Custom Floating-Point Computations for the Optimization of ODE Solvers on FPGA

Serena Curzel
Politecnico di Milano, Italy

Marco Gribaudo
Politecnico di Milano, Italy

Abstract

Mean Field Analysis and Markovian Agents are powerful techniques for modeling complex systems of distributed interacting objects, for which efficient analytical and numerical solution algorithms can be implemented through linear systems of ordinary differential equations (ODEs). Solving such ODE systems on Field Programmable Gate Arrays (FPGAs) is a promising alternative to traditional CPU- and GPU-based approaches, especially in terms of energy consumption; however, the floating-point computations required are generally thought to be slow and inefficient when implemented on FPGA. In this paper, we demonstrate the use of High-Level Synthesis with automated customization of low-precision floating-point calculations, obtaining hardware accelerators for ODE solvers with improved quality of results and minimal output error. The proposed methodology does not require any manual rewriting of the solver code, but it remains prohibitively slow to evaluate any possible floating-point configuration through logic synthesis; in the future, we will thus implement automated design space exploration methods able to suggest promising configurations under user-defined accuracy and performance constraints.

2012 ACM Subject Classification Hardware → Methodologies for EDA; Hardware → High-level and register-transfer level synthesis; Computer systems organization → Architectures; Hardware → Very large scale integration design; Hardware → Reconfigurable logic and FPGAs

Keywords and phrases Differential Equations, High-Level Synthesis, FPGA, floating-point

Digital Object Identifier 10.4230/OASIcs.PARMA-DITAM.2025.2

Supplementary Material *Software*: `https://github.com/ferrandi/PandA-bambu`

Funding Funded by the European Union – NextGenerationEU – PNRR – M4 – C2 – I1.3 – SERICS PE00000014 – Cascade Funding SPOKE 8 – (MAM-CYD) – CUP J33C22002810001.

1 Introduction

Analytical modeling techniques such as Mean Field Analysis (MFA) and Markovian Agents (MA) can be applied to predict and optimize the performance of systems composed of many interacting objects, including e.g., cyber-physical systems. MFA [29] describes the transient evolution and the stationary behavior of such systems dividing their constituent objects into classes, each one describing a specific behavior [6, 12]. MA extends MFA by allowing objects, also called agents, to be distributed in a space that can be either continuous or discrete [22]; each agent has its own local behavior, which is influenced by mutual interactions with other agents. MA provides a powerful and scalable technique for modeling complex systems of distributed objects, and as such it has been applied e.g., to study sensor networks [9], Covid-19 diffusion [23], and forest fire monitoring [10].

Both MFA and MA models are analyzed using linear systems of ordinary differential equations (ODEs). One (large) vector is used to count the number of objects in each state for each class in MFA models, and MAs extend this representation by repeating these components for each considered spatial location. A kernel function defines how each element in the

© Serena Curzel and Marco Gribaudo;
licensed under Creative Commons License CC-BY 4.0
16th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 14th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2025).
Editors: Daniele Cattaneo, Maria Fazio, Leonidas Kosmidis, and Gabriele Morabito; Article No. 2; pp. 2:1–2:13
OpenAccess Series in Informatics
OASICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

state vector will evolve in time according to the definitions of the individual components, and the transient evolution of the system is computed by integrating this kernel. This representation can easily be parallelized, which motivated us to explore hardware acceleration on Field Programmable Gate Array (FPGA) or Application-Specific Integrated Circuits (ASICs) as a faster and more energy-efficient solution than software execution on general-purpose processors. ASICs are the best solution in terms of performance, but they incur higher development costs; FPGAs are more accessible and can be quickly reconfigured, allowing to update accelerators according to the requirements of new applications or to try multiple configurations in a prototyping phase before committing to long and expensive ASIC manufacturing.

The design flow we envisioned for the implementation of hardware accelerators that will solve the MFA/MA ODE systems is based on High-Level Synthesis (HLS), which allows a faster and more reliable process than manual hardware design. HLS tools, in fact, automatically generate hardware accelerators starting from a software description (e.g., written in C/C++), greatly increasing developers' productivity and allowing application experts to obtain efficient designs without being experts in low-level circuit design [11].

A typical optimization opportunity available in HLS tools is the usage of custom data types instead of the standard IEEE floating-point types used in software. In fact, FPGA implementations of floating-point functional units are usually slower than the specialized floating-point units present in modern CPUs, and they require a considerable amount of resources. If the computational precision of the application allows it, fixed-point calculations are to be preferred as they can be implemented through simpler logic. While a few previous attempts at implementing fixed-point ODE solvers exist, in this paper we focus on the exploration of custom floating-point types, i.e., types with a non-standard number of bits for mantissa and exponent, through the TrueFloat framework [19] integrated into the Bambu HLS tool [17]. The main strength of TrueFloat compared to existing libraries of floating-point components is its integration within the HLS flow, allowing deeper optimization of the functional units during the process of generating the accelerator datapath. TrueFloat types allow us to improve the quality of results (QoR) of the generated accelerators while maintaining the desired accuracy, and we demonstrate it by testing the application of different TrueFloat configurations on a proxy ODE solver representative of the type of computation performed in MFA/MA models.

