nature communications

Article

https://doi.org/10.1038/s41467-025-68122-4

High-concurrency tri-mode memristor-based ordinary differential equation solver

Received: 3 June 2025

Accepted: 18 December 2025

Published online: 08 January 2026

Check for updates

Lianfeng Yu[1], Teng Zhang[1], Yang Han[1], Bowen Wang[1], Ziang Xie[1,2], Haochen Zhang[1], Jiaxin Liu[1], Longhao Yan[1], Pek Jun Tiw[1], Daijing Shi[1], Lei Cai[1], Yaoyu Tao[1,3,4] & Yuchao Yang[1,2,3,5]

Ordinary differential equations (ODEs) are widely used in science, engineering, and mathematics, but their numerical solution on traditional Von Neumann hardware is time- and energy-consuming, especially for high-order ODEs. Here, we present a high-concurrency memristor-based ODE solver supporting arbitrary order and three configurable modes: coarse, fine, and coarse-to-fine look-ahead, to meet diverse accuracy requirements. History-based memristor programming (HMP) accelerates device conductance programming by up to 3.29 × without compromising accuracy. The reconfigurable hardware implements coarse solver via analog compute-in-memory, fine solver via digital compute-in-memory, and coarse-to-fine solver using Parareal methods for high-concurrency numerical integration. We demonstrate its performance on exponential functions, Lorenz attractors, and three-body problems, achieving 601 × ~ 6.92 × 10^3 × speedup and 1.71 × 10^3 × ~ 3.93 × 10^3 × energy improvement over CPU/GPU, respectively, when solving the same ODE tasks. The memristor-based tri-mode solver pushes ODE solver hardware performance to a new paradigm with orders of magnitude concurrency improvements.

Ordinary differential equations (ODE) are fundamental mathematical theories in a wide range of science and engineering fields, including physics[1–4], biology[5–7], economics[8–10], or control systems[11–13]. The ability to solve ODEs efficiently is critical for real-time simulations, optimization, and decision-making processes. With increasing complexity of modern ODE applications, such as artificial intelligence, climate modeling, and autonomous systems, traditional ODE solvers based on general purpose processors (GPUs/CPUs) or FPGAs/ASICs often face limitations in terms of speed, energy efficiency, and scalability. This is because these hardware solvers are inherently limited by the Von Neumann architecture, which suffers from data transfer bottlenecks between processing units and memory. Additionally, iterative numerical methods for solving ODEs, such as Runge-Kutta (RK) or Euler methods, can be very computationally expensive, especially for stiff equations or high-dimensional systems. These limitations become particularly pronounced in real-time applications, where delays in computation can lead to suboptimal or even catastrophic outcomes.

Hardware ODE solvers have been extensively researched in the past several decades. CPU/GPU-based ODE solvers often employ instruction-level parallelism or architecture-level optimization[14–16], but are still limited by memory bandwidth when it comes to stiff equations and high-dimensional problems. Many researchers also use FPGAs[17–21] for hardware acceleration of the ODE solving process, but these prior arts also suffer from Von Neumann bottlenecks. Furthermore, most of them are fixed data flow architectures designed for specific equations, lacking necessary reconfigurability. The latest hardware for the ODE solver[22,23] on FPGAs/ASICs supporting the fourth-order Runge-Kutta method records a power consumption of 367mW. In order to resolve

[1]New Cornerstone Science Laboratory, Beijing Advanced Innovation Center for Integrated Circuits, School of Integrated Circuits, Peking University, Beijing, China. [2]New Cornerstone Science Laboratory, Guangdong Provincial Key Laboratory of In-Memory Computing Chips, School of Electronic and Computer Engineering, Shenzhen Graduate School, Peking University, Shenzhen, China. [3]Center for Artificial Intelligence Chips, Institute of Artificial Intelligence, Peking University, Beijing, China. [4]PKU-Wuhan Institute for Artificial Intelligence, Peking University, Wuhan, China. [5]Center for Brain Inspired Intelligence, Chinese Institute for Brain Research (CIBR), Beijing, China. e-mail: taoyaoyutyy@pku.edu.cn; yuchaoyang@pku.edu.cn

the bottlenecks of traditional Von Neumann architectures, computing-in-memory (CIM) architectures are progressing rapidly, where emerging memristors can enable computations within crossbar arrays to accelerate matrix-vector multiplication or sorting operations[24–32]. Recent work on CIM architectures has focused on hardware designs for partial differential equations (PDE) using Flash[33,34] or memristor[35,36] devices. However, these PDE hardware solvers are mainly designed for specific PDEs, and changes in PDEs require reprogramming of the devices, which is a big overhead for memristors. Also, the classical Jacobi methods used for PDE hardware solvers are difficult to apply directly for ODE solving, as they can only be used with a fixed step size in memristor-based hardware, which introduces lower accuracy and a larger array. Thus, there is a lack of a CIM-based ODE hardware implementation solution.

In this work, we develop an end-to-end memristor CIM-based ODE solver hardware supporting arbitrary ODE solver orders. The design features two distinct modes with accuracy and speed trade-offs. We build our ODE solver hardware with device-level investigations up to system-level optimizations. We fabricate 1-transistor-1-resistor (1T1R) memristor arrays using a 180nm technology and map RK ODE solver coefficients of any order onto them. In order to accurately program these coefficients, we propose a history-based memristor programming (HMP) scheme to speed up conductance programming by up to 3.29 × while maintaining excellent accuracy. A reconfigurable hardware architecture that enables the coarse solver to use analog compute-in-memory, the fine solver to use digital compute-in-memory, and the coarse-to-find look-ahead to use parallel methods is designed for high-concurrency numerical integration. We demonstrate the 2nd and 6th order RK ODE solvers with memristors that can be configured as coarse solver for exponential functions, the fine solver for Lorenz-attractor problems, and the high-concurrency coarse-to-find solver for classical 3-body problems. Experimental results on these problems show that our ODE solver hardware can deliver up to 601 × ~ 6.92 × 10^3 × speedup and 1.71 × 10^3 × ~ 3.93 × 10^3 × energy improvements compared to general-purpose CPU/GPU, respectively, when solving the same ODE tasks. The memristor-based coarse-to-fine look-ahead scheme pushes ODE solver hardware performance to a new paradigm with orders of magnitude concurrency improvements.

Results

Memristor-based tri-mode ODE solver

ODEs are ubiquitous in numerous applications, as shown in Fig. 1a. They are usually difficult to solve with analytical expressions but can be solved numerically. However, numerical methods are often iterative and very time-consuming while demanding large amounts of computational resources. The Runge-Kutta (RK) methods are a classical family of explicit and implicit iterative numerical methods[37] (see "Methods" for details) that are widely used in numerous applications and are also standard ODE solvers in commonly used software tools such as Matlab or Python. They require a large number of iterative matrix and vector computations and also involve storage and transmission of intermediate results, which are both time- and energy- consuming when running on traditional Von Neumann architecture. We build our memristor-based ODE solver hardware based on Runge-Kutta method, which can be reconfigured as coarse, fine, or course-to-fine look-ahead mode with arbitrary ODE order to address different application requirements: (1) For applications with low accuracy requirements but high demands on speed, we configure the hardware as a coarse solver, utilizing an analog memristor conductance and an in-memory stepsize computation method, which can perform complex computations in one step with only a small overhead; (2) For very high-accuracy applications, we configure the hardware as a fine solver with binary memristor conductance that can minimize accuracy loss due to memristor non-idealities; (3) For applications with mid-to-high accuracy and speed requirements, we configure the hardware as a coarse-to-fine look-ahead mode that utilizes Parareal methods with hybrid fine and coarse solvers as shown in Fig. 1b.

