Received: 31 May 2021 | Revised: 26 September 2021 | Accepted: 27 September 2021

DOI: 10.1002/spe.3043

SPECIAL ISSUE PAPER

WILEY

Generation of logic designs for efficiently solving ordinary differential equations on field programmable gate arrays

Silas Bartel | **Matthias Korch**

Department of Computer Science, University of Bayreuth, Bayreuth, Germany

Correspondence
Silas Bartel, Department of Computer Science, University of Bayreuth, Bayreuth, Germany.
Email: silas.bartel@uni-bayreuth.de

Funding information
German Research Foundation (DFG), Grant/Award Number: KO 2252/3-2

Abstract
Ordinary differential equations can be used to describe simulation models. As such, solving these equations is an important task in the high performance computing (HPC) domain. Field programmable gate arrays (FPGAs) are a promising platform, expected to be usable as efficient accelerators for such computations. While the use of hardware description languages (HDLs) can produce very efficient logic designs, their unique concept is hard to adopt for scientists and software engineers. High-level synthesis (HLS) tools promise faster development, but bear the risk of lower performance and increased resource consumption of the final design. But even when using HLS tools the user still requires specialized knowledge about FPGAs and circuit design. In order to reach a wide adoption of FPGAs in HPC applications, a need for simple to use tools which enable performant designs was identified. This article proposes a framework that is able to automatically generate specific and optimized solver logic from easy to handle configuration files. No manual development, nor special FPGA or programming knowledge is required. To measure the capability of the proposed tool, the performance was evaluated for different solver methods and compared with an alternative hand optimized HLS implementation. The logic generated by this improved approach is up to 43 times faster than its hand optimized HLS counterpart, depending on the solution method.

KEYWORDS
code generation, FPGA, hardware accelerators, ODE, reconfigurable architectures, Runge–Kutta method

1 | INTRODUCTION

Systems of ordinary differential equations (ODEs) are used to describe a wide variety of time-dependent processes in nature and engineering. However, since no explicit solution can be given for many of these systems, the use of numerical methods is necessary. Their efficient implementation on high performance computing (HPC) architectures is a subject of ongoing research. In addition to the use of CPUs, the use of graphics processing units (GPUs) as hardware accelerators has become well established in the field of HPC. In recent years, field programmable gate arrays (FPGAs) have also attracted attention in datacenter applications[1] and the HPC area.[2] This is driven by changes in requirements joint with the market introduction and greater distribution of FPGA-based PCI accelerator cards. FPGAs, which can be freely configured by the user, have been present in electrotechnical applications for years. Due to the recent attention on FPGAs in computer science, their efficient use in the HPC domain is currently the focus of intense research.

Traditionally, special languages, called hardware description languages (HDLs), have been used to develop logic designs. However, the requirements for the development of logic circuits, where a completely parallel description of a logic design takes place, differ strongly from the requirements of the classical software development concentrated around algorithms. This results in fundamental conceptual differences between programming languages and HDLs. In order to create efficient logic designs, special knowledge about circuit design and FPGAs is required. As a result, programmers and scientists do not find it easy to adapt FPGAs into their workflow.[3]

A key research question in the HPC domain is how to facilitate the use of FPGAs.[4] In the past, several solutions based on high-level synthesis (HLS) have been proposed.[5-9] During HLS, an algorithmic description usually in the form of software source code is automatically interpreted and a hardware circuit that reproduces the same behavior is generated. Despite a large research effort, this complex process still requires a lot of expertise in FPGAs and manual optimization work from the user.

This article proposes an alternative approach. Instead of pursuing the development of a general purpose translation tool which converts generic imperative or functional source programs to hardware, the implementation of application specific and thus optimized generation tools is suggested. FPGA specific expertise is only required once for the development of the tools. Suitable application-specific configuration options completely eliminate the individual development effort by the user that is required for the HLS approach.

In this work, a logic circuit generation tool for efficiently solving systems of ODEs on FPGAs is introduced: rtlode.[10] This tool demonstrates the main advantages of this method. All necessary settings like the description of the differential equation system to be solved as well as the choice of the solution method or the numerical number representation to be used are done in a simple configuration file. Thus, the individual development effort for the implementation of a solver is completely eliminated. This allows the use of FPGA accelerators even for users without FPGA or programming specific knowledge. Due to the restriction to a limited application area, the necessary logic can be implemented in HDLs. This allows the generation of efficient and automatically optimized logic designs. For evaluation, the performance of the generated circuits are compared with results achieved by a HLS approach.[5] In summary, the key contributions of this work are:

- An enhanced approach to tool development which enables efficient use of FPGAs without requiring specific knowledge of FPGAs circuit design or hardware and software development.
- rtlode: An open source tool that can automatically generate logic circuits from a user-friendly configuration file for efficiently solving systems of ODEs on FPGAs.[10]
- A performance evaluation of the proposed tool for different solution methods and internal number representations, as well as a comparison to a hand-optimized HLS-based alternative.[5]

In the scope of this work, we restrict ourselves to the support of explicit Runge–Kutta (RK) methods as solution methods for ODE systems. An extension to other mathematical solution methods is possible. Currently, the only supported hardware is the Arria GX 10 accelerator card from Intel (see Section 5.1). The use of a different accelerator card or support for application in an embedded system requires adjustments.

In the following, a short overview of the structure of this article and the content of the individual sections is given. In the next section the relevant mathematical background is presented. Additionally, the basics of circuit design necessary for understanding the work are discussed. In Section 3 the current state of the art in research is covered. Relevant papers are presented and discussed. Afterwards, Sections 4 and 5 discuss the concept and the implementation of our approach. During the explanation of the concept, a comparison to the concepts of the papers presented in Section 3 is made. In addition to describing the structure of the generated circuits, it is explicitly explained how the necessary calculations are optimized for and implemented on the FPGA. The user interface relevant to the operator is also discussed in this section. In Section 6 the designs generated by rtlode are evaluated and compared with an alternative approach. The results achieved are discussed in the following Section 7. As part of this, potential improvements, a comparison to conventional HPC architectures, portability as well as existing limitations are considered. At the end, Section 8 summarizes the findings of this article and gives an outlook on possible further research topics.

2 | BACKGROUND

2.1 | Mathematical background

In this article, explicit solution methods for initial value problems (IVPs) of ODE systems are considered. For an unknown solution function $y = (y_1, \dots, y_n) : \mathbb{R} \to \mathbb{R}^n$, the corresponding right-hand side $f = (f_1, \dots, f_n) : \mathbb{R} \times \mathbb{R}^n \to \mathbb{R}^n$ and an initial value at location $x_0 \in \mathbb{R}$ in the form of $y_0 = (y_{1,0}, \dots, y_{n,0}) \in \mathbb{R}^n$, the IVP can be stated as follows:

$$\begin{aligned} y'(x) &= f(x, y(x)), \\ y(x_0) &= y_0. \end{aligned} \tag{1}$$

Runge–Kutta (RK) methods are a family of solution techniques in which the function $f(x, y)$ is evaluated at one or more predefined points within the interval $[x_k, x_{k+1}]$ of time step k, depending on the variant. In each time step, starting with the input approximation $\eta_k \approx y(x_k)$, an s-stage RK method performs s evaluations of the function $f(x, y)$ to compute a new approximation value $\eta_{k+1} \approx y(x_{k+1})$. First, given a step size h, s stage vectors $v_1, \dots, v_s \in \mathbb{R}^n$ are computed according to:

$$v_j = f\left(x_k + h \cdot c_j, \eta_k + h \cdot \sum_{l=1}^{s} a_{jl} v_l\right) \quad \text{for } j = 1, \dots, s. \tag{2}$$

After calculating the stage vectors, the output approximation of the time step can be calculated as follows:

$$\eta_{k+1} = \eta_k + h \cdot \sum_{j=1}^{s} b_j v_j. \tag{3}$$

The coefficients $c = (c_1, \dots, c_s) \in \mathbb{R}^s, A = (a_{jl} \in \mathbb{R}^{s \times s})$ as well as $b = (b_1, \dots, b_s) \in \mathbb{R}^s$ depend on the individually used RK method. The coefficients of a method are usually specified in the so-called RK tableau (Butcher array):

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^{\mathrm{T}} \end{array} = \begin{array}{c|cccc} c_1 & a_{11} & a_{12} & \dots & a_{1s} \\ c_2 & a_{21} & a_{22} & \dots & a_{2s} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ c_s & a_{s1} & a_{s2} & \dots & a_{ss} \\ \hline & b_1 & b_2 & \dots & b_s \end{array}. \tag{4}$$

Equation (4) represents the general form of a RK tableau, which contains all the information needed to implement a particular RK method. RK methods are classified as explicit or implicit methods, which differ strongly in their numerical properties and thus in their use case. For more in-depth information, please refer to the monograph of Hairer et al.[11] The present work limits itself to explicit RK methods. For these methods, the entries a_{jl} are 0 for all $l \geq j$, that is, the RK tableau contains only zeros above the diagonal.

2.2 | Levels of abstraction in digital circuit design

Several abstraction levels can be distinguished in digital circuit design, these are shown in Figure 1 and are described below.

2.2.1 | Layout level

The layout level is the lowest level of abstraction used to describe a circuit. In addition to the interconnection of the individual elements, the exact spatial implementation in hardware is also specified.

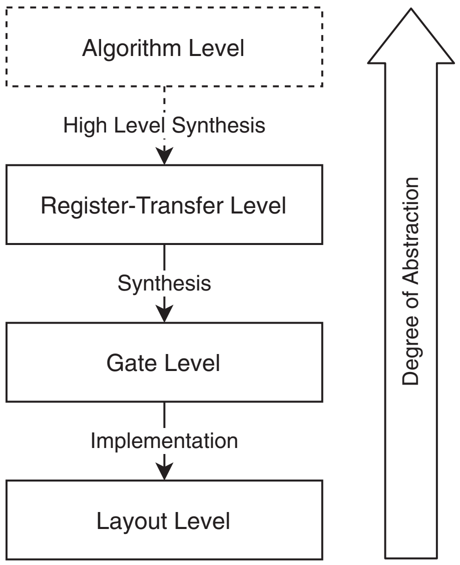

FIGURE 1 Levels of abstraction in digital circuit design

2.2.2 | Gate level

In the gate level, the design is described with the help of a netlist in which the interconnection of individual logic gates (and, or, xor, etc.) and memory blocks is defined. Netlists are usually created automatically. Through the process of implementation, a description on the layout level can be generated from a netlist. The position of the elements and the interconnection is defined manually or automatically.

2.2.3 | Register-transfer level

The register-transfer level (RTL) is a slightly abstracted and well readable, mostly textual but sometimes also graphical description of the signal flow of a design, from which a netlist can be generated by synthesis. In contrast to the gate level, the description of several bit signals together as one multibit signal is possible. Multiple instantiation of logic combined into modules is also supported. So-called hardware description language (HDLs) are used for the textual description. In contrast to classical programming languages, which are used to describe an algorithm and thus a sequential sequence of actions, HDLs define a fixed series of operations through which data pass and are thus processed.

2.2.4 | Algorithm level

In most cases, a solution strategy for a problem is defined as an algorithm, in other words a sequential series of instructions. Such a description is usually formally written in a programming language. In order to lower the entry barrier into the development of digital circuits as well as to shorten the development time, research has been conducted for years on the improvement of the so-called high-level synthesis (HLS). The HLS automatically generates hardware descriptions on the RTL which implement a given algorithm. The input to the HLS is given using libraries or frameworks (e.g., OpenCL[12]), generally in ANSI C/C++ or SystemC. The output of the HLS is usually in Verilog or VHDL, which are HDLs.