TrueFloat provides the required support to implement approximate applications in HLS, but it fully relies on the user to specify the desired floating-point configuration. A key step in the design process is thus to determine the minimal precision required to maintain an acceptable output error. We argue, however, that the configuration space is too wide to be explored exhaustively and that previous research in design space exploration (DSE) tools for approximate computing are too specific to software, so further research will be required to equip Bambu and TrueFloat with a useful DSE tool able to suggest good floating-point configurations without long logic synthesis and implementation runs.

In summary, this paper makes the following contributions:

- We present a practical application of TrueFloat custom floating-point computations in the synthesis of a non-trivial program;
- We highlight the need for automated methods for the search of good floating-point configurations in a wide design space;
- We demonstrate the QoR improvement provided by custom low-precision types for an ODE solver accelerated on FPGA.

The rest of the paper is structured as follows: Section 2 summarizes related work on the acceleration of ODE solvers on FPGA and on custom-precision floating-point formats, Section 3 describes our HLS-based methodology and the exploration of different formats targeting improved QoR and minimal effect on accuracy, Section 4 presents the results we obtained after FPGA synthesis, and Section 5 concludes the paper with final remarks and future research directions.

2 Related work

2.1 Solving ODEs on FPGA

ODE systems are a fundamental component of many scientific applications in high-performance computing, and a significant amount of research focuses on improving the performance of the numerical methods used to solve them. FPGA-based solutions have been explored because of their potential to provide high throughput and low energy consumption, making them an attractive alternative to GPUs and multi-core CPUs despite the steep learning curve of low-level hardware design.

The Differential Equation Processing Element (DEPE) co-processor [27, 28, 26] has been proposed as an alternative to customizing an FPGA accelerator for the solution of a specific ODE system. The co-processor was designed as a no-instruction-set computer together with its compiler and it solves ODEs with a Runge-Kutta fourth-order method implemented through fixed-point computations; the evaluation focuses on its low area consumption compared to HLS-generated designs. Another co-processor based on the RISC-V architecture was proposed targeting a specific class of ODEs [25], implementing Euler and Runge-Kutta methods through single-precision floating-point computations and obtaining faster execution time than a single-core CPU. Other research works present custom solver units where both the solver method and the ODEs are hard-coded into the accelerator [4, 5].

HLS tools raise the level of abstraction required to design FPGA accelerators [11], and they have been applied to implement accelerators for various ODE solvers [35]. Another possibility to simplify the design flow of ODE accelerators is to rely on domain-specific languages and map their primitives to optimized register-transfer level (RTL) primitives [3] or to exploit commercial tools provided within Matlab/Simulink to generate RTL components from model-based representations [1].

New computational models have also been proposed in place of conventional numerical solvers, as they might provide an advantage when implemented in custom hardware. For example, since analog components can solve ODEs faster than numerical methods, a possibility is to simulate analog components in FPGA logic through digital differential analyzers [15, 21]. The Euler solver can also be approximated through the use of stochastic integrators, which can be efficiently implemented as RTL components [30].

2.2 Approximate computing

Research in the field of approximate computing led to a variety of techniques that aim at improving the energy efficiency of an application through reduced precision, and tools that analyze the application to understand the impact of approximation on the quality of its results [31, 13].

The problem of finding the best approximation scheme has been addressed through both analytical models and learning-based approaches, mostly focusing on the few precision levels available in existing general-purpose CPUs (single- and double-precision floating-point

as defined by IEEE standards). Precimonious [33], for example, implements a dynamic program analysis pass that, given a set of representative inputs and a target accuracy, aims at improving the runtime of the program by reducing its precision. The tool automatically performs a search of the configuration space and suggests a floating-point type for each variable in the program which can be implemented with lower precision (converting from `long double` to `double` or `float`). The work presented by Ho et al. [24] focuses on reducing the number of mantissa bits to be assigned to floating-point variables through the GNU Multiple Precision Floating-Point Routines (MPFR) [20]; the search for the best solution is implemented in Python by running multiple versions of the application with different precision levels, and it can be extended to find a fixed-point configuration suitable for FPGAs. SmartFPTuner [8], instead, introduces a machine learning component that predicts the output error generated by a reduced precision configuration, and uses mathematical programming to search for the smallest possible format for each variable among those supported by a custom RISC-V platform; such a combined approach results in much faster time-to-solution.

For what concerns FPGA implementations of approximate computing, multiple RTL and HLS libraries exist that provide optimized implementations of fixed- and floating-point operators with arbitrary precision. VFLOAT [37] and FloPoCo [14] provide VHDL components for custom-precision floating-point arithmetic that users have to manually integrate into their designs. The AdaptivFloat representation [36] was motivated by the precision requirements of deep learning applications on FPGA, but the implementation of corresponding arithmetic units still requires significant manual effort. Proprietary HLS tools map fixed- and floating-point C++ types with arbitrary precision onto a back-end RTL library during the synthesis process [2, 34]. TrueFloat [19] stands out as it not only provides a library of customizable floating-point operators with arbitrary precision, but also is fully integrated within the HLS tool Bambu [17]; its results are competitive with hand-designed library operators when applied to isolated functional units, and it outperforms commercial HLS tools when synthesizing whole applications because it allows Bambu to optimize floating-point operators rather than treating them as black boxes.