We fabricate 32 × 36 1T1R (1-transistor-1-resistor) memristor arrays using 180nm technology (Fig. 2a) to build the ODE solver system. We use TiN/TaOx/HfO2/TiN memristors, and the transmission electron microscope (TEM) image is shown in Fig. 2b. By applying pulses with different conditions, the devices can be fully distinguished into two different conductance states corresponding to digital 0's and 1's. We program all devices in the six arrays using a single pulse ($2V$ $200ns$ for SET operation, $3.5V$ $500ns$ for RESET operation and $0.2V$ for read) without verify, and each device is successively programmed to the high and low conductance states, and the results are shown in Fig. 2c. Our devices can also be programmed to analog conductance states. Typically, the analog conductance programming of memristors needs to be achieved by iterative write-verify operations, which is time-consuming. Conventional programming schemes write the device conductance to the target value by iterative trial and error. However, the traditional process does not consider the conductance mismatch distance from each trial iteration, which wastes a lot of time before approaching the target conductance. Some other programming schemes have been proposed, such as COPS[38], but they are mainly designed to address the relaxation of the memristor device and are not optimized for programming speed.

Here, we propose history-based memristor programming (HMP), a write-verify method that can learn from the conductance mismatch distance of past programming trials, leading to a much faster convergence speed. The HMP classifies all possible situations encountered during the conventional write-verify process into different states (s_1, s_2, s_3), and stores the actions taken a after encountering a certain state in a state space (Fig. 2d). We use a set of reward and penalty rules to refine and update the actions stored in the state space based on the conductance of the device after taking that action. The state space gradually converges by repeatedly taking and updating actions. The state space includes the states and their corresponding actions, and each state contains three state variables, i.e., the current conductance of the device, the absolute value of the difference between the current and the target conductance, and the current intensity of the applied pulse. Note that the number of state variables can be set flexibly, and the above descriptions are the setup in our work for testing and demonstration. The actions represent the intensity of the pulses that will be applied when the next trial returns to this state. The HMP flow chart is shown in Fig. 2e. For each device, we set a target state (G_t) as well as a tolerable write error (err). The device is first read, and if the target is not reached, we search the current state of the device (s) in the state space. If this is not the first iteration, we need to calculate the updated action based on the read-out information through a dedicatedly designed reward and penalty mechanisms (detailed in Supplementary Note 2) and use them to update the actions corresponding to the state of the previous iteration. We then determine the action that needs to be taken based on the current state and applied to the memristor device. The above process is repeated until the write is successful or the iteration limit is reached.

With the proposed HMP scheme, we can learn from the write-verify process of each device and take the actions that best match the characteristics of the device in the current state, which greatly speeds up the writing process. To illustrate the performance of HMP, we iteratively program the device to 32 different conductance states using HMP and the conventional write-verify method, respectively, and record the average programming latency. We tested a total of 100 devices, and the average programming results are shown in Fig. 2f. From the results, we can see that with the increase of the cycle, the

https://doi.org/10.1038/s41467-025-68122-4

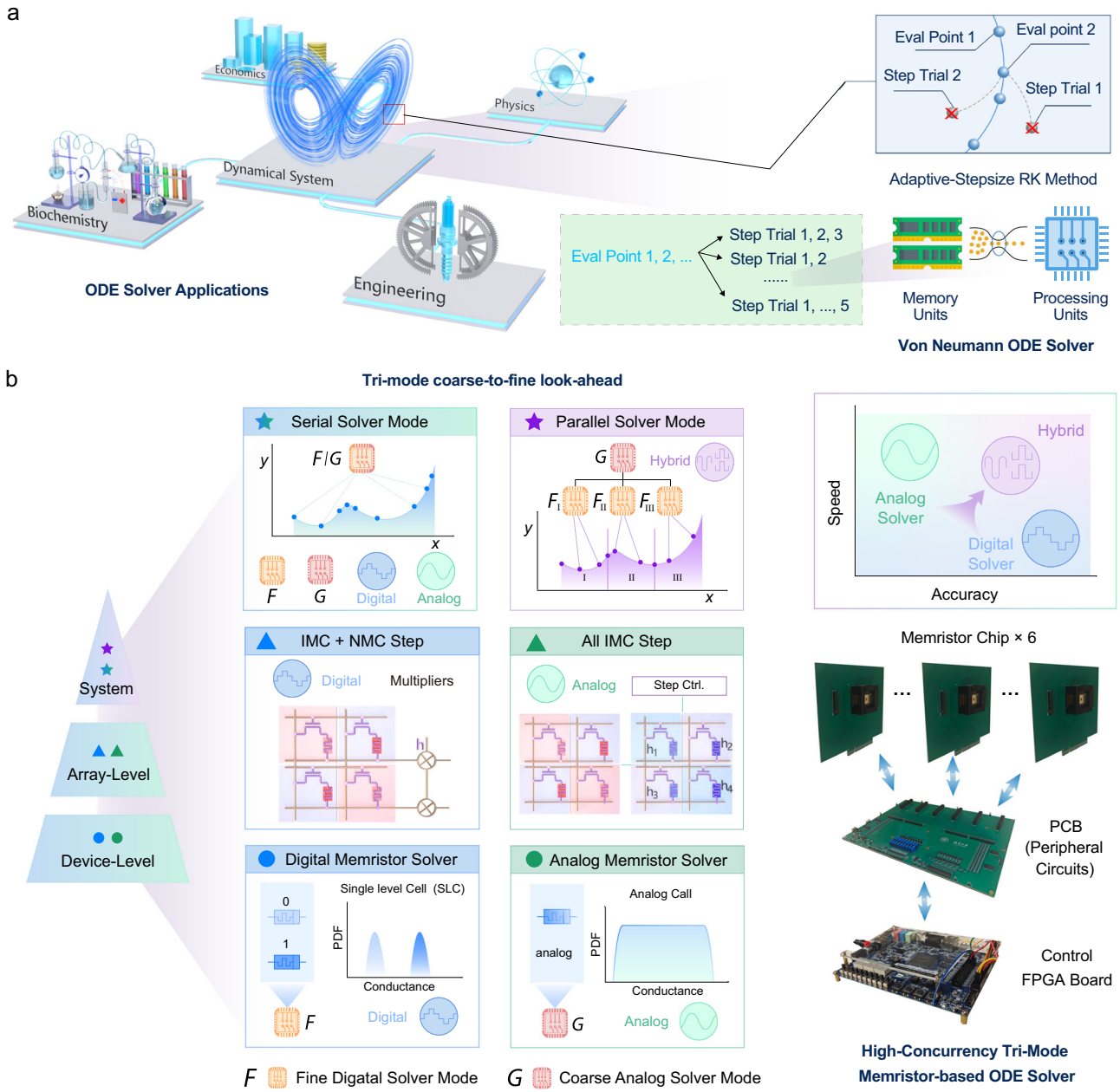

Fig. 1 | Overview of ODE solving system. a ODEs are ubiquitous in various applications, and they are typically solved numerically by the classical adaptive-stepsize Runge-Kutta method. **b** Overview of proposed high-concurrency memristor-based ODE solver with tri-mode coarse-to-fine look-ahead.

state space gradually converges, the writing speed is also gradually accelerated, after 60 cycles, the speedup stabilizes and can reach 3.29 × of the conventional write-verify method. Our devices can be programmed to any conductance state easily by HMP, and we choose 32 typical conductance states (3 μS to 52.6 μS, with a write error of ± 0.8 μS) to plot the cumulative distribution function (CDF) as shown in Fig. 2g. We conduct a total of 10000 programming experiments for each conductance state on 100 devices, and 99.72% of the programming is completed within the specified cycle (20 cycles in our experiments). It shows that HMP can program devices at a faster speed while maintaining a very high success rate, while demonstrates the analog conductance programming potential of memristors.