The fundamental conceptual difference between a sequential instruction-based algorithm description and a fully parallel description of hardware operations poses a major challenge even for modern HLS software. This is caused by the lack of information about possible parallelism, timing, and implementation details. Thus, automating the translation from a description on the algorithm level to a description on the RTL is significantly more complex than automating translations between lower abstraction levels. A potentially worse performance or resource utilization of the resulting design is opposed to a significantly lower development effort.[4]

3 | RELATED WORK

Osana et al.[13] propose a framework for simulating biochemical cell processes which are modeled by ODEs on FPGAs. The choice of FPGAs as platform is highlighted as a compact and cost-effective solution. Due to the narrow application field and thus the low diversity of relevant ODEs, the authors were able to implement by hand a solver for each function at the RTL. The proposed framework composes the final FPGA designs from the handwritten solvers and other static components. The use of advanced solution methods, such as Heun or RK, instead of the Euler method used in that article, is proposed in the outlook.

Fasih et al.[14] also address the question of how to solve ODE systems efficiently on FPGAs. In their work, they demonstrate a method for solving higher-order ODEs by solving the Rössler equations. They derive their approach from classical analog computers and try to replicate the necessary calculations as directly as possible in hardware. The classical Euler method is used for integration, even though it is not mentioned in the article. A discussion of other numerical solution methods is not given. The sequential logic of the proposed solver works with a common clock signal. In each cycle of the clock, the complete function evaluation as well as one step of the Euler method is performed. The handling of more complex ODE systems in which the function evaluation cannot be completed in one clock cycle using only pure combinational logic, is not discussed. The authors recommend the use of a fixed-point representation, adapted to the desired and required accuracy, for the representation of all internal signals. This allows a saving of resources as well as faster computations in comparison to a floating-point representation. In choosing the implementation, the authors adopt the same approach as Osana et al.,[13] that is, the proposed solver is described manually in Verilog code. In the outlook, the automatic generation of solvers from a data flow description, given in textual or visual representation, is proposed.

Stamoulias et al.[5] propose a different methodology for solving ODEs on FPGAs. In contrast to Fasih et al.[14] and Osana et al.,[13] the proposed technique does not directly use a HDL. The authors propose the use of HLS, which is supposed to allow an easier adaptability to other ODE systems and solution methods. Within the scope of the work, eight solvers for the Lotka–Volterra equations were implemented by hand, which differ in the floating-point representation used (single and double) as well as the solution method: Euler, ModEuler, Heun, and SSPRK3. The authors compare the proposed solvers in detail regarding performance and resource utilization. There is no comparison to solvers described natively in a HDL. The written HLS code was optimized by hand with additional directives which instruct the HLS tool to pipeline the computations on the FPGA and therefore allowing parallel processing of multiple independent IVPs within one solver. The overall performance with parallel processing is further enhanced by combining multiple of these hand-implemented solvers in one design. In a comparison with a single-core CPU, a speed-up of up to 14 could be determined.

A fundamentally different approach is taken by Huang et al.[15] Instead of developing a specific solver description for each ODE system and the solution method used, as done by Osana et al.,[13] Fasih et al.,[14] as well as Stamoulias et al.,[5] this work proposes a special softcore processor for FPGAs that is optimized to efficiently solve ODE systems. The authors note that the significant human effort required to develop circuits on the RTL are an obstacle to wider adaptation of FPGAs. HLS tools, according to the authors, do not tolerate the high complexity of some physical models and generate circuits with higher resource requirements than necessary. Poorer performance is accepted by the authors in terms of avoiding the specific development effort required per ODE system and solution method. The differential equation processing element (DEPE), as the softcore processor was named, uses a fixed-point representation for all signals internally.

In addition to the above-mentioned papers, which are relevant due to the same use case (solution of ODE systems on FPGAs), the following works are relevant because they present and discuss generic strategies to simplify the usage of FPGAs while maintaining efficient implementations.

By providing a library of optimized reusable hardware components, De Matteis and his colleagues[9] hope to reduce the manual development work required when using HLS for FPGAs. In their work, they present *FBLAS*, an implementation of basic linear algebra subprograms (BLAS) optimized for FPGAs. The components of the library are implemented in OpenCL and translated by HLS. Users can combine the individual optimized components and integrate them into their own logic. As an alternative, a template based code generator can be used to create and combine the needed components. By using the library, much of the manual optimization work normally required by the user is eliminated.

Ruan et al.[16] found a mismatch between the GPU-like programming models implemented by common HLS methods and the streaming-like models that best matched the needs of most applications. For this reason, in their paper they present their own programming platform *ST-Accel* which implements an alternative stream based model. To reduce the latency, the memory model is adjusted and the communication between the FPGA and the host is changed to a message-based concept. The actual user logic is written in a C++-based language, compiled by HLS and embedded in the *ST-Accel* framework, which is written in Verilog.

The work by Koeplinger and his colleagues[17] criticizes the mix of hardware and software abstractions in common HLS tools. For users, this makes optimization work much more difficult. The underlying reason is that existing HLS tools are mostly based on software languages like C, OpenCL, or Matlab, which were designed as control flow languages to implement algorithms and are not suitable to fully describe logic circuits. To overcome this problem but provide higher abstraction than the common HDLs, they propose a new (domain-specific) language: *Spatial*. This language is designed from the ground up to describe calculations by a cleaner control flow more suitable to derive logic circuits. In contrast to the proposed approaches of De Matteis et al.[9] and Ruan et al.,[16] no complex HLS process takes place during translation into hardware. The associated compiler creates an internal intermediate data flow representation and generates code in Chisel[18] (a modern HDL) for the FPGA part and in C++ for the host.

Other approaches directly start with a data flow description of the application. An example of a generic data flow language is CAPH.[19] The approach of CAPH is based on a set of interconnected processing elements called *actors*, the behaviors of which must be described explicitly by the programmer.

Similarly to Ruan et al.,[16] the tool Stream Processor Generator (SPGen)[20,21] targets stream processing applications. Similar to our work on time-step-based ODE methods, stream processing involves a loop of recurring computations which are then applied to the sequence of input vectors. But in contrast to Ruan et al., SPGen uses configuration files that describe the data flow of the user logic of an iteration.

Another related research area is the solution of ODE systems on established platforms such as CPUs and GPUs. This article was partially inspired by the preceding work of Korch and Werner,[22] who propose an automatic code generation approach for explicit ODE methods on GPUs, which is based on a data flow representation of the method. However, while the preceding paper aims at locality optimizations to exploit the memory hierarchy of a GPU by different tiling schemes, the present work is focused on automatically creating efficient pipeline layouts for the FPGA.

4 | CONCEPTS

In this work, an alternative approach to efficiently solve differential equations on FPGAs is proposed: the automated generation of logic circuits on the RTL. The major hurdle in using FPGAs as a computational accelerator is the high manual development effort required for each new differential equation system and solution method. Even when using HLS tools, as proposed by Stamoulias et al.,[5] manual implementation and subsequent optimization of algorithms for use on FPGAs is necessary and thus requires FPGA-specific knowledge in addition to programming knowledge. In the concept proposed here, after providing a configuration file in which the system of equations and desired solution method are specified, an optimized hardware design of a solver is automatically generated. No programming or FPGA knowledge is required by the user. This approach of specialized application-specific generators can be transferred to other numerical problems.

Figure 2 illustrates the workflows of different approaches presented in the related work section and in this work. Both Osana et al.[13] and Fasih et al.[14] take an approach where the user implements the logic for solving the differential equations by hand in a HDL on the RTL (A). This requires that the user has both domain and FPGA specific knowledge. In the approach of Stamoulias et al.,[5] illustrated in (B), the user manually implements an algorithm using a programming language. Using HLS tools, a logic description on the RTL which describes hardware performing the same task as the described algorithm is automatically generated. While it is possible to create very efficient implementations using HLS,[23] a lot of manual and platform-specific optimization has to be done, for which it is necessary that the user is not only a domain but also an FPGA expert. The concept presented in this article (C) also introduces another abstraction layer above the RTL. In this layer, the user, in this approach only a domain expert, specifies in a configuration file all the necessary information needed to automatically generate the solver. The software component responsible for generating RTL descriptions of the solver circuits from the configuration files, which is called logic generator in the following, was implemented once by an FPGA expert. Due to the very wide-ranging ambition to be able to translate algorithms into hardware and the accompanying necessary paradigm shift from a control flow model to a data flow model, HLS is an exceedingly complex process. The logic generator presented in this article, on the other hand, does not require this paradigm shift, all necessary logic is described using a HDL. Due to a restricted application domain (objective: generation of solvers for differential equations) a high degree of optimization is possible.

To take advantage of FPGAs, the developed generator generates the logic for all calculations in hardware as parallel as possible. Registers and internal random-access memory (RAM) are used to hold data locally and thus reduce expensive memory accesses. To find the best number representation of the internal numeric signals for his application, the

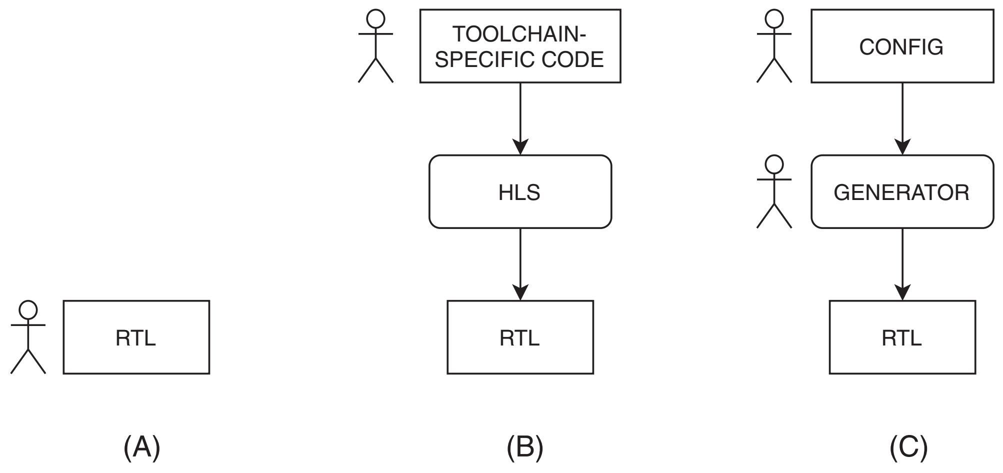

FIGURE 2 Workflow comparison: (A) manually written RTL code, (B) HLS approach, and (C) generated RTL code

user can choose between a single or double precision floating-point representation or a freely configurable fixed-point representation. All calculations are automatically optimized at the time of generation (see Section 5.4.3).

Similarly to Stamoulias et al.,[5] we consider the use case of solving not only a single, but also many identical ODE systems with different initial values efficiently in parallel. In the following, we refer to each such instance as an IVP. To achieve this, we automatically construct solver units as computing pipelines for the specific ODE system. This allows the logic to be used in every clock cycle. To further increase parallel performance, the circuits generated by the logic generator can contain several such independent solver units.

The software developed as part of this work was designed for use on an Intel Arria 10 GX Programmable Acceleration Card (see Section 5.1). For use on an accelerator card from another manufacturer or in embedded systems, adaptations are necessary. However, the underlying concept is universal.