3 Methodology

3.1 High-Level Synthesis

We based our design flow on High-Level Synthesis (HLS) because it allows us to quickly translate existing MFA- and MA-based C applications into FPGA accelerators, and to evaluate different configurations without having to manually modify the RTL design. For example, the output of HLS tools is a description in low-level Verilog/VHDL, tailored to a specific FPGA board to extract the best performance in terms of latency (number of cycles, clock frequency), resource consumption (number of FPGA resources used among the different categories of logic and memory elements available), or power consumption. It is possible to specify directives that prioritize one of these metrics over the other, to change the hardware target, or to request specific backend optimizations starting from the same input program with no manual rewriting of the code.

FPGA vendors typically offer HLS as part of their design toolkits (e.g., Vitis HLS from AMD/Xilinx or the Intel HLS compiler), but such tools only support FPGA boards from a single vendor, and they cannot be modified, as their source code is proprietary. Instead, we exploit the open-source HLS framework Bambu [17], which facilitates research into new design automation methods. Bambu has been an invaluable tool in previous projects where knowledge of the internal HLS process and the possibility of modifying it were crucial to generate better accelerators in a specific domain (e.g., big data [32], aerospace [18]).

Bambu supports most C/C++ constructs, including function calls, access to arrays and structs, parameters passed by reference or copy, pointer arithmetic, dynamic resolution of memory accesses, and module sharing. Moreover, it can also take as input intermediate representations from the GCC and Clang compilers, leading to the possibility of direct integration between Bambu and compiler-based toolchains [7]. The HLS flow in Bambu is similar to a software compilation process, beginning with a high-level specification and generating low-level code through a series of analysis and optimization steps divided into three phases (front-end, middle-end, and back-end). In the front-end, Bambu parses the input code and translates it into an intermediate representation (IR), while numerous target-independent analyses and optimizations are performed in the middle-end. The back-end performs the actual synthesis of Verilog/VHDL code ready for simulation, logic synthesis, and implementation on FPGA or ASIC through external tools.

When a software description is translated into a hardware accelerator through HLS, there are ample opportunities to include optimizations that drastically impact the QoR in terms of performance, area, and energy consumption. One common example is loop unrolling: replicating the instructions of independent loop iterations allows implementing them in parallel in the accelerator, increasing performance at the expense of resource consumption. Alongside techniques that aim at exploiting different degrees of parallelism present in the input applications, another possibility to generate efficient accelerators in terms of performance per area is to explore the usage of custom data types, avoiding the generation of floating-point functional units which are usually inefficient when compared to the specialized floating-point arithmetic units of modern CPUs and GPUs.

In this paper, we focus on the customization of floating-point computations in isolation, disregarding all other optimization opportunities available in Bambu. While this will likely result in sub-optimal results in absolute terms, which would not justify the choice of offloading computations to FPGA, our aim is to conduct an ablation study to understand the impact of this specific type of optimization on the QoR of generated accelerators, and on the application accuracy. We will then exploit information gathered from this study during future experiments with MFA- and MA-based applications to be accelerated.

3.2 TrueFloat

Experiments with custom data types are usually limited by the back-end libraries supported by HLS tools, mostly focused on fixed-point types; however, Bambu also integrates the dedicated TrueFloat framework for the generation of custom floating-point types. TrueFloat allows users to specify custom formats for floating-point computations and automatically synthesizes corresponding optimized arithmetic units; different TrueFloat encodings can be specified for different parts of the input application, resulting in the generation of multi-precision accelerators. The main strength of TrueFloat is the integration within the HLS process, providing effortless translation between different floating-point encodings through simple command-line options and integration with other optimization techniques present in the HLS flow. TrueFloat also opens the possibility of generating an equivalent representation of the synthesized accelerator at a higher level of abstraction, which could be used for fast and accurate software simulation.

Figure 1 describes the TrueFloat synthesis flow within Bambu. The input code is in one of the standard formats supported by Bambu, and it contains standard floating-point operations and types. The user expresses one or more required custom representations (one for each function in the input code) through command-line options, and the HLS flow autonomously handles type replacement, conversions, and custom arithmetic units generation, avoiding manual and error-prone code rewriting.

Figure 1 TrueFloat design methodology as presented in [19].

A compiler step called FPBlender handles all floating-point operations within the HLS flow in Bambu, exploiting information generated during previous analysis steps on the intermediate representation of the input program and allowing subsequent steps to apply more accurate optimizations after the custom floating-point format has been implemented. FPBlender generates ad-hoc functional units exploiting the TrueFloat library of templatized components, which contains optimized implementations for basic arithmetic operators (addition, subtraction, multiplication, division, and comparison) and bidirectional type conversion operators (floating-point to integer, integer to floating-point, and floating-point to floating-point). The TrueFloat library components are soft-float implementations in C built by combining basic integer operations; input and output parameters are defined as unsigned integers as well. All functions have arguments representing standard operands followed by a set of eight specialization arguments to indicate the number of exponent bits, fractional bits, the exponent bias, the rounding mode, the exception mode, whether hidden one is enabled, whether subnormals are enabled, and the sign mode.

