Towards an Accelerator for Differential and Algebraic Equations Useful to Scientists

Jonathan Garcia-Mallen, Shuohao Ping, Alex Miralles-Cordal, Ian Martin, Mukund Ramakrishnan, Yipeng Huang

***Abstract*—We discuss our preliminary results in building a configurable accelerator for differential equation time stepping and iterative methods for algebraic equations. Relative to prior efforts in building hardware accelerators for numerical methods, our focus is on the following: 1) Demonstrating a higher order of numerical convergence that is needed to actually support existing numerical algorithms. 2) Providing the capacity for wide vectors of variables by keeping the hardware design components as simple as possible. 3) Demonstrating configurable hardware support for a variety of numerical algorithms that form the core of scientific computation libraries. These efforts are toward the goal of making the accelerator democratically accessible by computational scientists.**

***Index Terms*—Reconfigurable hardware, Iterative methods, Hyperbolic equations**

I. Introduction

The central challenge of making domain-specific accelerators (DSAs) useful to many people is to find the right balance between configurability and ease-of-use. In one extreme, accelerators for cryptographic ciphers, media codecs, and network protocols have predefined standardized functionality and are already widely useful for consumers. In the other extreme, DSAs such as field-programmable gate arrays (FPGAs) are blank canvases that can only be harnessed by engineers. Democratized use of DSAs is an ongoing search for ways to constrain the configurability of hardware that make such hardware thereby useful for a broader set of computer users.

Scientific computing is one of the most impactful applications of accelerators. Democratized use of DSAs in this domain would entail allowing computational scientists to take a numerical problem and turn it into a problem-specific accelerator configuration. Despite recent progress, the state-of-the-art falls short of that goal in several ways: 1) GPUs are the only kind of DSA that are useful to most computational scientists. Despite the fact that FPGAs are widely available, few scientists can dedicate the engineering effort to map problems to configurable hardware. 2) Outside of a handful of examples where GPUs can be called from the languages and libraries that scientists use [1], [2], most of the functionality of GPUs needs kernel-level programming or at least familiarity with CUDA libraries. 3) In cases where specialized hardware has been shown to help with scientific computing, the case studies fail to demonstrate numerical properties that are important to computational scientists. The goal should be to demonstrate the expected convergence rates to the correct answers for a variety of problems.

J. Garcia-Mallen, S. Ping, A. Miralles-Cordal, I. Martin, M. Ramakrishnan, and Y. Huang are with the Department of Computer Science, Rutgers University, New Brunswick, New Jersey

Fortunately, the algorithms in scientific computing provide a clear path for democratic use of configurable DSAs. The heart of nearly all scientific computing numerical methods is a differential equation *time stepping* and/or an *iterative method* solver for algebraic equations. These numerical kernels feature common traits that DSAs can directly support: 1) A wide vector of (thousands of) variables is updated over many (up to millions of) iterations to reach a converged steady state solution. 2) The vector elements are interrelated via a fixed, sparse stencil. 3) The vector elements change each iteration by relatively small increments relative the full range of values.

In this work, we build a DSA that targets the above traits and has an adequate degree of configurability to support a variety of time stepping and iterative methods. We demonstrate the ability to solve benchmark problems such as solving a non-linear wave partial differential equation. The solver hardware demonstrates a high degree of numerical convergence, that is, a polynomial reduction of error relative to increasing the number of algorithm iterations, an essential numerical property that has not been demonstrated in prior work in configurable DSAs for scientific computing [3], [4], [5], [6], [7], [8], [9], [10], [11], [12]. The hardware design is able to support vector sizes of thousands of variables to provide low-latency solutions for real-world problems.

Equally important, we envision democratic use of such a DSA. We have made steps toward an end-to-end pipeline that supports numerical algorithms written in the Julia language, an increasingly important domain-specific language for computational science. Selected time stepping and iterative methods in that language can be mapped to the DSA without manual configuration, enabling access by computational scientists.

II. Tutorial: Generating sine and cosine by tallying increments

This section gives a tutorial example of generating the sine and cosine functions using hardware cells that can be composed as building blocks for numerical algorithms. These hardware cells consist of only a few registers, but they can form a variety of differential equation time stepping and algebraic equation iterative methods, with each algorithm step taking place within a fixed number of hardware clock cycles. We also validate that the hardware design achieves a high degree of numerical convergence, a property that has not been emphasized in prior work.