4.1 | Structure of the logic circuits

Figure 3 shows the structure of a single, highly simplified solver unit. The arrows represent the data flow between the components. A solver unit consists of a number of stages defined by the chosen solver method. The solver unit shown in Figure 3 uses a three-stage RK method to solve a system of three ODEs (k, l, m). The individual stages are run through sequentially one after the other by the data. Within a stage, the respective stage vector (see Equation 2) is calculated for each component of the ODE system. A run through the solver and thus through all stages corresponds to one time step

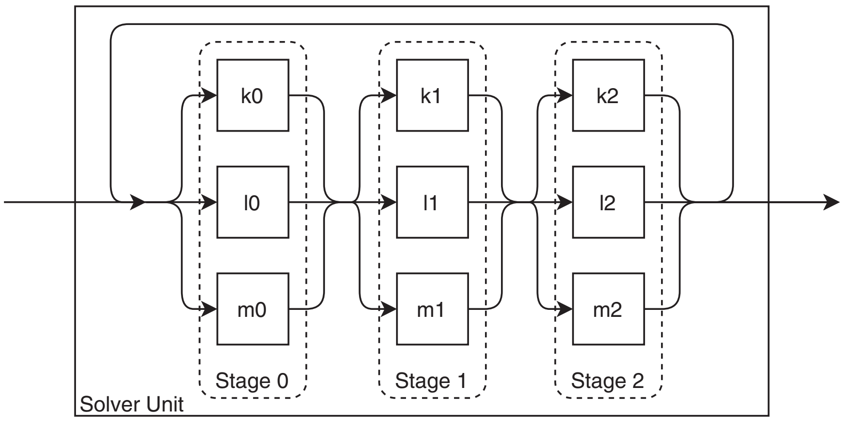

FIGURE 3 Structure of a solver unit

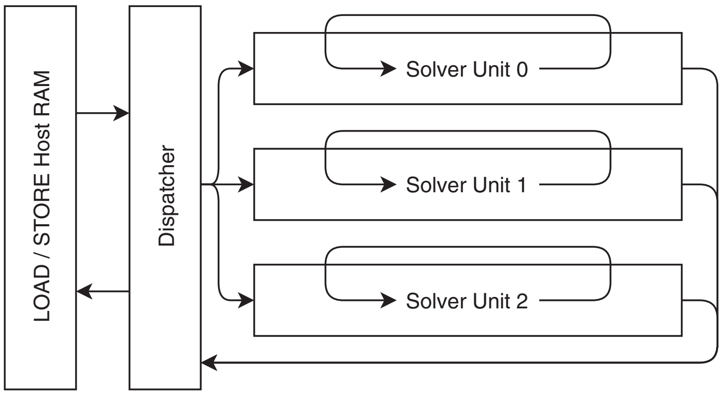

FIGURE 4 Overview of a solver circuit

of the RK method. A single IVP runs through a solver several times until the desired number of time steps has been completed.

To improve the parallel performance of the overall circuit, several solver units are usually used. Further logic components are necessary to distribute the IVPs internally and to exchange data with the CPU. Figure 4 shows an overview of a solver circuit consisting of three individual solvers and additional components for communication and distribution. The *host RAM handler* is responsible for loading and storing the IVPs as well as the results from and into the main memory of the CPU. An IVP consists of a data set with the start value x_0 of the independent variable, the initial values y_0 of the solution function as well as the desired step size h and the desired number of steps n. By specifying an identifier, the result can later be associated to a specific input data set (IVP). The *dispatcher* is responsible for assigning the data packets to the individual solver units as well as forwarding the results of the final step to the *host RAM handler*, which then transmits the specified identifier, the final value of the independent variable, and the calculated approximation values for each solved IVP back to the CPU.

4.2 | User interface

4.2.1 | Configuration

All required settings can be specified by the user in one or more configuration files. Therefore the user does not need any programming or FPGA-specific knowledge. Due to its simple and readable syntax, YAML[24] is used to define a configuration file. The ability to split the configuration files makes it easier to swap out and reuse individual parts of the configuration, such as the solution method. The solution method is configured under the key `method`. For this, the characteristic coefficients A, b, and c of the RK tableau (see Equation 4) must be specified. As described in the Introduction, we restrict ourselves to explicit RK methods in the context of this article. Listing 1 shows an example of the configuration of the Heun method, which is given by the following RK tableau:

$$\begin{array}{c|c} \mathbf{c} & A \\ \hline & \mathbf{b}^{\mathrm{T}} \end{array} = \begin{array}{c|cc} 0 & & \\ 1 & 1 & \\ \hline & 1/2 & 1/2 \end{array}. \tag{5}$$

```
method:
    A: [[],
        [1]]
    b: [0.5, 0.5]
    c: [0, 1]
```

Listing 1: Configuration of the RK method (here: Heun method)

When configuring the IVP (key: `problem`), only the definition of the differential equation system in an array under the key `components` is necessary. For this purpose the variables `x` (independent variable) and `y` (list of all y-values of the system) can be used. The initial values for x and y as well as the step size h and step count n can be specified optionally to be used as default values by the runtime. In Listing 2 the following ODE system, the Lotka–Volterra equations,[25] is defined:

$$\begin{aligned}
\frac{dy_0}{dt} &= f(y_0, y_1, t), \quad f(y_0, y_1, t) = 0.1 \cdot y_0 - 0.2 \cdot y_0 \cdot y_1, \\
\frac{dy_1}{dt} &= g(y_0, y_1, t), \quad g(y_0, y_1, t) = -0.2 \cdot y_1 + 0.4 \cdot y_0 \cdot y_1.
\end{aligned} \tag{6}$$

```
problem:
    components:
      - 0.1 * y[0] - 0.2 * y[0] * y[1]
      - -0.2 * y[1] + 0.4 * y[0] * y[1]
```

Listing 2: Configuration of the IVP

Additional configuration options can be used to influence the generated solver. The key *nbr_solver* can be used to specify how many parallel solver units should be generated. Under the key *numeric* the number representation to be used can be configured. Listing 3 shows some options as an example.

```
nbr_solver: 1

numeric:
    type: 'fixed'   # or 'floating'
    fixed_point_signed: True
    fixed_point_fraction_size: 41
    fixed_point_nonfraction_size : 12
#   floating_precision: 'single' or 'double'
```

Listing 3: Additional configuration options

4.2.2 | Generation

A command line tool is available for interacting with the logic generator and the runtime. To create a solver, the *build* command should be called with the configuration files as parameters. Listing 4 shows how to call rtlode to generate a solver for the Lotka–Volterra[25] IVP. It is assumed that the two configuration files given (`heun.yaml` and `predator-prey.yaml`) contain the contents of Listings 1 and 2. The call generates the logic circuit for the solver on the RTL and then triggers synthesis and implementation. The result, the finished bitstream for the solver and other information, is stored in a .slv file, a custom file format which is explained in Section 5.6.1.

```
>> rtlode build heun.yaml predator-prey.yaml
```

Listing 4: Call to generate a complete solver circuit

4.2.3 | Runtime

The same command line tool can be used to test a solver. To do that only the .slv file is needed, it contains all necessary information. Thus, generation and execution can also take place on different systems. Listing 5 shows an example of solving a single IVP. The data describing the IVP to solve are given by a JSON string. For more complex applications a Python library is provided to interact with the solver on the FPGA. An additional subcommand `benchmark` is also available, which performs a performance measurement for a solver. In addition to the time required to solve a single IVP, parallel solving of multiple problems is also automatically evaluated.

```
>> rtlode run heun_predator-prey.slv \
      --runtime_config='{x: 0, y: [0, 2], n: 60, h: 0.17}'
```

Listing 5: Command to start solving a single IVP

5 | IMPLEMENTATION

5.1 | Used hardware

For this work, an *Intel Programmable Acceleration Card with Intel Arria 10 GX 1150 FPGA*[26] was used. This PCIe expansion card contains the FPGA as well as memory and additional interfaces. The Arria 10 GX 1150 FPGA includes 1.15 million logical blocks, 65 MB internal RAM, and 3036 digital signal processing (DSP) blocks.[27] Two DSP blocks can be used with 27 bit fixed-point and single-precision floating-point numbers. By interconnecting multiple blocks, double-precision floating-point numbers can be used as well as more precise fixed-point numbers.

5.2 | Used software

In this work, MyHDL[28] is used as HDL. The open source library enables hardware description and verification on the RTL in Python. Python is known for its clear syntax and high development speed. The wide distribution of the language results in access to numerous tools and libraries. This enables a modern, software-like approach to circuit development. Powerful development environments can be used and module tests can be created for the hardware developed in MyHDL using the Python unit testing framework. When using MyHDL, no HLS is used. Python itself is a control flow language and thus is not suitable for describing hardware. Rather, MyHDL provides special language constructs in Python that are very close to classical HDLs such as Verilog or VHDL. Hardware described using these MyHDL constructs can be automatically transcribed to Verilog or VHDL, allowing easy integration into the standard circuit design workflow. In this approach, MyHDL is similar to Chisel,[18] the HDL used by Koeplinger et al..[17] In contrast to MyHDL, Chisel is based on Scala. A simulator provided as part of MyHDL allows efficient testing of the described logic. Additionally, MyHDL can be used together with a classical external simulator (e.g., *Icarus Verilog*[29]) for verification of the generated Verilog code.

Intel offers the open programmable acceleration engine (OPAE)[30] for its accelerator cards. As part of this extensive framework, in addition to tools for development as well as Linux drivers to access the connected FPGAs and libraries, Intel also provides interfaces for communication between FPGA and CPU. For these interfaces, in addition to software libraries in C and Python, the necessary circuits on the hardware side are also provided. A circuit designed for an accelerator card implementing the required interfaces is called an *accelerator function unit (AFU)*. The FPGAs used support partial reconfiguration at runtime. This allows a program on the CPU to flexibly load new AFUs for use on the FPGA. For synthesis and implementation of the circuits, the OPAE uses an installation of Quartus Prime.[31]

5.3 | Overview

An essential part of this article is the automatic generation of the individual solver circuits based on the configuration files as input. Figure 5 shows this generation process of the circuit descriptions with all intermediate steps. The labels of the dashed boxes indicate in which form the included representations are given. The generated parts are displayed with a dashed border. Parts which contain platform specific logic and therefore might have to be adapted when porting to other hardware are marked by an asterisk in the upper right corner.

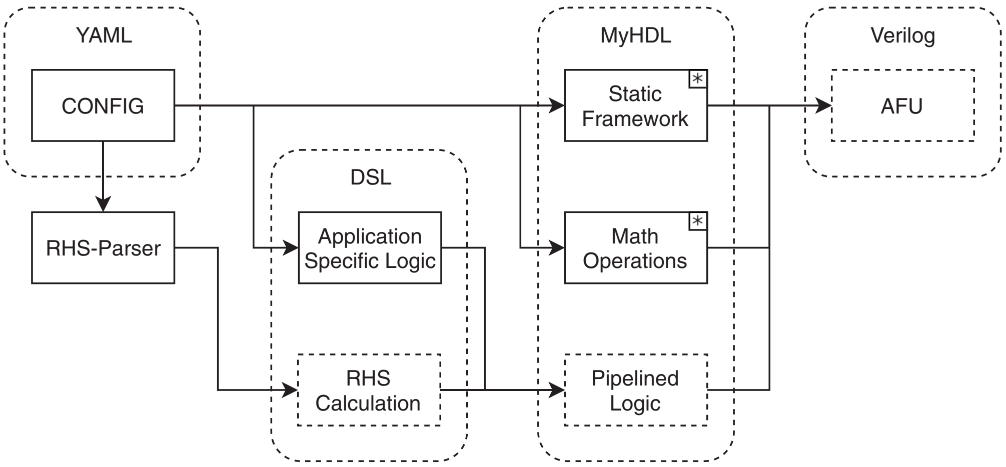

FIGURE 5 Overview generation process. Intermediate and generated parts are displayed with a dashed border. Parts containing platform-specific logic are marked with *

As input the process receives a user configuration file which, as already described in Section 4.2.1, is written in YAML. In addition to general parameters such as the internal number representation that affect the behavior of the static framework and math operations, it also contains the right-hand sides (RHSs) of the ordinary differential equations. In Section 5.5.1 the parser responsible for the translation is described. Its task is the conversion of the string representation of the RHSs from the configuration file into a further processable representation utilizing a domain-specific language (DSL) developed as part of this work.