The adaptive stepsize is one of the crucial factors for an ODE solver because a fixed stepsize that is too large may lead to low accuracy, while one that is too small may lead to too much computational overhead. Optimal ODE solver performance can only be achieved if the stepsize can be adaptively adjusted according to the error in the numerical process. However, existing implementations for adaptive stepsize heavily rely on peripheral circuits, which is not efficient. Previous work on solving PDEs with memristors uses a fixed step size[35], which cannot be adjusted during run-time. We propose an in-memory strategy to compute the stepsize and can accomplish the computations in one step within the memristor array without repeated programming for applications with low-accuracy requirements.

The explicit Runge-Kutta method is a set of serial algorithms, which are widely used due to their simple calculations. In contrast, implicit methods have better stability and fewer parameters, but typically require solving a system of equations, which is more complex, and currently lack suitable hardware solutions. We choose to use fixed-point iteration to solve the implicit method, which can transform the solution of the system of equations into continuous iterative matrix vector multiplication (MVM) to fully leverage the parallelism of CIM. The main formulas

https://doi.org/10.1038/s41467-025-68122-4

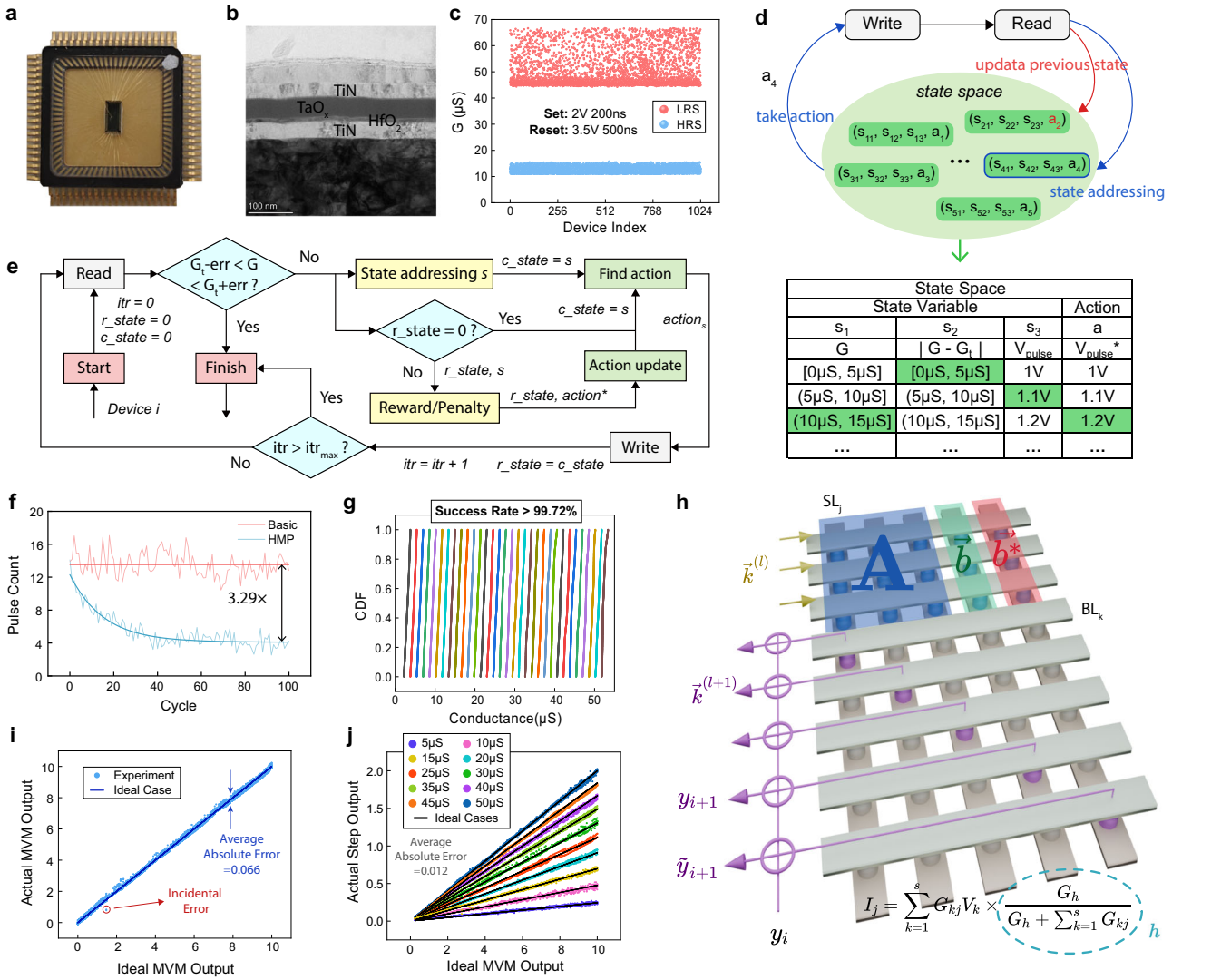

Fig. 2 | Memristor programming and in-memory step computing. a 1T1R memristor chip with 180nm technology. **b** TEM image of our memristor. **c** Digital conductance state distribution by applying a single pulse to program all the devices in six arrays. **d** Schematics of HMP (history-based memristor programming) and state space specifics (*s* represents the state variables, *a* represents the action taken). **e** Detailed flow chart of HMP, where G_t represents the programming target conductance. **f** Average number of programming pulses required per cycle to repeatedly program 100 devices to 32 conductance states using HMP and basic write-verify method. **g** The CDF (cumulative distribution function) of 32 conductance states programming by HMP. **h** In-memory step computing. **i** Normalized ideal and experimental MVM output. **j** Normalized ideal and experimental step size computing results with different G_h.

are as follows (see "Methods" for details):

$$\begin{aligned} &\mathbf{k}^{(0)} = \mathbf{0} \\ &\mathbf{k}^{(l)} = f(x_i + h_i{}^*\mathbf{c}, y_i + h_i{}^*\mathbf{k}^{(l-1)}\mathbf{A}),\ l = 1, \ldots, \sigma \\ &x_{i+1} = x_i + h_i \\ &y_{i+1} = y_i + h_i{}^*\mathbf{b} \cdot \mathbf{k}^{(\sigma)} \end{aligned} \tag{1}$$

where y is an unknown function of time x, which we would like to numerically solve. The initial conditions and the function $f(x, y)$ are known quantities, where $f(x, y)$ is the derivative of y with respect to x. Located in step y_i, we need to solve the coefficient vector **k** first and then obtain the integration result y_{i+1} for the next step. The above step-by-step process needs to be repeated until we reach the final point x_n. The Runge-Kutta matrix **A**, the weight **b**, and the node **c** are related to the order of the Runge-Kutta methods and do not need to be changed once the method is determined. This mapping scheme, based on the method we used, avoids the need for reprogramming that arises when traditional memristor-based PDE solvers encounter equation changes. Thus, implicit methods have been transformed into hardware-friendly iterative MVM (**kA**). Mapping the vector **k** to the input voltages on bitlines (BL) and the memristor array storing the matrix elements **A** and weight **b**, the cumulative currents on the source lines (SL) can be written as **kA** and **b** · **k** according to Kirchhoff's law, which is the conventional way to perform MVM. To support stepsize calculations, we store the mapped steps in the memristor array below the weight array and read the current output from the BL of the step element instead of the SL, as shown in Fig. 2h.