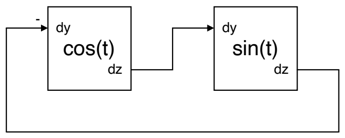

Fig. 1. Two hardware cells in a feedback loop for generating sine and cosine as the solution to an ODE IVP.

The sine and cosine functions can be generated as solutions to the ordinary differential equation (ODE) initial value problem (IVP):

$$y(t) = \begin{bmatrix}\sin t\\ \cos t\end{bmatrix}, \frac{dy}{dt} = f(y) = \begin{bmatrix}0 & 1\\ -1 & 0\end{bmatrix} y, y(0) = \begin{bmatrix}0\\ 1\end{bmatrix} \quad (1)$$

A. A simplistic first-order approach in hardware

The most basic differential equation time stepping numerical method for IVPs is the Euler method:

$$y_{n+1} = y_n + hf(t_n, y_n),$$

where y_n is the value of y at time step n, and h is the duration of each time step.

This Euler method time stepping can be realized using a cell with a (signed integer) register y holding the present value of y. The hardware cell takes as input dy, an update to y, and gives as output dz, an indication of how much y has changed in a time step. The cell would perform each time step in two clock cycles as follows:

```
phase 1: [dz,r] <= r + dt*y;
phase 2: y <= y + dy;
```

In the above register-transfer level specification, r is an accumulator register tracking changes to y; [dz,r] indicates that dz is the signed most-significant bits of the sum while r is the residual unsigned least-significant bits. dz is transmitted to other cells as output while r is stored locally in the cell representing a remainder. dt encodes h, and is scaled so that it is a power of two such that bitshifts supplant costly integer multiplication. The $\sin t$ and $\cos t$ components of vector y in Equation (1) would each be integrated in two cells connected in a feedback loop as indicated in Fig. 1. Fig. 2 shows the internal state of the cell for $\sin t$ and its output.

The hardware approach demonstrated here is called a *digital differential analyzer* and it has a history dating to early digital computers in the mid-20th century [13], [14], [15], [16].

B. Generalized hardware cell for higher-order integration

In practice, solving ODE IVPs needs higher order integration methods where a linear increase in time stepping resolution leads to a polynomial reduction in the solution error. More specifically, we say an integration method is of order s if an increase in resolution by factor h leads to a error reduction by factor h^s. Fig. 3 shows the effect of higher order integration on solution accuracy.

Higher order integration can be done using the hardware cells by correctly generalizing the update rules. Phase 1 of each time step stays the same as the Euler cell, while for a method of order s, phases $i = 2$ through $i = s$ perform the following register transfers:

phase i: $\texttt{dy}_{i-1}$ <= dy;

$$\texttt{[dz,r]<=r+dt*(y+}\sum_{j=1}^{i-2} a_{ij}\texttt{*dy}_j + a_{i,i-1}\texttt{*dy);}$$

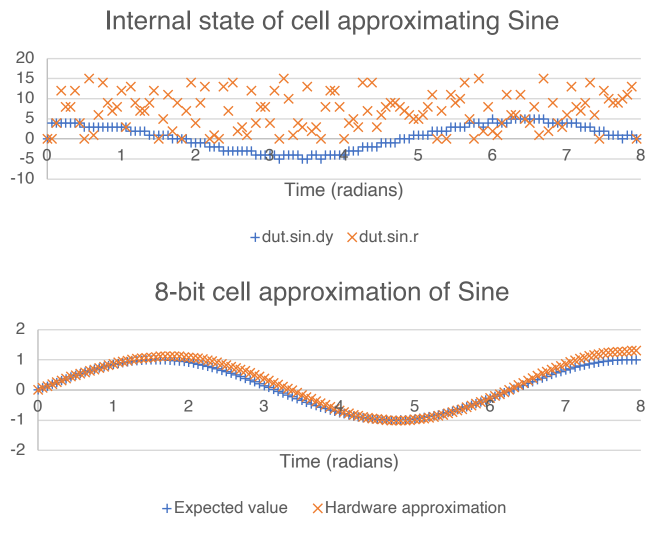

Fig. 2. The state and output of a cell-based approach for generating the sine function as a solution to an ODE. The output error relative to the expected value is due to a limited 8-bit number representation (roundoff error) and also a low-order integration method (truncation error). The error is asymptotically removed by improving both error sources.