This DSL, which can be used to simply describe calculations in form of a data flow, was also used to formulate the application specific logic. In the case of this article, these are the calculations of the RK methods as described in Section 5.5.2. The values in the configuration file such as the number of stages as well as the coefficients of the RK method to be used (specified by the Butcher tableau) are parameters of the application logic. In a subsequent step, the computations of the application logic together with the parsed RHSs can be automatically translated into a description of an optimized computation pipeline on the RTL. The approach for this is explained in Section 5.4. The automatically generated hardware description is only temporarily available in the system memory during the generation process in the form of an interconnected MyHDL data structure.

The interface to the CPU (see Section 5.6.2) as well as the dispatcher that manages the single solver units are parts of the static framework. These are logic parts which do not contain any direct application logic. The same applies to the implementation of the mathematical arithmetic operators which are used for the generation of the pipelines by the DSL. However, the static framework as well as the implementations of the operators contain platform-specific details which may need to be adapted to support other hardware. The hardware descriptions generated by the DSL together with the static framework and the implementations of the mathematical operators form the complete design of the AFU and can be transcribed to Verilog using MyHDL's automatic conversion feature.

This Verilog file is the output of the generator. When using the build command as described in Section 4.2.2, synthesis and implementation is automatically initiated afterwards. The resulting bitstream is then stored in a .slv file (see Section 5.6.1) with additional information for later use with the runtime.

5.4 | Implementation of calculations

5.4.1 | Description of calculations using the DSL

To implement the necessary calculations on the FPGAs side, an embedded DSL was developed, which can be used to easily describe the calculations in the form of a data flow. This allows the numerical computations to then be further processed and translated into hardware in an automated fashion. A data flow can be modeled well as a graph. In this case, the nodes represent individual operations and the paths represent the interdependencies of the operations. Figure 6 shows a simplified class diagram with the basic elements for modeling data flows in the DSL. Here, the `Node` class represents

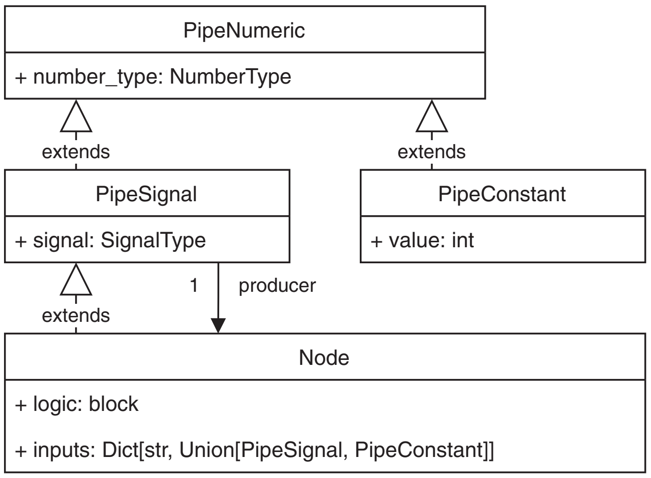

FIGURE 6 Simplified class diagram of the basic components for the internal data flow graph representation

an operation and thus a node in the graph. The associated hardware description of the logic is stored in the *logic* field. Instances of the class `PipeNumeric` and therefore also `PipeSignal` and `PipeConstant` can be used as input for an operation. The class `PipeSignal` associates a MyHDL signal, which is the hardware description of a connection, with a `Node` (field: *producer*) which drives this signal. A node can drive multiple signals as output and thus be listed for multiple `PipeSignals` in the *producer* field. The operations on which a `Node` depends is thus the set of all nodes listed in its inputs of type `PipeSignal` in the *producer* field.

Listing 6 shows the description of a simple data flow using the DSL. The occurring variables are of type `PipeNumeric`. The framework provides basic mathematical functions. By overloading the standard mathematical operators for all objects of type `PipeNumeric`, they can be used implicitly (lines 3 and 4) in addition to an explicit call (lines 2 and 5). To automatically convert the constants into the correct representation form (e.g., fixed-point numbers of different precision) and to avoid errors, the use of the class `PipeConstant` is enforced. It is crucial to emphasize that the represented code does not perform the calculations, but only describes them. In addition to the basic mathematical functions, the framework provides special nodes as an interface between classical logic and the computations described by the DSL. Listing 7 shows the use of these interface components (`PipeInput` and `PipeOutput`) to incorporate the calculations from Listing 6.

```
1 def calc(a, b):
2     add1 = add(a, PipeConstant.from_float(3))
3     add2 = add1 + b
4     mul1 = add1 * PipeConstant.from_float(5)
5     mul2 = mul(add2, a)
6     return (mul1, mul2)
```

Listing 6: Description of a example calculation as a data flow

```
data_in = PipeInput(in_valid, a=a_in_signal, b=b_in_signal)
res = calc(data_in.a, data_in.b)
data_out = PipeOutput(out_busy, a=res[0], b=res[1])
```

Listing 7: Interface between DSL and other logic

The complete data flow from Listing 6 in combination with Listing 7 can be visualized as a graph, as shown in Figure 7. This makes it easy to see the dependencies between the individual operations. Before the multiplication from line 5 in

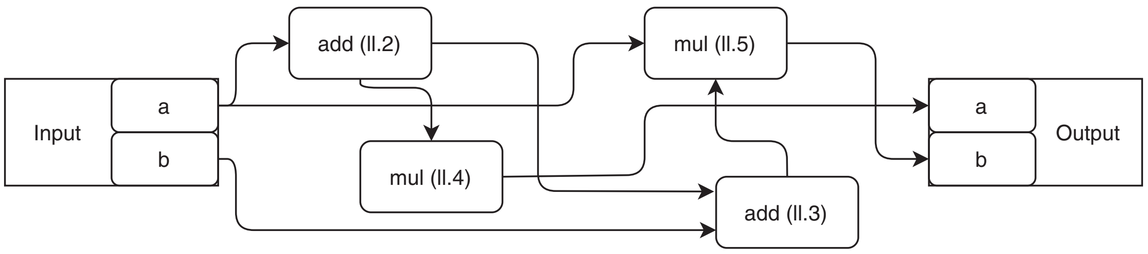

FIGURE 7 Representation of the data flow as a graph. Each inner node refers to a line of Listing 7

Listing 6 can be calculated, the addition from line 3 and thus also the addition from line 2 must be completed. Lines 3 and 4 from Listing 6, however, can be calculated simultaneously.

5.4.2 | Automated conversion of calculations into hardware

In order to be able to translate calculations of variable complexity into hardware automatically and without violating the time limits, the execution time of the logic is coded in each node. More complex computations can and should be implemented as a separate subflow using the basic functions provided by the framework. By doing so, all subsequent optimizations can be performed on the full resulting graph consisting of individual nodes implementing the needed mathematical base functions. In the following, the basic concept is presented on the assumption that all nodes need only one clock cycle to implement their functions. A naive approach to the conversion of the data flow (of Listings 6 and 7) into hardware is to impose the condition of waiting until all previous operations have completed before executing individual operations. To implement this, one could introduce a Boolean for each node that signals the end of its computation. A node monitors these signals for all its inputs and can thus wait to start its own calculation until all previous ones have completed. The number of clock cycles needed to compute the results for a set of inputs is determined by the longest path. In the given example calculation this is three clock cycles. It should be noted that in this type of implementation, the logic circuits of the individual operations are used only in one clock cycle and lie idle the rest of the time. The time spent waiting for the previous calculations and all clock cycles after the end of the calculation are unused.

The approach chosen in this work converts the data flow in a pipeline. The logic generated in this way can compute several data sets in parallel. Apart from the time needed to fill and empty all stages at the beginning and end, all logic is active in every clock cycle. The individual nodes are sorted according to their dependencies and assigned to the earliest possible stage, so that all dependencies have already been calculated in previous stages. For the example data flow, this is shown in Figure 8. A data set passes one stage per clock cycle. Signals that skip one or more stages must therefore be buffered using registers. These registers are automatically inserted into the data flow graph. Figure 9 shows the transformed graph.

When implementing a pipeline, the way of handling a halt of the pipeline is essential. Such a halt can occur if the subsequent logic cannot process any further data. The idle state of a pipeline caused by, for example, no new data being available at the input, must also be taken into account. The approach taken in this work is called *buffered handshake*.[32]

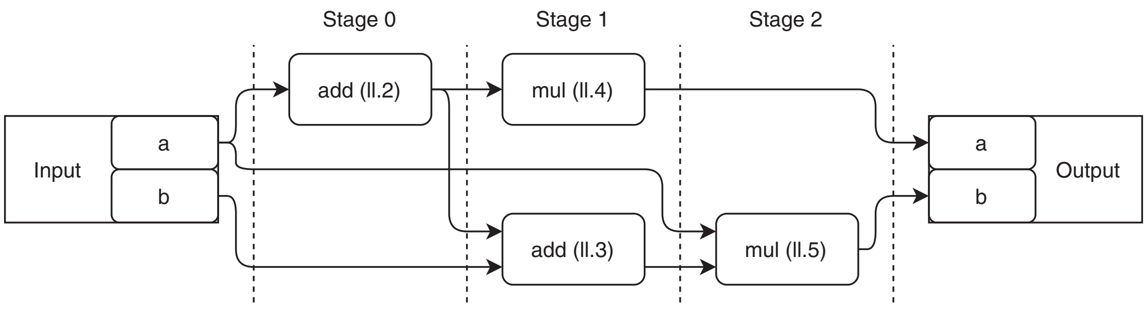

FIGURE 8 Presorted pipeline

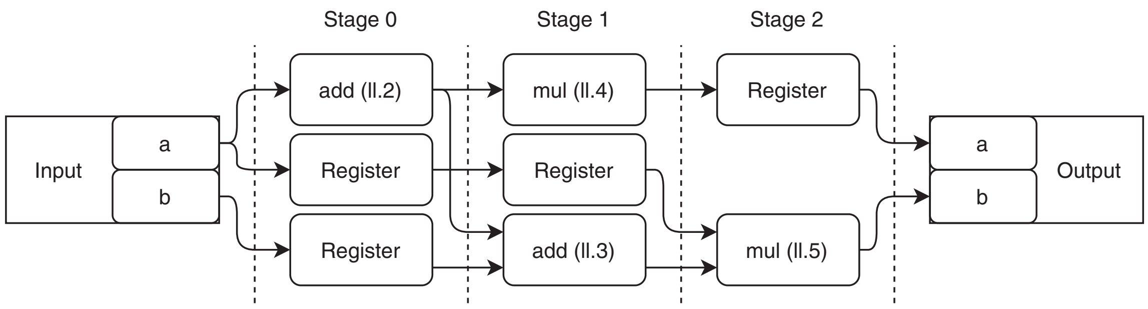

FIGURE 9 Pipeline with registers

The organizational logic for the pipeline operation is implemented in the individual stages. Whether the current input is valid is communicated by the previous stage via a Boolean signal. Another Boolean signal indicates whether the next stage is busy and therefore cannot receive data. A data transfer between two stages only takes place if the next stage is not busy and the current stage declares its data valid. Since only one preceding stage can be notified of the `busy` signal when the pipeline stops in each clock cycle, all nodes of a stage must be able to temporarily store a set of input values, the current output of the preceding stage. After evaluating the signals, the logic of a stage instructs its nodes whether to process the new data and pass it on directly, cache the result, or clear the memory. Figure 10 shows the control signals between the stages of the pipeline and the input and output nodes.