Users of TrueFloat should explicitly define a floating-point format for each function they want to customize in the input code through the `-fp-format` command-line option, which Bambu will use to replace the standard single- or double-precision data type present in the input file. In particular, `-fp-format` requires the name of the function that will be customized and a string that encodes the requested format. Functions called by the selected function will be implemented with standard types unless –fp-format-propagate is set, instructing Bambu to propagate the custom data type to all called functions. Besides the choice of the number of bits for mantissa and exponent, TrueFloat also allows tuning settings such as whether subnormals are supported or not, as they result in simpler logic and lower resource consumption. The format string following the function name is composed as follows:

```
e<exp_bits>m<frac_bits>b<exp_bias><rnd_mode><exc_mode><spec><sign>
```

The number of bits requested for the exponent and for the mantissa are set through `exp_bits` and `frac_bits`, and `exp_bias` indicates the bias added to the unsigned value represented by the exponent bits. The rounding mode `rnd_mode` can be either nearest even, which is the IEEE standard rounding mode, or truncate, where no rounding is applied. The exception mode `exc_mode` can be set to require IEEE standard exceptions, to saturation, where infinite is replaced with the highest possible value and not-a-number results in undefined behavior, or to overflow, where both infinite and not-a-number result in undefined behavior. Finally, with `spec` it is possible to select whether to enable the IEEE standard representation with hidden one and subnormal numbers, while `sign` specifies whether all values should be considered as negative numbers, positive numbers, or if IEEE dynamic sign should be used.

Accelerators that use floating-point types with lower precision have higher performance and to use fewer FPGA resources with respect to designs based on standard `float` or `double` calculations. Moreover, TrueFloat arithmetic operators have been shown to have similar or better performance than other implementations from state-of-the-art libraries [19]. However, to the best of our knowledge, this is the first time that TrueFloat gets applied to a realistic input application beyond the single arithmetic operation or small kernel.

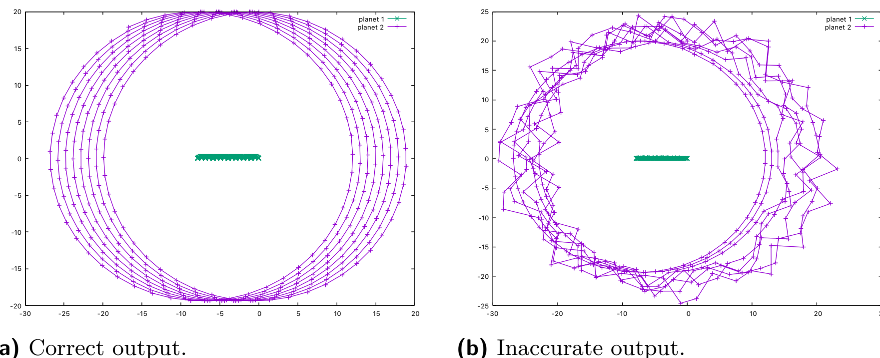
TrueFloat provides the required support to implement approximate applications in HLS, but it fully relies on the user to specify the desired floating-point configuration. Unfortunately, none of the approaches described in Section 2.2 can be directly applied to find an optimal TrueFloat configuration within an accuracy constraint. In fact, tools and techniques designed for software applications tend to consider only a few possible formats, which are the ones supported by the target CPU, while the design space of possible TrueFloat formats is much larger: it is possible to specify the exponent bitwidth and the mantissa bitwidth independently, and tune several other configuration options. Moreover, TrueFloat controls floating-point precision on a per-function level instead of variable by variable. Finally, the performance target for a hardware accelerator possibly includes resources consumption besides latency, and both metrics can only be assessed reliably after long logic synthesis runs. A learning-based approach would also not be feasible due to the absence of a reference dataset for our target application. Under these circumstances, manually sampling a limited number of points in the design space remains the simplest but most effective method to explore the trade-off between accelerator performance and accuracy of the results. Fortunately, this does not require any manual code rewriting, as TrueFloat automatically replaces the types of all variables and operations according to the specified Bambu command-line options.

4 Experimental results

4.1 Target application and configuration exploration

To keep synthesis times within reasonable limits (e.g., less than one hour per configuration), we evaluated a proxy application in place of the performance modeling use case. The proxy application solves the n-body problem through a Runge-Kutta-Fehlberg method with adaptive step size (RKF45) [16], and it has been implemented in C. The gravity interactions between bodies simulated by the application resemble the patterns of interaction among multiple agents in our target MFA/MA model, and the same RKF45 method will be used to solve the ODEs modeling the evolution of the cyber-physical system under investigation. Therefore, results from the synthesis of the n-body application can be used to decide which floating-point configurations shall be evaluated in the synthesis of the MFA/MA accelerator, keeping in mind that the requested precision may change when the number of interacting agents increases.

Our target application contains the kernel function to be accelerated (`solve`) together with a main program that runs the n-body simulation and plots its outputs. By looking at the output plots, it is easy to assess if the solution is converging correctly. Figure 2 shows the output plots obtained running simulations of the motion of two planets, i.e., the evolution of their orbits starting from an initial condition where the planet with the smallest mass has non-zero speed (purple trace) and the planet with the biggest one is still (green trace). The correct plot in Figure 2a is obtained with double-precision floating-point calculations, while using an extremely low precision in part of the simulation may lead to an incorrect plot such as the one in Figure 2b. Thanks to its advanced co-simulation features, Bambu is able to use the main function of the target application as a testbench to verify the correct behavior of



(a) Correct output. **(b)** Inaccurate output.