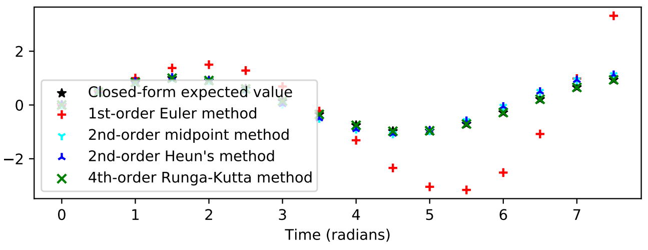

Fig. 3. Software and hardware solutions for generating the sine function, compared to the expected value. In the limited number of timesteps used to integrate across the time domain of 8 radians here, the lowest order Euler integration method has the greatest truncation error.

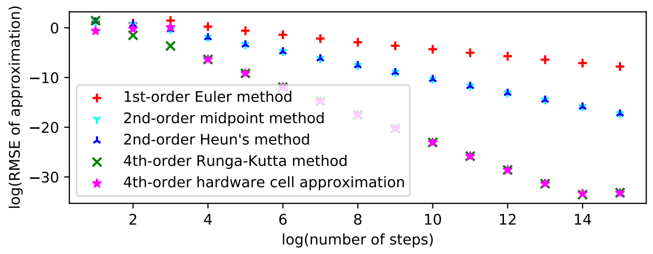

Fig. 4. Validation of up to fourth-order convergence for a hardware solution; the solution reaches the accuracy limited by the 64-bit number representation.

Finally, the iteration ends with a phase $i = s+1$:

$$\texttt{phase s+1: y <= y + } \sum_{j=1}^{s-1} b_j \star \texttt{dy}_j \texttt{ + } b_s \star \texttt{dy;}$$

The subscripted coefficients a and b in these rules come from the standard Butcher tableaus for Runga-Kutta methods [1].

Historically, digital differential analyzers routinely performed second- or third-order integration, though yet higher order integration was not done due to hardware complexity. With the above correct generalization (validated in Fig. 4), arbitrarily high order Runga-Kutta integration is now possible in modern FPGAs.

The key advantage of the hardware solution is that every numerical time step is completed in $s+1$ clock periods. The example in this section takes 2000 clock cycles amounting to 16μs on a commodity FPGA, roughly 10$\times$ faster than a compiled software solution. The next section will discuss how this advantage scales with larger problem sizes.

III. Demonstration of Spatial and Temporal Convergence for PDEs

In this section, we describe how a variety of standard numerical methods for scientific computing have clear mappings in the proposed configurable accelerator.

Partial differential equation (PDE) solver schemes typically begin with spatial discretization, after which the problem becomes solving a system of (generally nonlinear) ODEs. The order of convergence is also an important property for spatial discretization; here, it means an increase in spatial resolution leads to a polynomial decrease in error. In the configurable accelerator, a hardware cell would be instantiated for every spatial grid point. Higher-order spatial discretization schemes can be realized by different rules on how each cell's dz outputs get routed to neighboring cells' dy inputs. Finally, the temporal time stepping internal to each hardware cell representing a spatial grid point is done using high order Runga-Kutta schemes discussed in the previous tutorial section.

In the remainder of the section we demonstrate spatial and temporal convergence to the correct solution for a range of PDEs including a first-order advection equation, a second-order wave equation, and a nonlinear wave equation.

A. First-order advection PDE

The advection equation in one dimension has the form:

$$\frac{\partial y}{\partial t} = -\frac{\partial y}{\partial x}$$

First-order spatial discretization for this equation includes the upwind scheme, while second-order methods include the MacCormack method and the Lax-Wendroff method. Both second-order methods are realized using the proposed accelerator hardware by routing the dz and dy increments as follows:

For the MacCormack method, the register transfers for the cell located at spatial grid point k are:

$$\texttt{phase 1: dy}_k \texttt{ <= - (dz}_k \texttt{ - dz}_{k-1} \texttt{);}$$
$$\texttt{phase 2: dy}_k \texttt{ <= - (dz}_{k+1} \texttt{ - dz}_k \texttt{);}$$

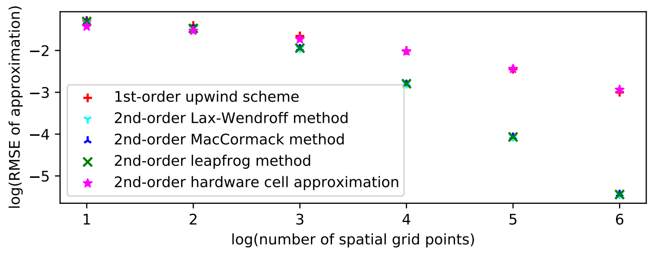

Fig. 5. Validation of second-order spatial convergence for a MacCormack hardware scheme for the one-dimensional advection PDE.