When building the graphs from the data flow, nodes with results that are not used any further are dropped. Table 1 compares the required clock cycles per solution step of the two described implementation variants. The pipelined implementation does not improve performance when solving a single IVP compared with the naive implementation. However, if we consider the performance when solving multiple IVPs, the time required decreases significantly. More complex solution methods perform more computations per step at the cost of additional hardware. As a result they can solve more IVPs simultaneously. Regardless of the solution method used, the pipeline requires only one additional clock cycle for each additional IVP at full capacity. This results in the following relationship: The more IVPs are solved, the closer the required clocks per step and per number of IVPs approaches 1. Comparing the FPGA resource requirements of the pipelined implementation to the naive implementation, the resource requirements are larger due to the additional logic

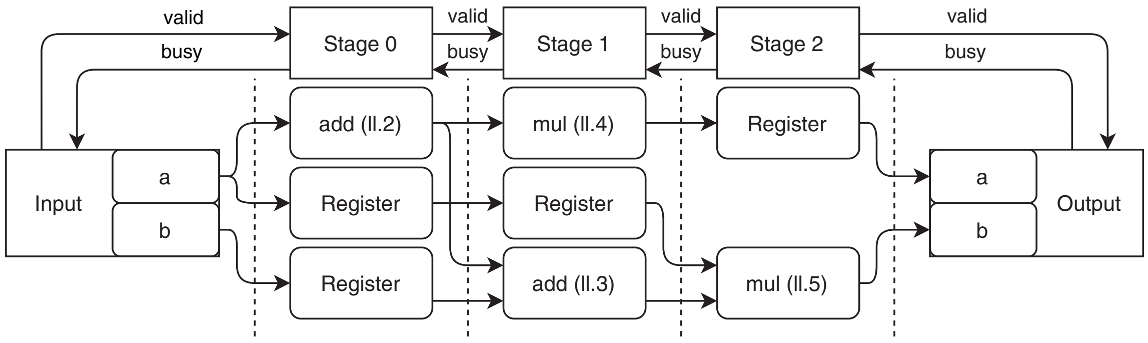

FIGURE 10 Final pipeline

TABLE 1 Comparison of clock cycles per step for the Lotka–Volterra equations[25] with internal fixed-point number representation

	Euler		RK4	
Number of IVPs	1	100	1	100
Naive implementation	6.2	607.4	24.2	2407.7
Pipelined implementation	6.2	108.2	24.2	109.8

circuits and registers required. However, the number of DSP blocks required does not differ. In practice, it has been shown that for computation-intensive circuits, the number of DSP blocks are the limiting factor. An optimal utilization of the DSP blocks, as realized by the presented pipeline implementation, is generally the target.

Due to the constraint that each stage is only one clock cycle long not all necessary logic can be represented even for elementary operations. This problem occurs, for example, when using fixed DSP blocks. A solution would be to adjust the duration in clock cycles of the stages to the longest elementary operation. But this would considerably limit the possibilities for parallelization of the operations. In this work a new type of node is introduced instead: the `MultiCycleNode`, which is used for logic that requires multiple clock cycles. The previous node is renamed to `OneCycleNode`. Figure 11 shows such a `MultipleCycleNode` lasting three clock cycles in a pipeline. In order not to lose data in case a halt of the pipeline occurs, the node must be able to store data for each of its stages. This is implemented internally with the help of a FIFO. With this approach faster operations can still be parallelized on the finest possible level (a clock cycle).

5.4.3 | Optimization of the basic mathematical operations

Since all calculations are always based on the basic mathematical operations provided by the framework, it is important to optimize them. Thereby the goals are the minimization of the needed DSP blocks as well as performance improvements by saving clock cycles or entire calculations. In order to efficiently pipeline combinatorial logic that does not require its own clock cycle, a new variant of a `Node` is introduced, the `ZeroCycleNode`. Depending on the type and the values of the parameters of the basic operations, different hardware circuits and thus implementations of them are used. In the following, such optimizations are demonstrated using the multiplication of signals with an internal fixed-point number representation as an example. The two parameters of the multiplication are called a and b. As specified in the implementation of the DSL, they can be either a `PipeConstant` or a `PipeSignal`. The behavior of the multiplication function differs for different pairs of types and values of the parameters. The respective behavior is described below by means of conditions. The first applicable condition is decisive. To simplify the conditions, it is assumed that parameter a is always of type `PipeConstant` as soon as one of the two parameters is of type `PipeConstant`.

- *a of type* `PipeConstant` *and a.value == 0:* Regardless of b, a `PipeConstant` with the value 0 is always returned.
- *a of type* `PipeConstant` *and b of type* `PipeConstant`*:* The result is a `PipeConstant` with value $a \cdot b$. Thus the calculation is performed at generation time.
- *a of type* `PipeConstant` *and a.value & (a.value - 1) == 0*: In *a.value* exactly one bit is set. Thus there is a multiplication or division by a multiple of 2. An arithmetic shift by n bits to the left is equivalent to a multiplication by 2^n. The variable n can also be negative, in which case an arithmetic shift to the right is performed, thus dividing. The implementation of an arithmetic shift by a constant value in hardware is possible without requiring resources by connecting the individual signals differently. Compared with an implementation with a DSP block, both resources (the DSP block) and time are saved. The function returns a `PipeSignal` which is driven by a `ZeroCycleNode`.
- all other cases: The multiplication is implemented using a DSP block. The result of the call is a `PipeSignal` which is driven by a `OneCycleNode`.

Table 2 lists information about the components of the generated pipelines for different solution methods, once with optimizations and once without optimizations. Additionally, the percentage of savings achieved by the optimizations of

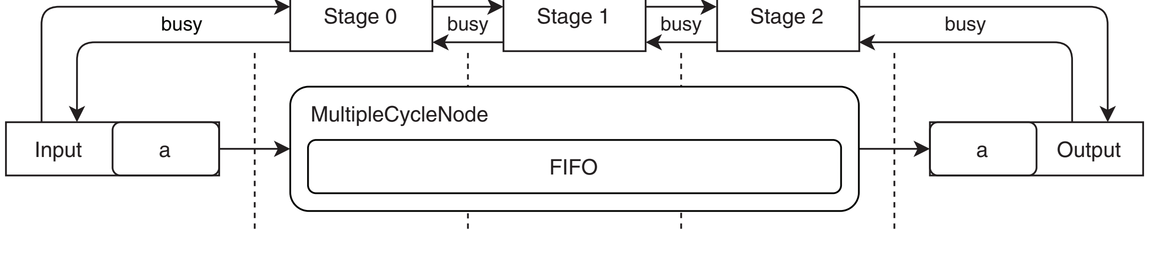

FIGURE 11 `MultipleCycleNode` in pipeline

TABLE 2 Comparison of the generated pipelines with and without optimization for the Lotka–Volterra equations[25] with internal fixed-point number representation

	Nbr_stages	Reg	Add	Sub	Mul_dsp	Mul_by_shift
Euler	8	68	7	1	13	/
Euler optimized	5	39	5	1	8	/
Saving	*37.5%*	*42.6%*	*28.6%*	*0.0%*	*38.5%*	
Heun	15	165	12	2	25	/
Heun optimized	11	116	10	2	16	4
Saving	*26.7%*	*29.7%*	*16.7%*	*0.0%*	*36.0%*	
RK4	31	519	28	4	56	/
RK4 optimized	23	325	20	4	40	5
Saving	*25.8%*	*37.4%*	*28.6%*	*0.0%*	*28.6%*	

the basic operations is given. Since each stage of a pipeline computes for exactly one clock cycle, the number of stages (*nbr_stages*) corresponds to the clock cycles required to run the pipeline. A saving of stages, such as by 37.5% for the Euler method leads to a performance improvement of the same percentage when solving a single IVP. In addition, significant savings can also be observed in the required DSP blocks for all methods. The knowledge that multiplication or division by a multiple of 2 can be implemented very efficiently for fixed-point number types in hardware can be taken into account when choosing a suitable solution method. The more coefficients of the Butcher tableau satisfy the condition, the more resource-efficient and performant a solution method can be implemented on a FPGA.

5.5 | Implementation of application specific solver unit internals

5.5.1 | Parsing of the ODE description

Most values of the configuration provided by the user can be used directly to parameterize the logic. This is not true for the specification of the ODE system. The RHSs of the equations are specified in the configuration file by the user in explicit form as a string. The task of the RHS parser is the automatic translation of the given expressions into the DSL which can then be used to automatically generate efficient and highly pipelined hardware circuits for the calculations, as described in Section 5.4. Therefore a grammar for infix expressions was defined using the library *PyParsing*.[33] After parsing the provided strings, an algorithm walks through the derived parse tree and reassembles the expressions using the DSL.

5.5.2 | Implementation of the RK methods

The developed data flow DSL allows a very direct and readable implementation of the calculations necessary for the RK methods. The general approach is illustrated below using the calculations for the stage vectors as an example. As already explained in Section 2.1 about the basics of RK methods, Equation (7) is used to calculate the stage vectors $v_1, \dots, v_s \in \mathbb{R}^n$ of an s-stage RK method:

$$v_j = f\left(x_k + h \cdot c_j, \eta_k + h \cdot \sum_{l=1}^{s} a_{jl} v_l\right) \quad \text{for } j = 1, \dots, s. \tag{7}$$

The `stage()` function shown in Listing 8 describes the calculation of a stage vector. The parameter `config` is used to pass a stage configuration defined at the time of generation. In addition to the index j of the stage, this also contains the coefficients a_{jl} for $l = 1, \dots, s$ and c_j as well as the right-hand sides (`config.components`) and the size of the system of equations n. In addition, the function receives one signal each via the parameters `h` and `x` and a list of signals of size n via `y`. The signals of the previous stage vectors are provided in a nested list via the parameter `v`. Line 2 defines the calculation of the *x* value for the function evaluation. Care must be taken to convert the constant to the correct data type. From line 3 on, the calculations of the *y* vector are described. The function `dot_product()`, which is implemented using the basic mathematical operations provided by the framework, calculates the scalar product of the two vectors passed to it. In line 8 the actual function evaluation takes place. Here, each right-hand side of the explicit equation system is parsed as described in Section 5.5.1. The implementation of the remaining calculations of the RK methods is analogous.

```
1  def stage(config, h, x, y, v):
2      rhs_x = x + h * PipeConstant.from_float(config.c)
3      rhs_y = [y[i] + h * dot_product(
4          [PipeConstant.from_float(el) for el in config.a],
5          [el[i] for el in v[:config.stage_index]]
6      ) for i in range(config.system_size)]
7
8      return [expr_parser.expr(rhs_expr, {
9          'x': rhs_x,
10         'y': rhs_y
11     }) for rhs_expr in config.components]
```

Listing 8: Implementation of the RK stage

5.6 | Interface

5.6.1 | .slv file format

The logic generator generates a description of the logic circuits on the RTL in MyHDL which is then automatically translated into Verilog. After synthesis and implementation, Quartus Prime[31] generates a bitstream which encodes the configuration of the FPGA. The tools of the OPAE store this, together with additional information required for the configuration process, in a .gbs file. For the later use of the generated AFUs by the runtime software provided in this work, further information is required. Both the chosen number representation and the size of the ODE system have an influence on the interface. To make the use and pregeneration of solvers as easy as possible, a new simple file format was developed as part of this work that combines the information needed for the runtime with the .gbs file. The structure is outlined in Figure 12. After a static ID identifying the file format, the length of the configuration header is stored in four bytes. This is followed by the header encoded in JSON with all the additional information needed later. Following this, the original .gbs file is embedded. A library for easy packing and unpacking of the .slv files has been implemented.