Figure 2 Plots generated by the target application with different floating-point accuracies.

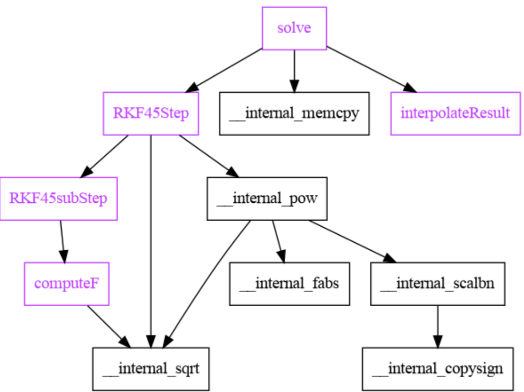


Figure 3 Call graph of the target application.

the accelerated kernel function; it is therefore possible to plot similar output graphs with results from the RTL simulation of the generated accelerator. TrueFloat configurations with unacceptably low precision will then be discarded either because the simulation does not produce a result within a given time limit or by comparing their outputs with Figure 2a.

As TrueFloat allows controlling floating-point configurations at the function level, it is important to analyze the call graph of the target application (Figure 3). The `solve` kernel function repeatedly calls subfunctions `interpolateResult` and `RKF45Step`; the latter contains a loop calling subfunction `RKF45SubStep`, which in turn calls multiple times the `computeF` function. The application also includes calls to floating-point mathematical functions from the C standard library. The amount of floating-point operations executed by the application partially depends on the floating-point format used, as there are loops that terminate based on the achieved precision; in a double-precision run, there are approximately 200.000 floating-point additions and just as many multiplications, floating-point divisions and other mathematical functions are less than 1000.

We kept the double-precision version of the target application as a baseline (configuration 0 in Table 1) and selected six TrueFloat configurations to compare against it, all of which produced correct simulation results. We first reduced the number of bits for mantissa and exponent for the whole accelerator, specifying a format for `solve` in the Bambu command-line options and requesting the propagation of the format to all called functions. The specialization string was selected to obtain IEEE-compliant behavior. The minimum number of bits was found to be 22 for the mantissa and 7 for the exponent, scaling the bias accordingly

Table 1 List of floating-point configurations to be evaluated.

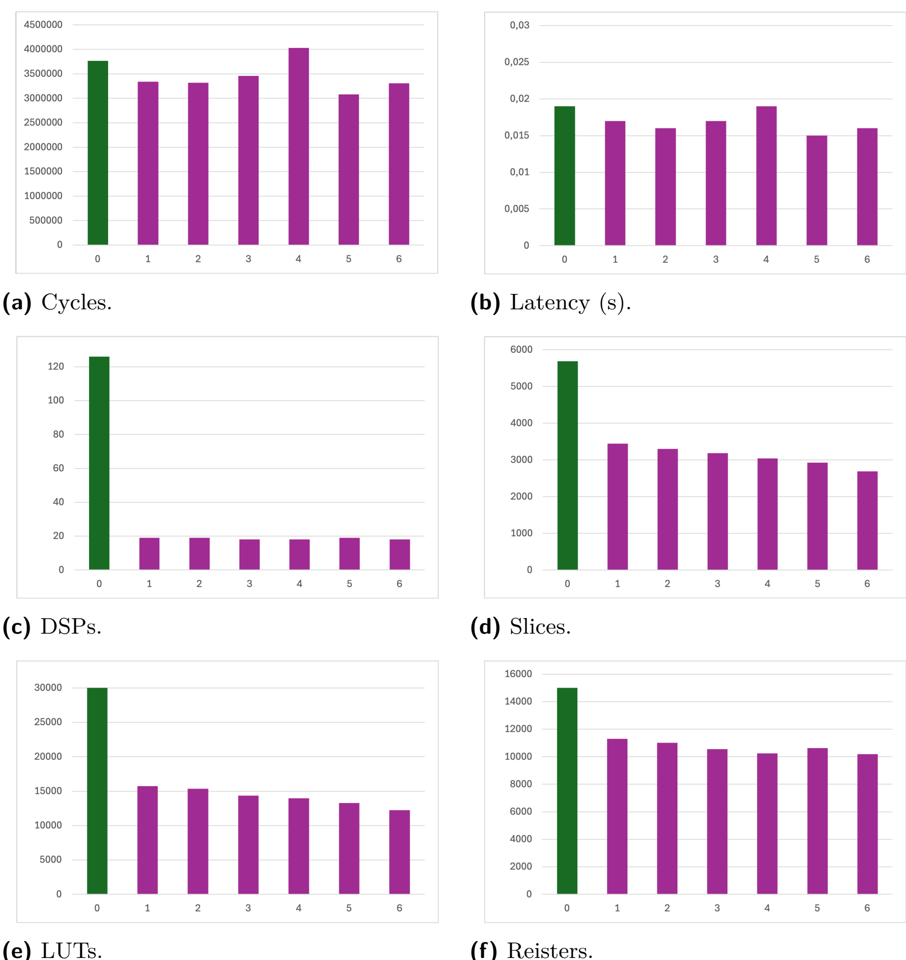
Configuration	Function affected	Exponent bits (bias)	Mantissa bits	Specialization string
0	`solve`	11 (-1023)	52	nihs
1	`solve`	8 (-127)	23	nihs
2	`solve`	7 (-63)	22	nihs
3	`solve`	7 (-63)	22	nihs
	`interpolateResult`	6 (-31)	10	nihs
4	`solve`	7 (-63)	22	nihs
	`interpolateResult`	6 (-31)	10	nihs
	`RKF45subStep`	6 (-31)	21	nihs
5	`solve`	7 (-63)	22	tih
6	`solve`	7 (-63)	22	tih
	`interpolateResult`	6 (-31)	10	tih