Let's take the jth column of the array as an example, assuming that the input vector has a length s. The conductance stored in the kth row of the first s rows of the array is G_{kj}, and in row $s + j$ is G_h, according to Kirchhoff's law, the output current on the $(s+j)^{th}$ BL can be written as:

$$I_j = \sum_{k=1}^{s} G_{kj} V_k \times \frac{G_h}{G_h + \sum_{k=1}^{s} G_{kj}} = \left(\sum_{k=1}^{s} G_{kj} V_k \right) \times h \tag{2}$$

https://doi.org/10.1038/s41467-025-68122-4

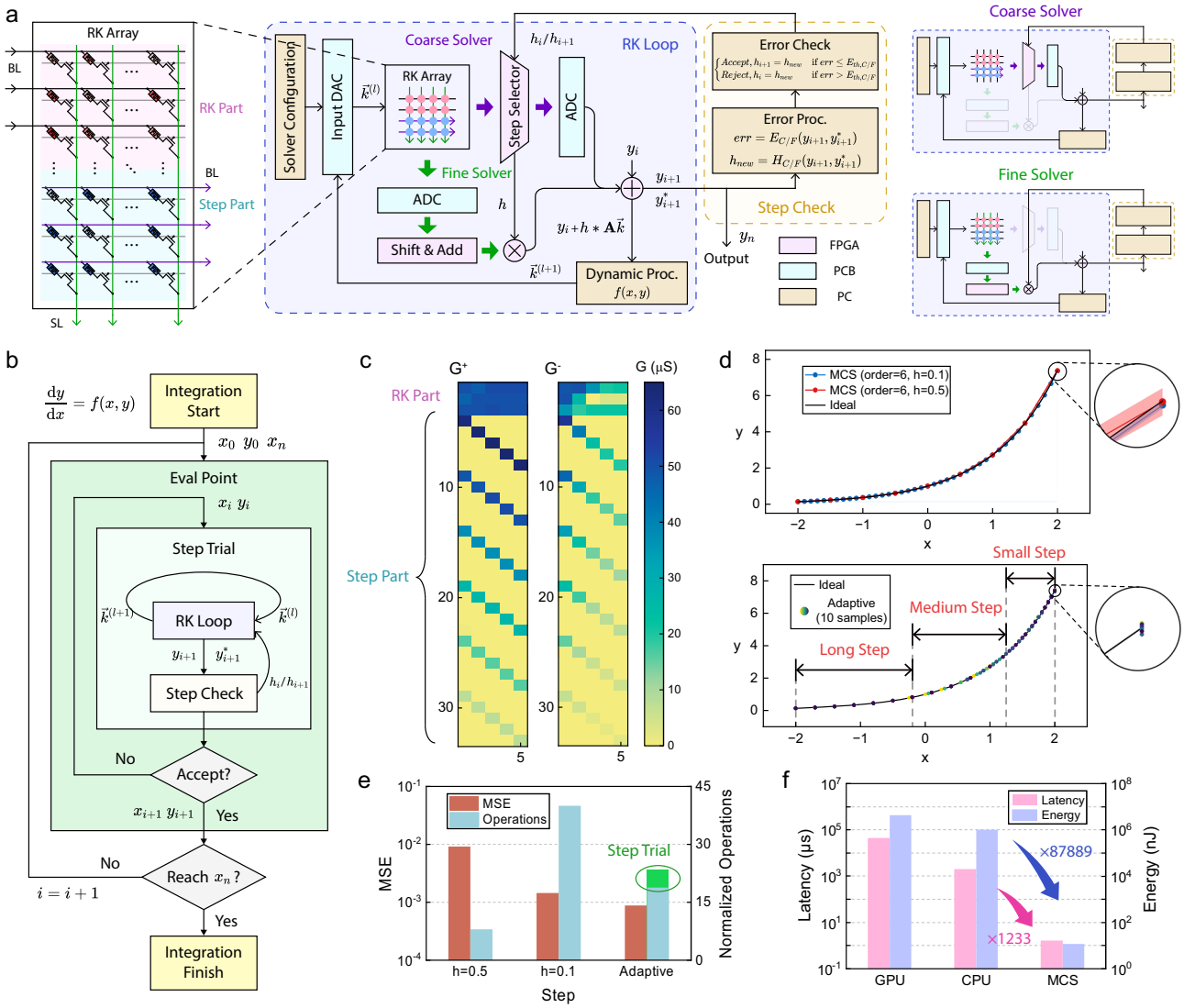

Fig. 3 | System architecture and coarse solver for solving the exponential function. a Our memristor-based ODE solving architecture, which can be reconfigured into coarse solver mode and fine solver mode. **b** System flowchart for using fixed-point iteration to solve Runge-Kutta methods. **c** Array mapping diagram of the implicit Gauss-Legendre method of order 6. We use 6 different step sizes to ensure efficient solving. **d** The result of solving the exponential function ten times using fixed step sizes of 0.1 and 0.5 (with error bars representing the standard deviation), and our developed adaptive stepsize. **e** Comparison of MSE (mean squared error) and normalized operations solved with three different step sizes. **f** Comparison of latency and energy consumption of CPU/GPU and MCS at the same accuracy.

Notice that the first part is exactly the results of the MVM, and the second part can just be used as the step size. We store many such step parts in the memristor array and develop a corresponding adaptive stepsize algorithm. The error weights $\mathbf{b}^*$ are stored in the memristor array, and a step selector is designed to control which step to use (details in Supplementary Note 5). In order to realize parallel step calculations, for the same step part, we store the step conductance at the diagonal of the array and use memristor devices without forming at other locations, so that the conductance at these locations can be ignored during the in-memory calculations. Our designed hardware architecture is shown in Fig. 2h. We can implement MVM and step multiplications in one step by in-memory computing, and we are able to implement adaptive stepsize control inside the chip.

Our computational arrays can be reconfigured into two modes; one mode is normal MVM for the fine solver with binary conductance, and the other is in-memory step computations for the coarse solver with analog conductance. In order to study the performance of our memristor array, we collect MVM data from the memristor array with different inputs and weights, and the normalized data are shown in Fig. 2i. The experimental data and the ideal results show a very high degree of match, with a mean absolute error of 0.066. This confirms the reliability of our memristor calculations, both with binary and analog conductance. We also choose different stepsize conductance (G_h) and test the results of in-memory step computing (Fig. 2j). We selected a total of ten different G_h while keeping $\sum_{i=1}^{n} G_{ij}$ consistent and measured the mean absolute error of 0.012, demonstrating the feasibility of our proposed one-step stepsize calculation within memory.