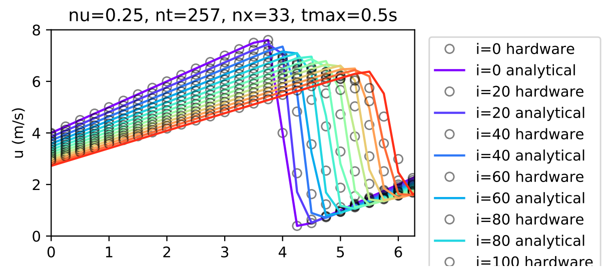

Fig. 6. Validation of correct solution for the nonlinear Burgers' PDE.

The above specifications of how neighboring cells update each other is paired with Heun's method second-order Runga-Kutta time stepping to complete the PDE numerical scheme.

For the Lax-Wendroff method, the register transfers are:

$$\texttt{phase 1: dy}_k \texttt{ <= dz}_{k+1}\texttt{/2 - dz}_k \texttt{ + dz}_{k-1}\texttt{/2;}$$
$$\texttt{phase 2: dy}_k \texttt{ <= - (dz}_{k+1} \texttt{ - dz}_{k-1} \texttt{) / 2;}$$

The above is paired with the midpoint method second-order Runga-Kutta time stepping to complete the scheme.

The convergence of the hardware solution to the correct solution with second-order accuracy in spatial resolution is validated in Fig. 5. This type of convergence on a high-resolution grid was never demonstrated in prior work on digital differential analyzers due to the high hardware cost before modern FPGAs, and recent work on PDE accelerators have not emphasized the importance of spatial convergence.

B. Second-order wave PDE

The wave equation is a second-order hyperbolic PDE. In one dimension it has the form:

$$\frac{\partial^2 y}{\partial t^2} = \frac{\partial^2 y}{\partial x^2}$$

A typical second-order discretization scheme for the wave equation is the finite-difference time-domain (FDTD) method, which has widespread use in electromagnetics simulation. In fact, if the scheme is defined on a single spatial grid point, the problem reduces to the sine-cosine pair given in the previous tutorial section. We implemented the FDTD scheme in the proposed hardware accelerator and observed the expected second-order in space and second-order in time rate of convergence.

C. Nonlinear Burgers' PDE

Finally, in order to demonstrate that nonlinear problems can be handled as well, we demonstrate solving the quasilinear hyperbolic Burgers' equation:

$$\frac{\partial y}{\partial t} = -y\frac{\partial y}{\partial x}$$

This equation is important in fluid dynamics as it forms the nonlinear core of the Navier-Stokes equations.

Following spatial discretization, the integration of the nonlinear y^2 term in the PDE is done using digital differential analyzer cells configured to be variable-variable multipliers [16]. An example hardware solution result is given in Fig. 6. The expected order of convergence is also validated against the purely software numerical scheme.

A hardware synthesis study shows that on the order of one thousand cells can be instantiated on a commodity FPGA. Yet larger PDE spatial grids can be accommodated in the hardware scheme by multiplexing the cells, using more clock cycles for each algorithm time step in favor of greater problem size.

IV. A HIGH-LEVEL INTERFACE TO THE ACCELERATOR

This section elaborates our pipeline from the user interface to the hardware cells. We chose the Julia package DifferentialEquations.jl as the user interface. Alternatives have not used accelerators or would be cumbersome for a scientist who is not already familiar with lower-level programming. Finally, we discuss the state of our pipeline (Fig. 7) and the challenges we foresee.

Julia, Python, and MATLAB are the primarily languages used by our envisioned users as each of these languages have standard interfaces to describe ODEs and PDEs. Among these, Julia is increasingly important: Julia's PDE solving ecosystem has coalesced around the package DifferentialEquations.jl [17]. The package exposes over fifty different PDE solvers, including GPU methods [18]. We can thus leverage existing accelerator work. In contrast, Python/SciPy does not prominently feature GPU acceleration for PDEs, and while MATLAB offers GPU arrays, they are not immediately geared for solving PDEs. Taken together, Julia is the best high-level language for us to target.

We have a successful proof of concept where numerical operations are already offloaded from the Julia language to an FPGA. Preliminary studies have been done on a ZYNQ-7010 development board; it has an FPGA and an ARM Cortex-A9 on the same die. The ARM core coordinates the configuration of the FPGA and the communications with the host computer running Julia. The actual arithmetic hardware reside on the FPGA, and AXI exposes registers to the C code on the ARM core.