(configuration 2); trimming the bitwidths further resulted in no convergence or unacceptable errors in the output plots. Interestingly, this means that IEEE single-precision floating-point would also be an acceptable configuration, and so we included that in the evaluation (configuration 1). Next, we tested whether we could further reduce the number of bits with respect to configuration 2 by selecting a different format for one or more subfunctions (configurations 3 and 4); in fact, lowering the precision of functions that contain fewer operations has a smaller impact on the accelerator QoR, but also a smaller impact on the application accuracy. Finally, we acted on the specialization string of configurations 2 and 3 removing support for subnormal numbers and using truncation instead of rounding to nearest even (configurations 5 and 6), as this was previously shown to be beneficial, especially in terms of area consumption [19].

4.2 Synthesis results

We synthesized six versions of the n-body application with Bambu according to the configurations of Table 1 and evaluated the QoR of the generated accelerators after place-and-route (p&r). We chose an Alveo U55C FPGA from AMD/Xilinx as target and requested a clock period of 5ns, which was achieved by all configurations; all other Bambu options were left as default in order to focus on the effects of customizing floating-point formats on performance. We evaluated and reported in Figure 4 the number of clock cycles from simulation, the latency considering the achieved frequency post p&r, and area consumption in terms of number of digital signal processing blocks (DSPs, the scarcest resource on FPGA and required to implement floating-point multiplications), slices, lookup tables (LUTs), and registers.

Green bars represent synthesis results for configuration 0, i.e., the 64-bit double-precision baseline; all other configurations use floating-point formats with fewer bits, which translates directly into fewer resources consumed (Figures 4c-4f). The comparison is especially striking in Figure 4c, as the other configurations use $\sim 85\%$ fewer DSPs. Looking at performance (Figures 4a and 4b), however, configuration 0 is not the slowest one: configurations 3, 4, and 6 perform worse than configurations 1, 2, and 5, with configuration 4 even running slower than the double-precision baseline. Despite using custom formats with fewer bits, in fact, these configurations have to pay the price of converting data between different types every time that one of the affected subfunctions is called. (It is likely possible to restructure the code to mitigate this issue, e.g., by unrolling loops that call the same function multiple times.) The suggestions of moving from rounding to truncation and of dropping support for subnormals were proven to be beneficial, as configurations 5 and 6 result in better QoR than configurations 2 and 3 along all considered metrics.



(a) Cycles. **(b)** Latency (s).

(c) DSPs. **(d)** Slices.

(e) LUTs. **(f)** Reisters.

Figure 4 Synthesis results for six different floating-point configurations.

From these results, we can conclude that using custom floating-point formats generated by TrueFloat is highly beneficial to reduce the area consumption of ODE solvers accelerated on FPGA. The latency of the generated accelerators can decrease when reducing the number of bits used for floating-point computations, but other factors, such as the amount of required conversion operators, may counter the improvement. In absolute terms, the accelerators we generated are quite slow (~ 7 times higher latency than software execution); however, we did not exploit any of the available parallelism in the application, and thus we are confident that it would not take too much effort to obtain higher-performance versions of the accelerator. Even in the worst case that we evaluated, the resources required occupy less than $\sim 4\%$ of the target FPGA, suggesting that, for example, aggressive unrolling of loops should be possible (considering also that the application is highly compute-intensive and thus memory bandwidth should not be a bottleneck if intermediate results are kept on-chip).

5 Conclusion

We applied the TrueFloat framework to an ODE solver performing double-precision floating-point calculations to generate FPGA accelerators with custom precision, aiming at studying the effect of lower mantissa and exponent bitwidth on performance and area consumption.

The results we obtained highlight the possible savings in terms of resources and latency but also the care required to choose the best floating-point configuration in a wide design space where the loss of accuracy in the application output is also a concern; future research on automated design space exploration will undoubtedly improve the usability of TrueFloat.

References

1 Hassan Al-Yassin, Mohammed A. Fadhel, Omran Al-Shamma, and Laith Alzubaidi. Solving Lorenz ODE System Based Hardware Booster. In *Intelligent Systems Design and Applications (ISDA)*, pages 245–254, 2019. doi:10.1007/978-3-030-49342-4_24.

2 AMD/Xilinx. Arbitrary Precision Data Types Library, 2024. URL: https://docs.amd.com/r/en-US/ug1399-vitis-hls/Arbitrary-Precision-AP-Data-Types.

3 Silas Bartel and Matthias Korch. Generation of logic designs for efficiently solving ordinary differential equations on field programmable gate arrays. *Software: Practice and Experience*, 53(1):27–52, 2023. doi:10.1002/spe.3043.