We build a memristor-based ODE solver hardware that can be configured to implement coarse, fine, or coarse-to-fine look-ahead solvers (Fig. 3a), and the flow chart is shown in Fig. 3b. The design can solve both implicit and explicit Runge-Kutta methods by fixed-point iterations (see "Methods" for details). As mentioned in the previous section, we decompose the array into an RK part and a step part. Reading data from the BLs of the step part configures the hardware as a coarse solver mode. In this mode, we program the devices into analog

https://doi.org/10.1038/s41467-025-68122-4

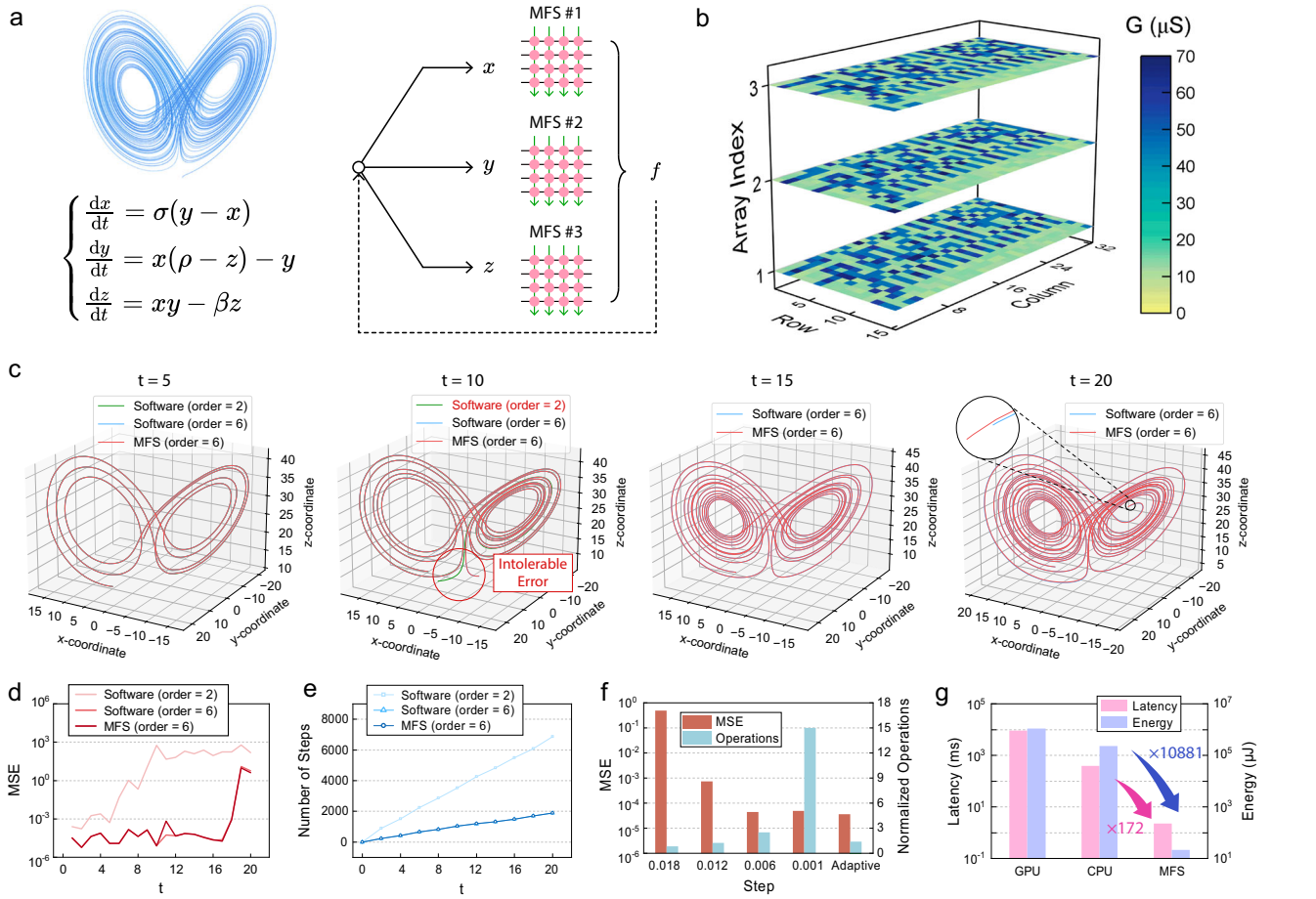

Fig. 4 | Fine solver for solving Lorenz-attractor problem. a Lorenz system containing three first-order ODEs, we use MFS (memristor-based fine solver) to solve for each of the three dependent variables (x, y, z). **b** Arrays mapping of MFS, with all weights quantized to 32-bit. **c** The solution results of software coarse/fine solvers and MFS at different integration endpoints. **d** Comparison of MSE among three different solvers. **e** Comparison of normalized operations among three different solvers. **f** The MSE and normalized operations for the different step size cases of MFS when solving at $t = 15$. **g** Comparison of latency and energy consumption of CPU/GPU and MFS achieving the same accuracy.

conductance using HMP and enable one-step in-memory implementation for stepsize computation to get better efficiency. We store the error weights b^* in the RK part, which can be used as a basis to adjust the stepsize. Furthermore, analog operations introduce noise during memristor programming and reading, and the calculation results may not be very accurate. Therefore, this coarse solver mode is only suitable for application scenarios with high energy efficiency demands but low accuracy requirements.

On the other hand, if the step part is skipped and we read directly from the SLs, the hardware can be configured as a fine solver mode. The input vector **k** and the Runge-Kutta parameters are both quantized to 32 bits for high-accuracy purposes. Binary conductance devices are easier to program with fewer bit errors, and the length of k is small in most widely used methods, so we can process directly within the array using current accumulation without introducing many errors. The fine solver stepsize is implemented using an external multiplier, and we adjust the stepsize using a mathematical estimation of the local truncation error of a single Runge-Kutta step[39]. This is done by having two function estimations, one of order p and the other of order $p-1$. These two estimations share the same **A** and **c**, only with different **b**. Therefore, we simply change the error weights b^* to the weights of order $p-1$ in our system (detailed in Supplementary Note 5). Due to digitalized input bits, the fine solver requires multi-cycle inputs, and the peripheral circuits and step control are more complex; therefore, it is only suitable for application scenarios with high accuracy requirements, but can tolerate some speed degradations compared to the coarse solver.

The iterative computing flow of the ODE solver hardware is controlled by the FPGA. Firstly, for step i, $\mathbf{k}^{(l)}$ is assigned to the input voltage on the BLs of the RK part via digital-to-analog converters (DACs). For the coarse solver mode, we use a step selector to control the step part to select a step (h_i). The results are read by analog-to-digital converters (ADCs) and then sent to a processor to calculate the function $f(x, y)$. The fine solver mode does not enable the step part, and the readout current of the SLs is fed to the processor after passing through the ADCs, peripheral shift adders, and multipliers. The above process completes one iteration in the RK methods. The implicit RK methods require p_0 iterations, and the number of iterations for the explicit RK methods is the length of **k**. Once the maximum number of iterations has been reached, we iterate again to obtain y_{i+1} and y^*_{i+1}, which are sent into the step check module to derive the next step. The step check module determines whether to accept or reject the current step and generates the next step size to be used (detailed in Supplementary Note 5). The process repeats until we reach the end (x_n). Our system consists of 1T1R chips, PCB, FPGA, and PC. The operations performed by each component are labeled in different colors in Fig. 3a.