5.6.2 | Interface between CPU and AFU

The OPAE provides the core cache interface (CCI) for communication between the CPU and a AFU on the FPGA. Basically there are two possibilities for data exchange. The control/status registers (CSRs) are an address space for which the CPU

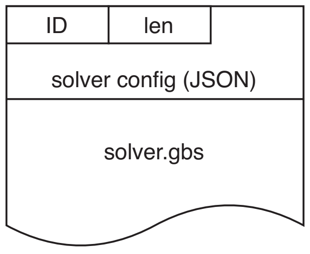

FIGURE 12 Structure of the .slv file format

can make write and read requests. The AFU is only a passive participant. The exact implementation on the side of the AFU is left to the developer. The second available variant of data exchange is reading and writing from and to the RAM of the CPU. Here the AFU can initiate the transfer of up to four cache lines to or from a RAM address simultaneously. A cache line is 512 bits = 64 bytes in current processors. Since the AFU cannot access the CPU's virtual memory management, it requires the physical address for all requests. To keep an allocated memory area at the same physical address, the corresponding RAM page must be pinned by the runtime software. The size of a memory area accessible to the FPGA is thus limited by the page size (default: 4096 bytes). If you want to make larger memory areas accessible, the use of the so-called hugepages is necessary.[34]

The solvers generated in this work must be sent the IVPs to be computed. In addition, the AFU must transmit the results back to the CPU at the end of the calculation. Two memory areas in the RAM of the CPU are used to realize this data exchange. One is written by the runtime with the initial values and thus serves as input for the AFU. The AFU always retrieves four cache lines, called a chunk in the following, together. The second memory area is used for the output of the results. For the physical addresses of the two areas, the AFU provides two control registers at fixed addresses. In addition, the number of chunks to be read as well as a Boolean to start the solution process and the current status of the solver are specified in further registers.

```
class InputData(StructDescription, metaclass=StructDescriptionMetaclass):
    id = BitVector(num.INTEGER_SIZE)
    x_start = BitVector(num.TOTAL_SIZE)
    y_start = List(system_size, BitVector(num.TOTAL_SIZE))
    h = BitVector(num.TOTAL_SIZE)
    n = BitVector(num.INTEGER_SIZE)
    _bit_padding = BitVector(len_padding)
```

Listing 9: Description of the input data format (shortened)

The data format of the input data is defined in Listing 9. With the help of the `StructDescription` library developed as part of rtlode, bit-precisely packed data types can be described in MyHDL. For the IVPs, an ID is defined in addition to the specification of the initial values for x and y as well as the step size h and the number of steps n. This ID can later be used to match the result data set to the corresponding IVP. A constant order between input and output data is not guaranteed. With the help of an optional padding the total length of an input data set is extended to the next byte boundary. This simplifies the handling on the side of the CPU.

Figure 13 visualizes the loading of IVPs. The data sets are stored in the input memory area by the runtime in such a way that no chunk boundaries are crossed. This simplifies the logic on the FPGAs side, since the positions of each data set thus remain constant relative to the chunk. Accordingly, at the end of a chunk, parts of the memory area remain unused. This can be seen in Figure 13 in the third and seventh cache lines. In the generated solvers on the FPGA, the host RAM

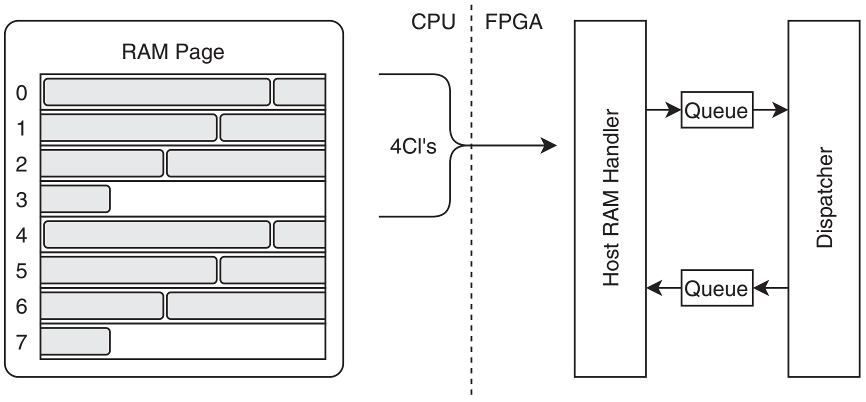

FIGURE 13 Loading of input data sets (the IVPs)

handler is responsible for managing CPU memory accesses. This part of the AFU logic, once the solver process is started, loads one chunk of data at a time from the input memory area. As soon as all four cache lines have arrived, the individual data sets are separated and stored in a queue. This serves the purpose to decouple the main solver logic from the memory accesses. The dispatcher module then reads the data sets from the queue and distributes them to the individual solver units. Saving of the results works the same way. As soon as all input data sets have been processed and the results have been transferred to the CPU, the AFU signals this with a change of its status value in the CSRs.

6 | RESULTS

6.1 | Time needed for generation

The process proposed in this article to generate circuit descriptions from configuration files naturally requires additional time on top of the time required for synthesis and implementation. The generation as well as synthesis and implementation were performed on a host with an Intel(R) Xeon(R) CPU E5-2630 v2 @ 2.60 GHz. In Table 3 the time in seconds needed to generate the Verilog descriptions of the complete AFUs is shown depending on the chosen solution method and number of parallel solvers for different number representations. It should be noted that the generation process currently uses only one CPU core and has not been optimized.

From the results of Table 3 it can be seen that a doubling of the number of solver units also almost doubles the time required. In addition, the use of more complex solution methods also leads to a significant increase in the required time. Regarding the number representations, it can be observed that the fixed point number representation can be generated the fastest. Changing the precision of the floating point number representation from single to double also increases the required time noticeably. In summary, configurations that lead to more or more complex logic also take longer to generate.

Quartus Prime takes $1.60 \cdot 10^4$ s on the same host machine to generate a finished bitstream from the Verilog description of a AFU with 32 single precision floating point Euler solvers. The time required to generate the description $2.06 \cdot 10^2$ s is several orders of magnitude smaller and negligible in practice.

6.2 | Comparison to Stamoulias et al.[5]

In the work of Stamoulias et al.,[5] the choice of using HLS tools was justified by a reduced development time. With the newly proposed concept in this article, the specific development effort is reduced to describing the solver and the problem in a configuration file. Based on this, rtlode can automatically generate a logic circuit. A direct comparison with the work from Stamoulias et al.[5] is unfortunately not possible due to the inaccessible source code and a different hardware at hand. However, the information about their solution and the hardware used is sufficiently accurate to allow a comparison with some limitations. Unless otherwise stated, the data in the following refer to the use case of Stamoulias et al.,[5] which is

TABLE 3 Needed time in (s) to generate a Verilog description of the complete AFU

		Number of parallel solver units					
		1	**2**	**4**	**8**	**16**	**32**
Euler	SP	6.93	$1.34 \cdot 10^1$	$2.75 \cdot 10^1$	$5.21 \cdot 10^1$	$1.04 \cdot 10^2$	$2.06 \cdot 10^2$
	DP	9.81	$1.85 \cdot 10^1$	$3.71 \cdot 10^1$	$7.32 \cdot 10^1$	$1.46 \cdot 10^2$	$2.92 \cdot 10^2$
	54b fix	2.23	3.83	7.13	$1.36 \cdot 10^1$	$2.71 \cdot 10^1$	$5.38 \cdot 10^1$
Heun	SP	$1.46 \cdot 10^1$	$2.86 \cdot 10^1$	$5.60 \cdot 10^1$	$1.18 \cdot 10^2$	$2.28 \cdot 10^2$	$4.57 \cdot 10^2$
	DP	$2.15 \cdot 10^1$	$4.19 \cdot 10^1$	$8.52 \cdot 10^1$	$1.69 \cdot 10^2$	$3.32 \cdot 10^2$	$6.69 \cdot 10^2$
	54b fix	3.72	6.61	$1.26 \cdot 10^1$	$2.72 \cdot 10^1$	$4.86 \cdot 10^1$	$1.01 \cdot 10^2$
SSPRK3	SP	$2.59 \cdot 10^1$	$5.04 \cdot 10^1$	$9.98 \cdot 10^1$	$1.98 \cdot 10^2$	$4.19 \cdot 10^2$	$8.16 \cdot 10^2$
	DP	$3.78 \cdot 10^1$	$9.44 \cdot 10^1$	$1.50 \cdot 10^2$	$2.97 \cdot 10^2$	$6.13 \cdot 10^2$	$1.23 \cdot 10^3$
	54b fix	6.17	$1.12 \cdot 10^1$	$2.18 \cdot 10^1$	$4.24 \cdot 10^1$	$8.51 \cdot 10^1$	$1.68 \cdot 10^2$

the solution of multiple instances of the Lotka–Volterra equations.[25] The clock frequency for the generated logic circuits was fixed at 200 MHz, the same value used by Stamoulias et al.[5] for their performance analysis. Table 4 compares the main resources of the two FPGAs involved. The *Xilinx Kintex UltraScale KU060* used by Stamoulias et al.[5] has fewer of the important DSP blocks in addition to a smaller number of logical blocks and a smaller internal RAM compared with the *Intel Arria 10 GX 1150* used in this work. In order to still allow a comparison, hardware differences have to be compensated for. In our use case the internal RAM is less relevant. The number of DSP blocks and logical units is much more important. To avoid any advantage from the estimation, a very conservative approach has been chosen by scaling the reference values of Stamoulias et al.[5] to 63.09%. This value corresponds to the ratio of the logical blocks between the FPGAs which is the lower one of both relevant ratios. In reality, this ideal speedup of the reference values caused by the correction would not be achievable.

In their work, Stamoulias and his colleagues[5] use single and double-precision floating-point number representations. Our tool rtlode also supports fixed-point number representations. As an example, a 54 bit fixed-point number representation, with 16 pre- and 34 postdecimal places, is also considered in the following. In order to allow a fair comparison with floating-point number representations, for the given IVP each solver method was verified to achieve at least the same accuracy as with the double-precision floating-point representation. This was done, in accordance with Stamoulias et al.,[5] by calculating the mean sum of errors compared with an Euler double-precision floating-point reference solution with $1 \cdot 10^7$ steps. Table 5 shows the required DSP blocks per solver unit for different solution methods and different internal number representations: single-precision floating-point (SP), double-precision floating-point (DP), and the mentioned 54 bit fixed-point (54b fix).

With increasing complexity of the solution method the resource requirements per solver unit increase. However, it should be noted that each solver unit uses a pipeline internally and thus more internal computations also lead to more IVPs that can be solved parallel in one solver unit. Table 6 compares the time required to solve 10k IVPs for different generated circuits with the scaled results of the HLS approach from Stamoulias et al.[5] The step size and number of steps for each method are based on the data of their work and are tuned by them to achieve comparable accuracy in the results.