Our end goal is a pipeline from DifferentialEquations.jl to an FPGA and back. We want to take the data that the package's interface expects, send it to the ARM core, process it using hardware cells, and communicate the result back. These efforts are toward a configurable accelerator for scientific computation that is accessible to scientists as the end users.

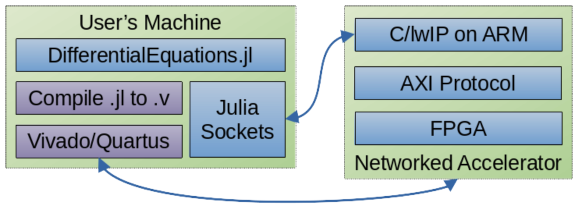

Fig. 7. An envisionsed pipeline from DifferentialEquations.jl to the FPGA.

REFERENCES

[1] Karsten Ahnert, Denis Demidov, and Mario Mulansky. Solving ordinary differential equations on GPUs. In *Numerical Computations with GPUs*, 2014.

[2] Tim Besard, Christophe Foket, and Bjorn De Sutter. Effective extensible programming: Unleashing Julia on gpus. *IEEE Transactions on Parallel and Distributed Systems*, 30(4):827–841, 2019.

[3] Wang Chen, Panos Kosmas, Miriam Leeser, and Carey Rappaport. An FPGA implementation of the two-dimensional finite-difference time-domain (FDTD) algorithm. In *Proceedings of the 2004 ACM/SIGDA 12th International Symposium on Field Programmable Gate Arrays*, FPGA '04, page 213–222. Association for Computing Machinery, 2004.

[4] 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.

[5] 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), apr 2013.

[6] 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), sep 2013.

[7] Jaeha Kung, Yun Long, Duckhwan Kim, and Saibal Mukhopadhyay. A programmable hardware accelerator for simulating dynamical systems. In *Proceedings of the 44th Annual International Symposium on Computer Architecture*, ISCA '17, page 403–415, 2017.

[8] Thomas Chen, Jacob Botimer, Teyuh Chou, and Zhengya Zhang. An SRAM-based accelerator for solving partial differential equations. In *2019 IEEE Custom Integrated Circuits Conference (CICC)*, pages 1–4, 2019.

[9] Thomas Chen, Jacob Botimer, Teyuh Chou, and Zhengya Zhang. A 1.87-mm² 56.9-GOPS accelerator for solving partial differential equations. *IEEE Journal of Solid-State Circuits*, PP:1–10, 01 2020.

[10] Gagandeep Singh, Dionysios Diamantopoulos, Christoph Hagleitner, Juan Gomez-Luna, Sander Stuijk, Onur Mutlu, and Henk Corporaal. NERO: A near high-bandwidth memory stencil accelerator for weather prediction modeling. In *2020 30th International Conference on Field-Programmable Logic and Applications (FPL)*, pages 9–17, 2020.

[11] Junjie Mu and Bongjin Kim. A dynamic-precision bit-serial computing hardware accelerator for solving partial differential equations using finite difference method. *IEEE Journal of Solid-State Circuits*, 58(2):543–553, 2023.

[12] Jiajun Li, Yuxuan Zhang, Hao Zheng, and Ke Wang. FDMAX: An elastic accelerator architecture for solving partial differential equations. In *Proceedings of the 50th Annual International Symposium on Computer Architecture*, ISCA '23, 2023.

[13] Robert B. McGhee and Ragnar N. Nilsen. The extended resolution digital differential analyzer: A new computing structure for solving differential equations. *IEEE Transactions on Computers*, C-19(1):1–9, 1970.

[14] Paul William Baker. *Algorithms for Higher Level Functions in Machine Hardware.* PhD thesis, AUS, 1976. AAI0596601.

[15] Philip George Mccrea. *Raster Scan Computer Graphics and Incremental Computing Systems.* PhD thesis, AUS, 1976. AAI0596564.

[16] Bernd Ulmann. *Analog Computing*. De Gruyter Oldenbourg, Berlin, Boston, 2022.

[17] Christopher Rackauckas and Qing Nie. DifferentialEquations.jl–a performant and feature-rich ecosystem for solving differential equations in Julia. *Journal of Open Research Software*, 5(1), 2017.

[18] Utkarsh Utkarsh, Valentin Churavy, Yingbo Ma, Tim Besard, Tim Gymnich, Adam R Gerlach, Alan Edelman, and Christopher Rackauckas. Automated translation and accelerated solving of differential equations on multiple GPU platforms. *arXiv preprint arXiv:2304.06835*, 2023.