Experimental results using coarse and fine solvers for dynamical systems

We implement the memristor-based tri-mode ODE solver hardware with six 32 × 36 memristor chips on the PCB with the help of control FPGAs, as shown in Fig. 1b. We then configure the hardware as coarse, fine, or coarse-to-fine look-ahead mode and carry out experiments on representative ODE solving problems, including exponential functions, the Lorenz-attractor problem, and the classical three-body problems. For the coarse mode, we use two arrays with analog conductance devices to represent positive and negative weights, respectively. For the fine mode, each dimension requires an array for calculation, and there are three dimensions in our experiment (Lorenz attractor). We quantify all the Runge-Kutta coefficients into 32-bit and map them to three arrays with a binary device. And for the coarse-to-fine look-ahead mode, two arrays are set to coarse mode, and four arrays are set to fine mode. Detailed information on hardware implementation and computing pipelines is provided in Supplementary Note 11.

We first experiment our coarse solver to solve exponential functions, which is one of the most important basic functions denoted by $f(x) = a^x$. a is generally Euler's number e. Here, we show how our memristor-based coarse solver (MCS) can easily solve arbitrary exponential functions. For an ordinary differential equation $\frac{dy}{dx} = f(x, y)$: If we make $f(x, y) = ky$, then the solution can be expressed as $y = e^{kx}$. At this point, the function $f(x, y)$ involves only the simplest multiplication operations on y, which can be easily performed. Note that there are intrinsic errors associated with memristors of analog conductance, so the MCS results are slightly off compared to the software results for methods of the same order. In the experiment, our MCS uses the implicit RK method of order 6 (i.e., Gauss-Legendre methods) and sets 6 alternative step sizes (more detailed experiments of different orders are provided in Supplementary Note 6). We use HMP to program our devices to the desired analog conductance states, and the results are shown in Fig. 3c. We use a differential mapping scheme to improve the computational accuracy, as shown in Supplementary Note 4. The length k of the Gauss-Legendre method is only 3, so we can use more memristor arrays for adaptive stepsize to improve accuracy. In contrast, explicit RK methods of the same or lower order have a larger k. For example, in the most widely used explicit RK method of order 5 (i.e., ode45 used in Matlab), the length of k is 7, which implies a larger overheads in memristors. The maximum stepsize we chose is 3 times the minimum stepsize.

In the experiments, we set the initial condition as $\frac{dy}{dx} = y$, $y(-2) = e^{-2}$, and the target to be solved is $y(2)$. With both fixed stepsize and adaptive stepsize, we solve the problem ten times using MCS of each stepsize choice, and the results are shown in Fig. 3d. Regardless of which fixed stepsize is used, the results of MCS are almost identical to the ideal exponential function, and, in addition, the distribution interval of the solution becomes significantly smaller as the stepsize becomes smaller. For adaptive stepsize experiments, we set the minimum stepsize to be 0.05 and the maximum stepsize to be 0.3. It can be found that when the slope of the function is relatively small, i.e., x is small, the step size controller selects the bigger step size. On the other hand, when x increases, the slope also grows, and the stepsize choices will gradually decrease to control the truncation error within an error tolerance.

We measure the mean squared error (MSE) of the solutions as well as the normalized number of operations (defined as the number of steps taken to solve the problem, which is proportional to the number of MVM operations in the memristor array) computed for three different stepsizes, as shown in Fig. 3e. Using a large stepsize (h = 0.5) results in the least number of operations, but the MSE is also the largest. A smaller stepsize (h = 0.1) reduces the MSE, but the number of operations also grows. The adaptive stepsize that we developed achieves an MSE that is better than that with stepsize h = 0.1, while requiring fewer operations. We compare the speed and power consumption required in this task using CPUs/GPUs and MCS, achieving the same precision as shown in Fig. 3f. From the experimental results, it can be seen that MCS can achieve 1233 × speedup and 87889 × reduction in energy consumption, while the solution achieves the same accuracy. The high parallelism of the memristor-based analog computing allows us to implement higher-order ODE solving methods at a much smaller cost. We also analyze the performance of MCS in more detail in Supplementary Note 6.

The MCS based on analog CIM achieves very high hardware efficiency but may also introduce unavoidable accuracy loss in some high-precision applications. To illustrate the problem, we first apply MCS to solve the Lorenz-attractor problem. It is famous for its chaotic solutions for certain parameter values and initial conditions, while the Lorenz attractor is one of such a set of chaotic solutions (Fig. 4a). The "butterfly effect", by which small changes in initial conditions or parameter values result in completely different trajectories, significantly impacts the Lorenz attractors in real-world applications. This implies that this kind of ODE is very sensitive to numerical computational accuracy, and any small accuracy loss can lead to a huge difference in the final results. Detailed information on the Lorenz attractor system and its solving steps is provided in Supplementary Note 3.

We use the implicit Gauss-Legendre method of order 6 to experiment with the Lorenz-attractor problem on our memristor-based fine solver (MFS), and the mapping results of the arrays are shown in Fig. 4b. We set the initial condition to $t_0 = 0$, $x_0 = 5$, $y_0 = 10$, $z_0 = 10$, and solve the results in different t, as shown in Fig. 4c. The results of the MFS solution are compared with the software simulations, and the three results coincide when $t = 5$. However, when t increases to 10, the coarse solver causes an intolerable error and evolves to a completely different trajectory. The MFS shows good performance until $t = 20$, when it begins to differ significantly from the software results. Note that the error is proved to be caused by the quantization error. This is because we quantify the Runge-Kutta coefficients as 32-bit, and if we quantify them into 64-bit numbers, MFS can always sustain the same level of accuracy as the software results (Supplementary Note 7). We measure MSE and the number of steps forward for different methods, as shown in Fig. 4d, e. It is worth noting that the coarse solver has more steps instead, and this is because the step size of the coarse solver needs to be taken very small in order to achieve the error required for step control. However, because of the low ODE solver order of the coarse solver in this experiment, even small stepsize cannot guarantee high accuracy. In high-accuracy application scenarios, we prefer to use higher-order ODE solver methods. We measure MSE and MFS operation counts for different stepsizes when $t = 15$ (Fig. 4f). It can be seen that the Lorenz attractor is very sensitive to the stepsize, and larger stepsizes lead to dispersion of the results. The adaptive stepsize method can achieve the desired accuracy with the smallest number of operations. We also compare the speed and energy consumption required to perform this task using CPUs/GPUs and MFS (with order 6) that achieve the same accuracy as shown in Fig. 4g. MFS achieves 172 × acceleration and 10881 × energy reduction compared to the CPU.

Design and experiments of the coarse-to-fine look-ahead solver

In some practical applications, there is often a high demand for accuracy, which cannot be met by a coarse solver, while using a fine solver takes too long a computation time. Therefore, we propose a memristor-based coarse-to-fine look-ahead parallel solver (MPS) based on the Parareal method[40,41], which can achieve higher speed while guaranteeing high accuracy. MPS is able to fully utilize the computing resources on the chips to achieve better performance. The main idea can be summarized as follows: we first predict a rough solution using a high-speed coarse solver and then correct the results using a high-