For the Euler method, it can be seen that an increase in the number of parallel solver units steadily reduces the required computation time. With 32 parallel solver units the double-precision floating-point representation is ≈6.5 times faster than the scaled reference value.[5] Even the single-precision solver is ≈4.8 times faster. The other solver methods also outperform the reference consistently. However, it can be seen that an increase in the number of parallel solver units for

TABLE 4 Comparison of FPGAs used in this work and by the reference work[5]

	DSP blocks	Logical blocks	Internal RAM
Xilinx Kintex UltraScale KU060[35]	2760	725,550	9.1 MB
Intel Arria 10 GX 1150[27]	3036	1,150,000	65 MB
Ratio	*90.91%*	*63.09%*	*14.00%*

TABLE 5 Comparison of DSP blocks needed per solver method and unit for the Lotka–Volterra equations.[25]

		Mul	Add/Sub	DSP blocks
Euler	SP	10	5	30
	DP	10	5	80
	54b fix	8	5	64
Heun	SP	22	11	66
	DP	22	11	176
	54b fix	16	11	128
SSPRK3	SP	36	19	110
	DP	36	19	288
	54b fix	30	19	240

TABLE 6 Needed time in (ms) to solve 10k IVPs with different numbers of steps

		Number of parallel solver units						Reference 5	
		1	2	4	8	16	32	Scaled 63.09%	Speedup
Euler (10k steps)	SP	501.21	250.62	126.03	64.43	33.63	16.92	80.76	4.77
	DP	501.61	250.83	127.63	66.03	/	/	426.49	6.46
	54b fix	501.12	250.52	125.76	64.05	33.28	/		
Heun (60 steps)	SP	3.04	1.54	0.88	0.86	1.01	/	2.39	2.78
	DP	3.05	1.69	/	/	/	/	4.18	2.47
	54b fix	3.04	1.52	0.90	0.87	/	/		
SSPRK3 (12 steps)	SP	0.86	0.85	0.86	0.86	/	/	1.40	1.65
	DP	1.37	1.36	/	/	/	/	59.80	43.97
	54b fix	0.88	0.85	0.84	/	/	/		

both the Heun and SSPRK3 methods does not result in a lower computation time. A value between ≈0.9 and ≈1.3 ms depending on the number representation is not exceeded. The small time interval of the IVP in combination with a large step size compared with the Euler method, due to the better numerical properties of the multilevel methods, leads to a very small number of steps (Heun: 60, SSPRK3: 12). Thus, the limiting factor is no longer the computation on the FPGA, but the interface between CPU and FPGA, which is not able to transfer the data of the 10k IVPs sufficiently fast. But despite this limitation, the SSPRK3 solver with double-precision floating-point numbers is more than 43 times faster than the already scaled HLS variant of Stamoulias et al.[5]

6.3 | Comparison of solution methods

To be able to look at the performance of the computations in the generated logic circuits, again 10k IVPs were solved with the different methods. However, in contrast to the IVPs of Stamoulias and his colleagues,[5] the size of the time interval to be calculated was increased from 10 to 100. A reference solution was calculated using the Euler double-precision floating-point solver with $1 \cdot 10^9$ steps. The number of steps required by the different number representations and solution methods to obtain an averaged sum error of less than $2.5 \cdot 10^{-5}$ is shown in Table 7. Using the Euler method, the single-precision floating-point number representation and the 54 bit fixed-point number representation cannot achieve the desired accuracy. For all other combinations, the performance results are shown in Table 8.

We again observe that the number format does not influence the execution time of a solver noticeably, but with a smaller number format more solver units can be used. As expected, increasing the number of parallel solver units now also leads to an improvement in performance for the Heun and SSPRK3 methods, which speeds up the execution time nearly linearly. For example, the speedup of SP Heun using 16 solver units is 14.6. Moreover, we can confirm that it always takes 5 ns to compute one time step of one IVP on one solver unit, which corresponds to the cycle time at 200 MHz, that is, the solver units have an optimal throughput of one IVP per clock cycle, independent of the number of stages. As a result, using higher order solution methods has now become a significant advantage. While the fastest configuration of the Euler solver (eight solver units) needs more than ≈2.6 s, even the slowest configuration (only one solver unit) of Heun and SSPRK3 can calculate the same result in less than 76 and 17 ms, respectively. Even though SSPRK3 can use

TABLE 7 Number of steps required to achieve an averaged accuracy sum smaller than $2.5 \cdot 10^{-5}$ at $t = 100$

	Euler	Heun	SSPRK3
SP	/	1500	330
DP	$4 \cdot 10^6$	1500	330
54b fix	/	1500	330

TABLE 8 Needed time in (ms) to solve 10k IVPs using a simulation time interval of 100 s

		Number of parallel solver units				
		1	**2**	**4**	**8**	**16**
Euler ($4 \cdot 10^5$ steps)	DP	20063.22	10031.63	5103.83	2639.92	/
Heun (1500 steps)	SP	75.21	37.83	19.35	9.72	5.15
	DP	75.25	37.99	/	/	/
	54b fix	75.25	37.81	19.37	9.80	/
SSPRK3 (330 steps)	SP	16.59	8.32	4.26	2.27	/
	DP	16.68	8.36	/	/	/
	54b fix	16.60	8.32	4.27	/	/

less solver units than Heun, because the hardware resources are needed to implement the additional stages, it requires only about one third of the number of time steps and is thus more than two times faster than Heun when comparing the fastest configurations.

7 | DISCUSSION

7.1 | Achieved performance and potential improvements

To give an example of the floating point performance achieved, for Heun's method using the single-precision floating-point representation and 16 solver units, the solution of 10k IVPs using 1500 steps at 200 MHz takes 5.15 ms. Since the solution of an IVP requires 33 FLOP per step, this yields a floating point performance of about 96 GFLOP/s. However, the vendor specifies a peak performance of 1.366 TFLOP/s when using fused multiply-add (FMA) operations. Hence, for a design using a mixture of FMA and other floating point operations one could estimate about 800–1000 GFLOP/s being achievable.

The pipeline of a solver unit already achieves an optimal throughput of one IVP per clock cycle. Thus, improvements should aim at optimizing the resource usage (DSP and logic blocks) so that more solver units can be placed on the FPGA device. The number of DSPs used in this example is only 1056, where 3036 DSPs are available on the device. This indicates a suboptimal fitting of the solver units. If this will be solved, we could gain a factor of about 2.9. The clock frequency currently is at 200 MHz. By optimizing the design, achieving a higher clock frequency of about 450 MHz could be a realistic goal, thus gaining another factor of 2.3. Joining both optimization approaches could yield about 640 GFLOP/s.

Since currently multiplications and additions are not yet fused to FMA operations, this is another possibility to improve the performance and to get closer to the peak.

7.2 | Comparison to conventional HPC architectures

The main research questions addressed by this article are how to efficiently utilize FPGAs as emerging architectures and how to bridge the gap between the complexity of efficient logic circuit development and easy usage by domain experts to enable user acceptance of FPGAs as HPC accelerators. The latter point may raise the question how FPGAs currently compare to conventional HPC architectures, such as CPUs and GPUs.

While a detailed and careful comparison is outside the scope of this article, we would however like to give some rough impression on the current competitiveness of FPGAs for our specific use case. It should be noted that the comparison presented here cannot be used to make a fundamental statement about the suitability or future potential of the different architectures. Unlike FPGA accelerators, CPUs and GPUs have been used as standard compute devices in the HPC domain for several decades now. Their fundamental development and research is additionally co-funded by significant end-user unit sales, which are not (yet?) occurring for FPGAs. The intent of this comparison is only to give a feeling about the potential of FPGAs as accelerators in the HPC domain.

For the cross-architecture comparison, Heun's method with double floating point precision is used. On all architectures, IVPs with 10^6 steps and a step size of 0.001 are calculated. The number of IVPs is chosen individually for each platform to reach a proper device usage. The generated AFU on the FPGA combines three parallel solvers at a clock rate of 200 MHz. When computing 10k IVPs, the FPGA takes about 1.7 ms per IVP with an average power consumption of about 27 W resulting in an energy-to-solution per IVP of about 45 mJ. The information about the power consumption was taken from the onboard power sensor of the accelerator card.

Based on a generic open source library that provides different implementations of Runge–Kutta solvers,[36] two differently optimized CPU-based platforms were benchmarked: one compute node with two Intel Xeon Gold 6248 CPUs (Cascade Lake microarchitecture, 20 cores each) and a notebook with an Intel Core i7-4600U CPU (Haswell microarchitecture, two cores). To measure the energy consumption, likwid[37] was used which reads the running average power limit (RAPL) counters of the CPU. On the compute node the implementation *mpi_PipeD* was performing best. This implementation which exploits the temporal locality of read accesses through its blocked and skewed loop structure is parallelized using MPI. For the measurements, the clock rate was fixed at 2.5 GHz. To solve 100k IVPs, the compute node needs about 770 μs per IVP with an average combined power consumption of about 260 W. Although being slightly faster per IVP compared with the FPGA, the energy-to-solution per IVP of about 200 mJ is more than four times higher. On the notebook, a sequential implementation (*seq_PipeDls*) with similar loop structure achieves the best results. With this implementation, the notebook requires about 8.5 ms per IVP when solving 10k IVPs with an average power consumption of about 13 W. This results in an energy-to-solution per IVP of about 110 mJ. The notebook CPU is thus significantly slower than the compute node and the FPGA, but in terms of energy efficiency it ranks between the two.

To compare with a GPU, a hand-written CUDA implementation optimized for the specific right-hand side was used to measure the performance of a Nvidia Titan Volta GPU. In order to properly utilize the GPU, 163 840 IVPs were solved. The NVIDIA Management Library (NVML)[38] was used to sample the power usage. With an average power consumption of 110 W (running at 1.3 GHz) the GPU requires only 5.8 μs per IVP. This results in an energy-to-solution per IVP of only 630 μJ, which significantly outperforms both the CPU and the FPGA implementation.

Hence, for this small example case and for the implementations used, the FPGA can be considered competitive to CPUs since it beats both the server and even the mobile CPU in energy-to-solution while the time-to-solution ranks between the two CPUs. Assuming that performance improvements of our logic design were possible by a factor of about 7 (cf. Section 7.1), we could beat the 40 Cascade Lake cores also in time-to-solution. However, the open source library used on the CPUs may also have room for improvements.

Since our test case can easily be scaled to highly data-parallel problem sizes, it is also well-suited for GPUs. Thus, the Volta GPU needed significantly less time and energy per IVP than the FPGA and the two CPUs. However, this proportion may be different for test cases and applications which require a higher degree of pipeline parallelism and, thus, can benefit more from a custom-built pipeline. This may be the case, for example, if problem sizes are small. Moreover, it is not yet clear how these architectures will evolve in the future. In particular, as sales numbers increase and make it more profitable, FPGAs might use the same small manufacturing technologies as GPUs and, thus, be equipped with more DSP and logic blocks, faster memory, caches, higher clock frequencies, and so on, so that they can better play out the strengths of custom computing.

7.3 | Portability

In the scope of this article, we limited ourselves to the support of only one specific FPGA accelerator card (see Section 5.1). This is sufficient to present our approach as a proof-of-concept. In order to use rtlode on a different hardware, some adjustments have to be made. Mainly, the basic math operators need to be reimplemented. In most cases replacing the IP core they are based on should be sufficient. For FPGAs from vendors other than Intel, the FPGA–CPU interface including the runtime environment of the host must also be adapted. Instead of maintaining multiple variants of rtlode for different target FPGAs, rtlode should be adjusted to support multiple FPGAs which can then be selected as target platforms for the logic generation.

Worth pointing out in this context is that the effort needed to adapt to other FPGA platforms is not specific to HDL. In order to achieve performant HLS implementations, an FPGA expert must perform significant manual optimization work, which are specific for the respective FPGA.[23] Switching the FPGA vendor even requires considerably more work, since most currently used HLS tools are vendor specific.

