheduling.jl, that convert the Julia code into a Control Data-Flow Graph (CDFG) made from elastic components detailed in the Dynamics tool [7] which then generates VHDL for synthesis with Vivado HLS, as shown in Figure 1.

3.1 Initial Experiments

The dynamic scheduling in Dynamics formed the baseline for the initial results of the toolflow. The testcases used were chosen to test simple programmatic constructs like conditional statements and looping as well as the decomposition of mathematical functions by the Julia compiler.

Program	Basic Blocks		Components	
	Julia	C++	Julia	C++
if_else	5	3	41	32
power	4	4	61	64
newton_raphson	10	6	225	147

Table 1: Synthesis results for the Julia HLS toolflow.

The first testcase was an if-else conditional with three return statements based on the computation of the two input variables. This was chosen to test the processing of branching in the SSATools construction of the CDFG. Another testcase was a loop implementation of the power function, where the first input is raised to the power of the second through repeated multiplication in a while loop. The Julia compiler decomposes the loop into several basic blocks as part of the typed IR, and the CDFG generation determines the dependencies of the variables within those blocks. The final testcase implemented the Newton-Raphson method for determining the roots of a simple polynomial. This combined mathematical operations, conditionals and looping to fully explore the current version of the toolflow.

The results of these testcases are outlined in Table 1, and are compared to the Dynamics baseline for the same testcases implemented in C++. The number of components directly corresponds to FPGA resource usage, since the programs are dynamically scheduled.

4 EVALUATION AND FUTURE WORK

The difference in resources between the Julia HLS toolflow and Dynamics is due to the number of basic blocks generated for the IR. The Dynamics flow initially had more basic blocks in the LLVM IR in each case, but the LLVM optimization passes consolidated blocks together. Equivalent versions of these passes should be created for the Julia flow. The main limitation of the current toolflow is the lack of data memory integration. Adding memory support requires the extraction of additional information from the typed IR, as well as component support for handling memory accesses on the FPGA. Additionally, static scheduling for the operations should be implemented to allow for more resource-efficient designs. Finally, Julia is designed to be an open-source language, so the HDL output of our tool should be usable in open-source tools like Yosys [11] as well as closed-source tools like Vivado. Any IP cores required should be generated using FloPoCo [3] or other open-source HDL libraries.

5 CONCLUSION

The aim of this work was to detail the key features of the Julia language that make it a powerful and adaptable language for the generation of custom hardware designs for FPGAs. The higher-level nature of the language makes complex mathematical functions and algorithms easier to represent while also generating optimal low-level machine code. We have discussed several of the benefits the language provides, and showed our first attempt at designing a Julia-based toolflow that is able to translate a subset of the Julia language’s typed IR into a CDFG for VHDL code generation. This work forms the basis for an array of future work, including the implementation of memory support and alternative scheduling methods.

REFERENCES

- [1] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Tomasz Czajkowski, Stephen Brown, and Jason Anderson. 2013. LegUp: An Open-Source High-Level Synthesis Tool for FPGA-Based Processor/Accelerator Systems. *ACM Transactions on Embedded Computing Systems* 13, 2 (2013), 1–27. <https://doi.org/10.1145/2514740>
- [2] Radosław Cieszewski, Krzysztof Pozniak, and Ryszard Romaniuk. 2014. Python based high-level synthesis compiler. In *Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2014*, Ryszard S. Romaniuk (Ed.), Vol. 9290. International Society for Optics and Photonics, SPIE, 981–988. <https://doi.org/10.1117/12.2075988>
- [3] Florent Dinechin and Bogdan Pasca. 2011. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design and Test of Computers* 28, 4 (2011), 18–27. <https://doi.org/10.1109/MDT.2011.44>
- [4] Keno Fischer and Elliot Saba. 2018. Automatic Full Compilation of Julia Programs and ML Models to Cloud TPUs. *arXiv* 1810.09868 (Oct. 2018).
- [5] Nick Heath. 2018. Why is programming language Julia growing so fast and where is it going next? <https://www.techrepublic.com/article/julia-whats-next-for-pythons-fast-growing-programming-language-rival/> Accessed: 16/02/2022.
- [6] Jeff Bezanson, Alan Edelman, Stefan Karpinski, Viral Shah. 2015. Julia: A Fresh Approach to Numerical Computing. *SIAM Rev.* 59, 1 (2015), 65–98. <https://doi.org/10.1137/14100671>
- [7] Lana Josipović, Radhika Ghosal, and Paolo Jenne. 2018. Dynamically Scheduled High-level Synthesis. In *Proceedings of 2018 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA 2018)*. ACM, Monterey, CA, USA, 127–136. <https://doi.org/10.1145/3174243.3174264>
- [8] Alexandre Quenon and Vitor Ramos. 2021. Towards Higher-Level Synthesis and Co-design with Python. In *Workshop on Languages, Tools, and Techniques for Accelerator Design (LATTE'21)*. Virtual, World. <https://capra.cs.cornell.edu/latte21/paper/20.pdf>
- [9] Tim Besard, Christophe Foket, Bjorn De Sutter. 2019. Effective Extensible Programming: Unleashing Julia on GPUs. *IEEE Transactions on Parallel and Distributed Systems* 30, 4 (2019), 827–841. <https://doi.org/10.1109/TPDS.2018.2872064>
- [10] Tim Besard, Valentin Churavy, Alan Edelman, Bjorn De Sutter. 2019. Rapid software prototyping for heterogeneous and distributed platforms. *Advances in Engineering Software* 132 (2019), 29–46. <https://doi.org/10.1016/j.advengsoft.2019.02.002>
- [11] YosysHQ. [n.d.]. Yosys Open Synthesis Suite. <https://github.com/YosysHQ/yosys>. Accessed: 04/01/2020.