https://doi.org/10.1038/s41467-025-68122-4

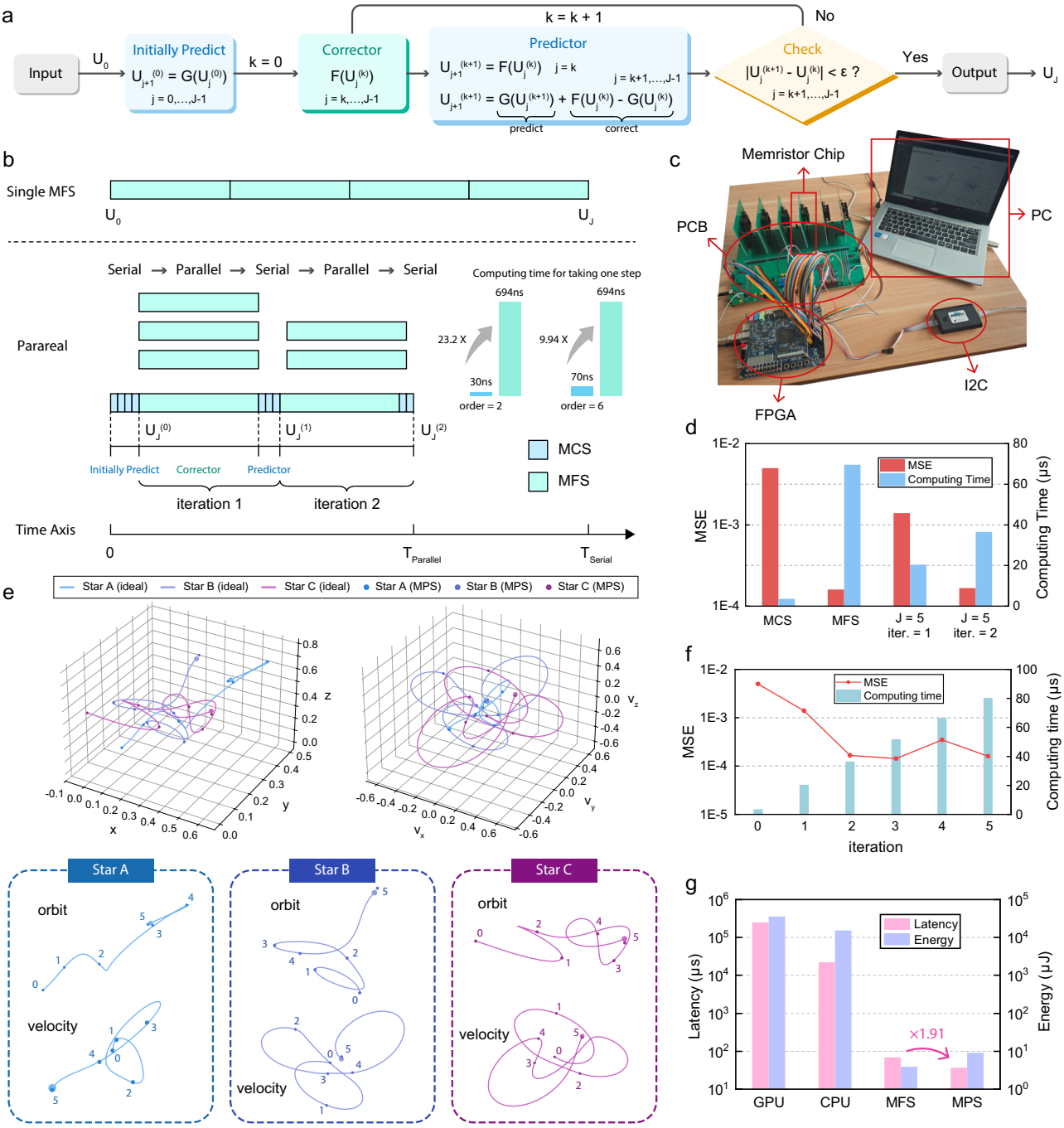

Fig. 5 | Parallel solver based on Parareal for solving three-body problem. a Flow chart of parallel solving using Parareal. The entire integration interval is divided into J parts, and the initial value of each interval is denoted by U_j ($j = 0, \ldots, J-1$). The symbol $G(U_j)/F(U_j)$ denotes the solving using a coarse or fine solver in the $(j+1)^{th}$ interval with the initial value U_j. **b** Timing plots for solving with a single MFS and parallel solving, as well as single-step time comparisons for MCS and MFS. **c** The memristor-based reconfigurable ODE solving system we build with I2C (inter-integrated circuit) communication protocol. **d** Comparison of MSE and computing time for solving the three-body problem using MCS, MFS, and MPS ($J = 5$). **e** Comparison of the results of the coordinates and velocities of the three stars solving by MPS ($J = 5$, iter. = 2) with the ideal case. **f** MSE and computing time for different iteration cycles when solving the three-body problem using MPS ($J = 5$). **g** Comparison of latency and energy consumption of CPU/GPU, MFS and MPS at the same accuracy.

accuracy fine solver that runs in parallel on a number of temporal sub-intervals. This approach allows us to solve in parallel over temporal sub-intervals, which can speed up the numerical process, and the correction using a fine solver ensures the accuracy of the final results.

The flow chart for implementing Parareal on MPS is shown in Fig. 5a. Before starting the solving process, we divide the entire integration interval (t_0 ~ t_n) into J parts, and the initial value of each interval is denoted by U_j ($j = 0, \ldots, J-1$), while we need to solve the numerical integration results reaching the last interval U_J. The symbol $G(U_j)/F(U_j)$ denotes the solving using a coarse or fine solver in the $(j+1)^{th}$ interval with the initial value U_j. Firstly, we make an initial prediction based on the initial condition U_0. Starting from the first interval, we use the coarse solver (MCS) to find the end-of-interval value serially and use them as the initial values for the next interval until we reach the last interval. Secondly, we enter an iterative phase, which is divided into two steps, one for the corrector and one for the predictor. The

corrector uses J parallel fine solvers (MFSs) to obtain more accurate end-of-interval values based on the predicted initial values of each interval. The predictor uses a serial coarse solver to predict the new end-of-interval values, starting from the first interval. We used MCS with the new initial value of each interval to obtain the prediction values. Note that in the kth iteration, the first k intervals have been completely solved by the fine solver, and there is no need to continue iterating. In the next iteration, it is only necessary to start predicting from the $(k+1)^{th}$ interval. In other words, after k iterations, the first k time slices (at minimum) are converged, as the exact initial condition has been propagated by the fine solver at least k times. After each iteration, the differences between the end-of-interval values before and after the correction are used to determine whether to terminate the iteration or not. If all the differences are less than a set threshold, the iteration is terminated; otherwise, it continues to the next iteration.

To clearly illustrate the acceleration effect of Parareal, we compare the latency using a single MFS with Parareal, as shown in Fig. 5b. In this figure, we divide the entire integration interval into J parts ($J = 4$) and assume that it takes the same amount of time to solve each part using MFS (in practice, the solving time will be very slightly different due to different initial conditions). For the single MFS, we need to solve serially J times, assuming that a single solution takes T_F latency. Therefore, a total of $T_{Serial} \approx JT_F$ time is needed. The running time for each iteration is divided into two parts: serial coarse solving and parallel fine solving. Assuming that the initial prediction is the iteration 0, the time consumed by the serial coarse solver in the iteration k is $(J-k)T_G$. For the kth iteration, the time for the parallel fine solvers is the same, both are T_F, although the number of fine solvers that need to be parallelized at the same time becomes $J-k+1$. The total time consumption of our parallel method and the speedup ratio are as follows:

$$T_{Parallel} \approx kT_F + \sum_{i=0}^{k}(J-i)T_G = kT_F + (k+1)(J-\frac{k}{2})T_G \quad (3)$$

$$\text{Speedup} = \frac{T_{Serial}}{T_{Parallel}} \approx [\frac{k}{J} + (k+1)(1-\frac{k}{2J})\frac{T_G}{T_F}]^{-1} \quad (4)$$

From Equation (4), to maximize the acceleration of the parallel method, both the convergence rate k and the ratio T_G/T_F should be as small as possible. For the choice of k, there are detailed mathematical derivations in the previous work[42]. We also carried out experiments to study k on MPS, which is discussed in detail in Supplementary Note 8. We count the average time to perform a one-step computation using MCS and MFS, as shown in the bar graph on the right side of Fig. 5b, from which it can be seen that MFS takes 23.2 × longer time to take a step than MCS of order 2 and 9.94 × longer time than MCS of order 6. Note that when running Parareal on our practical hardware, the fine solver needs to perform more computation steps in order to achieve sufficient accuracy, and thus the proportion of time spent by the coarse solver is even smaller in practice. We perform more detailed latency comparisons of coarse, fine and coarse-to-fine look-ahead solvers in Supplementary Note 8.