4 Soham Bhattacharya and Dwaipayan Chakraborty. Design-Space Exploration of the Runge-Kutta Hardware Accelerator for Solving Ordinary Differential Equation. In *2023 IEEE International Conference on Electrical, Automation and Computer Engineering (ICEACE)*, pages 260–264, 2023. doi:10.1109/ICEACE60673.2023.10442673.

5 Soham Bhattacharya and Dwaipayan Chakraborty. Implementation of a Hardware Accelerator with FPU-Based Euler and Modified Euler Solver For an Ordinary Differential Equation. In *2023 International Conference on Computational Science and Computational Intelligence (CSCI)*, pages 1106–1112, 2023. doi:10.1109/CSCI62032.2023.00182.

6 Andrea Bobbio, Marco Gribaudo, and Miklós Telek. Analysis of Large Scale Interacting Systems by Mean Field Method. In *2008 Fifth International Conference on Quantitative Evaluation of Systems*, pages 215–224, 2008. doi:10.1109/QEST.2008.47.

7 Nicolas Bohm Agostini, Serena Curzel, Jeff Jun Zhang, Ankur Limaye, Cheng Tan, Vinay Amatya, Marco Minutoli, Vito Giovanni Castellana, Joseph Manzano, David Brooks, Gu-Yeon Wei, and Antonino Tumeo. Bridging Python to Silicon: The SODA Toolchain. *IEEE Micro*, 42(5):78–88, 2022. doi:10.1109/MM.2022.3178580.

8 Andrea Borghesi, Giuseppe Tagliavini, Michele Lombardi, Luca Benini, and Michela Milano. Combining learning and optimization for transprecision computing. In *Proceedings of the 17th ACM International Conference on Computing Frontiers*, pages 10–18, 2020. doi:10.1145/3387902.3392615.

9 Dario Bruneo, Marco Scarpa, Andrea Bobbio, Davide Cerotti, and Marco Gribaudo. Analytical modeling of swarm intelligence in wireless sensor networks through Markovian agents. In *Proceedings of the Fourth International ICST Conference on Performance Evaluation Methodologies and Tools*, 2009. doi:10.4108/ICST.VALUETOOLS2009.7672.

10 Lelio Campanile, Mauro Iacono, Fiammetta Marulli, Marco Gribaudo, Michele Mastroianni, et al. A DSL-Based Modeling Approach For Energy Harvesting IoT/WSN. In *36th International ECMS Conference on Modelling and Simulation*, pages 317–323, 2022. doi:10.7148/2022-0317.

11 Jason Cong, Jason Lau, Gai Liu, Stephen Neuendorffer, Peichen Pan, Kees Vissers, and Zhiru Zhang. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems*, 15(4):1–42, 2022. doi:10.1145/3530775.

12 Francesca Cordero, Daniele Manini, and Marco Gribaudo. Modeling Biological Pathways: An Object-Oriented like Methodology Based on Mean Field Analysis. In *2009 Third International Conference on Advanced Engineering Computing and Applications in Sciences*, pages 117–122, 2009. doi:10.1109/ADVCOMP.2009.25.

13 Ayad M. Dalloo, Amjad Jaleel Humaidi, Ammar K. Al Mhdawi, and Hamed Al-Raweshidy. Approximate Computing: Concepts, Architectures, Challenges, Applications, and Future Directions. *IEEE Access*, 12:146022–146088, 2024. doi:10.1109/ACCESS.2024.3467375.

14 F. de Dinechin and B. Pasca. Designing Custom Arithmetic Data Paths with FloPoCo. *IEEE Design & Test of Computers*, 28(4):18–27, 2011. doi:10.1109/MDT.2011.44.

15 Alireza Fasih, Tuan Do Trong, Jean Chamberlain Chedjou, and Kyandoghere Kyamakya. New computational modeling for solving higher order ODE based on FPGA. In *2009 2nd International Workshop on Nonlinear Dynamics and Synchronization*, pages 49–53, 2009. doi:10.1109/INDS.2009.5227969.

16 Erwin Fehlberg. *Low-order classical Runge-Kutta formulas with stepsize control and their application to some heat transfer problems*, volume 315. National aeronautics and space administration, 1969.

17 Fabrizio Ferrandi, Vito Giovanni Castellana, Serena Curzel, Pietro Fezzardi, Michele Fiorito, Marco Lattuada, et al. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*, pages 1327–1330, 2021. doi:10.1109/DAC18074.2021.9586110.

18 Fabrizio Ferrandi, Michele Fiorito, Claudio Barone, Giovanni Gozzi, and Serena Curzel. High-Level Synthesis Developments in the Context of European Space Technology Research. In *15th Workshop on Parallel Programming and Run-Time Management Techniques for Many-Core Architectures and 13th Workshop on Design Tools and Architectures for Multicore Embedded Computing Platforms (PARMA-DITAM 2024)*, volume 116, pages 1:1–1:12, 2024. doi:10.4230/OASIcs.PARMA-DITAM.2024.1.