7.4 | Current limitations

The underlying concept of mapping all necessary calculations to solve an ODE system directly to hardware imposes a limit on the size of the ODE systems due to the limited resources (e.g., number of DSPs) of the FPGAs used. In particular, even on larger FPGA devices available today it will not be possible to generate logic for an n dimensional ODE system consisting of n different equations if n is too large. Following the work of Stamoulias et al.,[5] our current benchmark example is a replication of a small ODE system, where the replicated instances of the ODE system can be solved with different initial values. However, in many large ODE systems, several equations share similar or even identical computational patterns, for example, in ODE systems resulting from a semi-discretization of a PDE system by the method of lines. A future extension of rtlode to support such systems is possible. Currently, the tool does not check whether the resources of the FPGA are sufficient for the user-requested number of solver units.

In our approach, a specific logic is generated for each configuration of RK tableau, ODE system, and numerical precision, which is then synthesized, placed and routed. Therefore, it takes a relatively long time to get from a configuration to running a customized solver on the FPGA. The same applies to all approaches that work with custom logic which is specific to the user-defined ODE system, for example, the HLS–based approach proposed by Stamoulias et al.[5] and the manually written HDL variant of Fasih et al.[14] An approach to which this does not apply is the ODE processor of Huang et al.[15] Although it is optimized for solving ODE systems in general, it does not provide any problem-specific logic and thus only needs to be implemented once.

In addition to these conceptual limitations, there are some functional limitations in the current rtlode tool. Since the current version of the tool only serves as proof-of-concept, only the operations required for the test problem considered have been implemented by now, that is, $+$, $-$, and $*$. However, additional mathematical operations can be added as needed. Furthermore, only one FPGA platform (see Section 5.1) is supported at the moment. The possibilities for porting rtlode to other FPGA platforms are discussed in Section 7.3.

8 | CONCLUSION

In this work, a novel approach for solving differential equations on FPGAs was presented. By using the rtlode tool developed here, circuits solving systems of ODEs can be generated automatically. Through a simple user interface, circuits for other differential equation systems or solution methods can be generated without FPGA- and programming-specific knowledge.

This approach does not only save the previously necessary specific development effort per solution approach (ODE system and solution method), but also significantly expands the possible user group. The logic of the solvers is described as a data flow. This approach, which is closer to hardware than HLS, allows for a high-performance implementation and subsequent optimization. As the evaluation shows, these automatically generated circuits consistently outperform comparable circuits generated with algorithms hand-optimized for FPGAs using HLS tools[5] and are up to 43 times faster.

If the goal is to create a single high-performance implementation of a computation (e.g., in the development of an application-specific integrated circuit [ASIC]), manual optimization on the RTL may be worthwhile. To save development effort for single circuits of any form, HLS tools can be used. However, for repetitive tasks, the approach of this work to generate circuits automatically based on simple configuration files is recommended. The necessary development effort has to be done only once and additionally the generated circuits are more performant compared with the HLS approach.

If the value and intermediate value range of the IVP for the ODE system is known, the use of a fixed-point number representation may be worthwhile. Due to the more efficient implementation on the FPGA and more optimization possibilities a faster result can be expected.

9 | FUTURE WORK

Based on the logic generator rtlode developed here, further adaptations and extensions are possible. An automatic analysis of the IVP and a resulting automated adjustment of the precision of the fixed-point number types can increase the usability and adaption of the more efficient fixed-point number types. Support for other solution methods, such as implicit RK methods and parallel iterated Runge–Kutta (PIRK) methods, or the support for adaptive step-size controls, may also be a component of future research. Another major area of research is the efficient support of large ODE systems that can be solved, for example, with stencil-shaped access pattern.

In Section 5.4.3, the efficient implementation of multiplications and divisions with a multiple of 2 for fixed-point numbers was discussed. In this context, it would also be worthwhile to investigate different RK methods and their suitability for efficient use on FPGAs. The development of special mathematical methods is conceivable.

By clearly separating and further developing the underlying framework from the rtlode tool presented here, it could serve as a general basis for the development of application-specific efficient logic generators. A multitude of further optimizations of the translation logic is imaginable, some of them are already mentioned in Section 7.1.

ACKNOWLEDGMENTS

The authors thank Tim Werner for providing a GPU implementation and measurement results for comparison. This work has been supported by the German Research Foundation (DFG) under grant KO 2252/3-2.

CONFLICT OF INTEREST

The authors declare no potential conflict of interests.

FINANCIAL DISCLOSURE

None reported.

DATA AVAILABILITY STATEMENT

The rtlode framework is available as open source on GitHub.[10] Data sharing is not applicable to this article as no data sets were generated or analyzed during the current study.

ORCID

Silas Bartel https://orcid.org/0000-0003-0771-3508
Matthias Korch https://orcid.org/0000-0001-7267-0378

REFERENCES

1. Putnam A, Caulfield AM, Chung ES, et al. A reconfigurable fabric for accelerating large-scale datacenter services. *IEEE Micro*. 2015;35(3):10-22. doi:10.1109/MM.2015.42
2. Lant J, Navaridas J, Luján M, Goodacre J. Toward FPGA-based HPC: advancing interconnect technologies. *IEEE Micro*. 2020;40(1):25-34. doi:10.1109/MM.2019.2950655
3. Bacon D, Rabbah R, Shukla S. FPGA programming for the masses: the programmability of FPGAs must improve if they are to be part of mainstream computing. *Queue*. 2013;11(2):40-52. doi:10.1145/2436696.2443836
4. Nane R, Sima V, Pilato C, et al. A survey and evaluation of FPGA high-level synthesis tools. *IEEE Trans Comput Aid Des Integr Circuits Syst*. 2016;35(10):1591-1604. doi:10.1109/TCAD.2015.2513673
5. Stamoulias I, Möller M, Miedema R, Strydis C, Kachris C, Soudris D. High-performance hardware accelerators for solving ordinary differential equations. Proceedings of the 8th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2017); 2017.
6. Yang C, Sheng J, Patel R, Sanaullah A, Sachdeva V, Herbordt MC. OpenCL for HPC with FPGAs: case study in molecular electrostatics. Proceedings of the 2017 IEEE High Performance Extreme Computing Conference (HPEC); 2017:1-8
7. Sanaullah A, Herbordt MC. FPGA HPC using OpenCL: case study in 3D FFT. Proceedings of the 9th International Symposium on Highly-Efficient Accelerators and Reconfigurable Technologies (HEART 2018); 2018.
8. de Fine LJ, Blott M, Hoefler T. Designing Scalable FPGA architectures using high-level synthesis. Proceedings of PPoPP'18: 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming; 2018:403-404.
9. De Matteis T, de Fine LJ, Hoefler T. fBLAS: streaming linear algebra on FPGA. Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC'20); 2020
10. Bartel S rtlode; 2021. https://github.com/UBT-AI2/rtlode. doi: 10.5281/zenodo.5514321
11. Hairer E, Nørsett S, Wanner G. *Solving Ordinary Differential Equations I. Nonstiff Problems*. 2nd ed. Springer; 2000.
12. Intel Corporation Intel FPGA SDK for OpenCL software technology; 2021. https://www.intel.com/content/www/us/en/software/programmable/sdk-for-opencl/overview.html
13. Osana Y, Fukushima T, Yoshimi M, et al. A framework for ODE-based multimodel biochemical simulations on an FPGA. Proceedings of the 2005 International Conference on Field Programmable Logic and Applications (FPL); 2005:574-577.
14. Fasih A, Trong TD, Chedjou JC, Kyamakya K. New computational modeling for solving higher order ODE based on FPGA. Proceedings of the 2nd International Workshop on Nonlinear Dynamics and Synchronization; 2009:49-53

15. Huang C, Vahid F, Givargis T. A custom FPGA processor for physical model ordinary differential equation solving. *IEEE Embed Syst Lett*. 2011;3(4):113-116. doi:10.1109/LES.2011.2170152
16. Ruan Z, He T, Li B, Zhou P, Cong J. ST-Accel: a high-level programming platform for streaming applications on FPGA. Proceedings of the 26th IEEE International Symposium on Field-Programmable Custom Computing Machines (FCCM 2018); 2018:9-16.
17. Koeplinger D, Feldman M & Prabhakar R et al. Spatial: a language and compiler for application accelerators. Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2018); 2018:296-311.
18. Chisel/FIRRTL developers Chisel/FIRRTL hardware compiler framework; 2020. https://www.chisel-lang.org/
19. Sérot J, Berry F. High-level dataflow programming for reconfigurable computing. Proceedings of the 2014 International Symposium on Computer Architecture and High Performance Computing Workshop; 2014:72-77.
20. Sano K. DSL-based design space exploration for temporal and spatial parallelism of custom stream computing. Proceedings of the Second International Workshop on FPGAs for Software Programmers (FSP 2015); 2015:29-34.
21. Mondigo A, Ueno T, Sano K, Takizawa H. Scalability analysis of deeply pipelined tsunami simulation with multiple FPGAs. *IEICE Transactions on Information and Systems*. 2019;E102.D(5):1029-1036. doi:10.1587/transinf.2018RCP0007
22. Korch M, Werner T. An in-depth introduction of multi-workgroup tiling for improving the locality of explicit one-step methods for ODE systems with limited access distance on GPUs. *Concurr Comput Pract Exp*. 2021;33(11):e6016. doi:10.1002/cpe.6016
23. Gorlani P, Kenter T, Plessl C. OpenCL implementation of cannon's matrix multiplication algorithm on intel stratix 10 FPGAs. Proceedings of the 2019 International Conference on Field-Programmable Technology (ICFPT); 2019:99-107
24. YAML Project. YAML: YAML Ain't markup language; 2009. https://yaml.org/
25. Brauer F, Castillo-Chavez C. *Mathematical Models in Population Biology and Epidemiology. Volume 40 of Texts in Applied Mathematics*. Springer-Verlag; 2001.
26. Intel Corporation. Intel Programmable Acceleration Card (PAC) with Intel Arria 10 GX FPGA data sheet; 2020; Document DS-1054. Accessed October 26, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/ds/ds-pac-a10.pdf
27. Intel Corporation. Intel Arria 10 core fabric and general purpose I/Os handbook; 2020. . Document A10-HANDBOOK. Accessed May 11, 2020. https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/hb/arria-10/a10_handbook.pdf
28. MyHDL Community MyHDL – from python to silicon!; 2018. http://www.myhdl.org/
29. Williams S. Icarus Verilog; 2000. http://iverilog.icarus.com
30. Intel Corporation OPAE; 2019. https://01.org/OPAE
31. Intel Corporation Intel quartus prime software suite – the intuitive high-performance design environment; 2021. https://www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html
32. Gisselquist Technology, LLC. Strategies for pipelining logic; 2017. https://zipcpu.com/blog/2017/08/14/strategies-for-pipelining.html
33. PyParsing Project PyParsing – a python parsing module; 2003. https://github.com/pyparsing/pyparsing
34. Linux Kernel Developers Linux Hugepage support; 2009. https://www.kernel.org/doc/Documentation/vm/hugetlbpage.txt
35. Xilinx, Inc. UltraScale architecture and product data sheet: overview; 2020. Document DS890 (v4.0). Accessed March 16, 2020. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf
36. Collection contains a set of sequential and parallel implementations of embedded Runge-Kutta solvers. https://github.com/UBT-AI2/rk
37. Gruber T, Eitzinger J, Hager G & Wellein G
38. NVIDIA Corporation NVIDIA management library (NVML); 2021. https://developer.nvidia.com/nvidia-management-library-nvml

How to cite this article: Bartel S, Korch M. Generation of logic designs for efficiently solving ordinary differential equations on field programmable gate arrays. *Softw Pract Exper*. 2021;1-26. doi: 10.1002/spe.3043