19 Michele Fiorito, Serena Curzel, and Fabrizio Ferrandi. TrueFloat: A Templatized Arithmetic Library for HLS Floating-Point Operators. In *Embedded Computer Systems: Architectures, Modeling, and Simulation: 23rd International Conference, SAMOS 2023, Samos, Greece, July 2–6, 2023, Proceedings*, pages 486–493, 2023. doi:10.1007/978-3-031-46077-7_35.

20 Laurent Fousse, Guillaume Hanrot, Vincent Lefèvre, Patrick Pélissier, and Paul Zimmermann. MPFR: A Multiple-Precision Binary Floating-Point Library with Correct Rounding. *ACM Trans. Math. Softw.*, 33(2), 2007. doi:10.1145/1236463.1236468.

21 Jonathan Garcia-Mallen, Shuohao Ping, Alex Miralles-Cordal, Ian Martin, Mukund Ramakrishnan, and Yipeng Huang. Towards an Accelerator for Differential and Algebraic Equations Useful to Scientists. *IEEE Computer Architecture Letters*, 22(2):185–188, 2023. doi:10.1109/LCA.2023.3332318.

22 Marco Gribaudo, Davide Cerotti, and Andrea Bobbio. Analysis of On-off policies in Sensor Networks Using Interacting Markovian Agents. In *2008 Sixth Annual IEEE International Conference on Pervasive Computing and Communications (PerCom)*, pages 300–305, 2008. doi:10.1109/PERCOM.2008.100.

23 Marco Gribaudo, Mauro Iacono, and Daniele Manini. COVID-19 Spatial Diffusion: A Markovian Agent-Based Model. *Mathematics*, 9(5), 2021. doi:10.3390/math9050485.

24 Nhut-Minh Ho, Elavarasi Manogaran, Weng-Fai Wong, and Asha Anoosheh. Efficient floating point precision tuning for approximate computing. In *2017 22nd Asia and South Pacific Design Automation Conference (ASP-DAC)*, pages 63–68, 2017. doi:10.1109/ASPDAC.2017.7858297.

25 Andrew Hollabough and Dwaipayan Chakraborty. An Open-Source Co-processor for Solving Lotka-Volterra Equations. In *2022 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 1690–1694, 2022. doi:10.1109/ISCAS48785.2022.9937835.

26 Chen Huang, Bailey Miller, Frank Vahid, and Tony Givargis. Synthesis of networks of custom processing elements for real-time physical system emulation. *ACM Trans. Des. Autom. Electron. Syst.*, 18(2), 2013. doi:10.1145/2442087.2442092.

27 Chen Huang, Frank Vahid, and Tony Givargis. A Custom FPGA Processor for Physical Model Ordinary Differential Equation Solving. *IEEE Embedded Systems Letters*, 3(4):113–116, 2011. doi:10.1109/LES.2011.2170152.

28 Chen Huang, Frank Vahid, and Tony Givargis. Automatic synthesis of physical system differential equation models to a custom network of general processing elements on FPGAs. *ACM Trans. Embed. Comput. Syst.*, 13(2), 2013. doi:10.1145/2514641.2514650.

29 Thomas G. Kurtz. Solutions of Ordinary Differential Equations as Limits of Pure Jump Markov Processes. *Journal of Applied Probability*, 7(1):49–58, 1970. URL: http://www.jstor.org/stable/3212147.

30 Siting Liu and Jie Han. Hardware ODE Solvers using Stochastic Circuits. In *Proceedings of the 54th Annual Design Automation Conference (DAC)*, 2017. doi:10.1145/3061639.3062258.

31 Sparsh Mittal. A Survey of Techniques for Approximate Computing. *ACM Comput. Surv.*, 48(4), 2016. doi:10.1145/2893356.

32 Christian Pilato, Subhadeep Banik, Jakub Beránek, Fabien Brocheton, Jeronimo Castrillon, Riccardo Cevasco, et al. A System Development Kit for Big Data Applications on FPGA-based Clusters: The EVEREST Approach. In *2024 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1–6, 2024. doi:10.23919/DATE58400.2024.10546518.

33 Cindy Rubio-González, Cuong Nguyen, Hong Diep Nguyen, James Demmel, William Kahan, Koushik Sen, et al. Precimonious: Tuning assistant for floating-point precision. In *SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–12, 2013. doi:10.1145/2503210.2503296.

34 Siemens Digital Industries Software. HLS Libs, 2024. URL: https://hlslibs.org/.

35 Ioannis Stamoulias, Matthias Möller, Rene Miedema, Christos Strydis, Christoforos Kachris, and Dimitrios Soudris. High-Performance Hardware Accelerators for Solving Ordinary Differential Equations. In *Proceedings of the 8th International Symposium on Highly Efficient Accelerators and Reconfigurable Technologies (HEART)*, 2017. doi:10.1145/3120895.3120919.

36 T. Tambe, E. Y. Yang, Z. Wan, Y. Deng, V. Janapa Reddi, et al. Algorithm-Hardware Co-Design of Adaptive Floating-Point Encodings for Resilient Deep Learning Inference. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020. doi:10.1109/DAC18072.2020.9218516.

37 Xiaojun Wang and Miriam Leeser. VFloat: A Variable Precision Fixed- and Floating-Point Library for Reconfigurable Hardware. *ACM Trans. Reconfigurable Technol. Syst.*, 3(3), 2010. doi:10.1145/1839480.1839486.