The three-body problem is a fundamental model in celestial mechanics. It refers to the problem of the laws of motion of three celestial bodies with arbitrary masses, initial positions, and initial velocities that can be regarded as plasmas under the action of mutual gravity. When three celestial bodies orbit each other, the resulting dynamical system is chaotic under most initial conditions. Since most three-body systems do not have solvable equations, the only way to predict the motion of the three bodies is to estimate it using numerical methods. In the general three-body problem, the equations of motion of each celestial body under the gravitational pull of the other two bodies can be expressed as three second-order ODEs or six first-order ODEs. Therefore, the equations of motion of the general three-body problem are eighteenth-order equations, or equivalently, a system of differential equations consisting of 18 first-order ODEs. A more detailed description and solution methodology are given in Supplementary Note 3.

We demonstrate the solution of this problem on our memristor-based coarse-to-fine look-ahead parallel solver system. We experiment with cases with different J choices to demonstrate the functionality of our system. We use one memristor chip for a serial coarse solver and other memristor chips for parallel fine solvers. The parallel solving of the fine solvers is limited by the number of chips in our system, so for the case where J is very large, we do it in batches with all the chips on the PCB. The specific details of the experiment are shown in Supplementary Note 3. We test the results using only MCS or MFS and compare the first and second iteration results ($U_5^{(1)}/U_5^{(2)}$) when $J = 5$ in our parallel solving system (Fig. 5c) to show the effects of Parareal. As shown in Fig. 5d, MCS is fast but has a worse MSE, while MFS can achieve a better MSE but takes a long time. Our hybrid MPS becomes more accurate as the number of iterations increases. The same accuracy as MFS can be achieved in iteration 2 while greatly reducing the computation time. We plot the experimental results in iteration 2 in Fig. 5e, containing the orbits and velocities of the three stars, where the ideal case is obtained by the software fine solver with stepsize 1×10^{-7} (also used for the true values in the MSE). The solutions for each interval ($J = 5$) are labeled in detail in Fig. 5e, and they all match well with the ideal software solutions. The MSE and the computing time for different iterations are shown in Fig. 5f. We choose iteration 2 because the MSE converges at this point while taking only half the time of MFS. We also compare the latency and energy consumption required to perform this task using CPUs/GPUs, MFS and MPS, as shown in Fig. 5g. The Parareal method consumes more hardware resources for the fastest speed. Compared to MFS, MPS achieves 1.91 × speedup but also increases energy consumption by 2.28 ×. We provide more detailed experiments for different J and different Runge-Kutta methods in Supplementary Note 8. We also record a demonstration video to illustrate the solving process using our parallel ODE solver (Supplementary Video 1).

Discussion

In this article, we report an end-to-end memristor-based ODE solver system using TaOx/HfO2 based memristor chips, periphery circuits, and FPGAs. We develop reconfigurable hardware supporting arbitrary-order Runge-Kutta methods and adaptive step sizes. The proposed solver can be configured into three modes, coarse solver mode for low-accuracy and high-speed applications, fine solver mode for high-accuracy and low-speed scenarios, and coarse-to-fine look-ahead parallel solver mode with a mixture of coarse and fine solvers, which can achieve fine solver accuracy while greatly accelerating the solving speed of the solver. We propose a history-based memristor programming method with a write-verify scheme based on state-space searching to speed up device programming by up to 3.29 ×, while maintaining excellent conductance programming accuracy. We also develop an in-memory stepsize computing method and its associated adaptive stepsize control logic. We experimentally demonstrate the exponential function, the Lorenz attractor, and the three-body problem using each of the three solver modes. Compared with general-purpose CPUs/GPUs, our ODE solver can deliver up to 601 × ~ 6.92×10^3 × speedup and 1.71×10^3 × ~ 3.93×10^3 × energy improvements when solving the same ODE tasks. To the best of the authors' knowledge, we develop the first end-to-end CIM-based ODE solver with a coarse-to-fine look-ahead mechanism, pushing ODE solver hardware performance to a new paradigm with orders of magnitude concurrency improvements.

Methods

1T1R memristor chip fabrication

A 1T1R (one-transistor-one-resistor) memristor crossbar array was designed and fabricated using a 180nm process. The transistors,

fabricated in the Front End of Line (FEOL), function as the select units of the 1T1R cells. Five metal layers were implemented for interconnections. Following the formation of the final tungsten (W) via and the Chemical Mechanical Polishing (CMP) process, the wafers were transferred to a memristor production line for further fabrication. The exposed W vias on the CMOS substrates were initially cleaned with argon plasma to remove native oxide. Subsequently, memristor cells featuring HfO2 as the switching layer were deposited on top of the W vias. The TiN electrodes for both the top and bottom contacts were deposited by sputtering, while the HfO2 dielectric layer was applied using Atomic Layer Deposition (ALD). Finally, an additional via was formed, and the metal layer was deposited to complete the fabrication process.

Hardware ODE solver comparison

The CPU and GPU data mentioned in this work are obtained by running Python codes on an Intel Xeon Gold 6342 and a NVIDIA A100 80GB PCIe GPU, respectively. For ASIC comparison, we build a Runge-Kutta method accelerator using Verilog/RTL and obtain the speed and power consumption data of such an ASIC through synthesis using Synopsys Design Compiler (DC) tools. We design peripheral read/write circuits of memristors on the PCB to implement the ODE solving system (details in Supplementary Note 11). However, since the independent components of the PCB are not specifically designed for our system, the delay and power consumption limited by these components cannot truly reflect the performance of our design. If we create a dedicated chip to implement the ODE solver system, it can achieve better performance. Therefore, we estimate the theoretical performance of our system using state-of-the-art ADC data[43] with the same specifications and combine them with test data from our memristor arrays. The calculation latency is obtained by dividing the operating cycles obtained through our experiments by the maximum frequency of the ADC. Note that at the operating frequency of the ADC, the memristor array and peripheral digital circuits have sufficient time to complete the calculations, and the ADCs are the timing bottleneck. Power consumption is obtained by adding the power of the memristor array, ADC, and peripheral digital circuits. A more detailed comparative description is available in Supplementary Note 13.

Runge-Kutta methods

The proposed ODE solver is built based on Runge-Kutta methods, which are a family of explicit and implicit iterative methods in numerical analysis[37]. The Runge-Kutta methods are the most widely used method in engineering and science and are methods for solving ODEs in mainstream software tools such as MATLAB and Python. Suppose that an initial-value ODE problem is specified as follows:

$$\frac{dy}{dx} = f(x, y),\ y(x_0) = y_0 \tag{5}$$

where y is an unknown function of time x, which we would like to solve numerically. The function $f(x, y)$ and the initial conditions x_0, y_0 are given. The Runge-Kutta methods are used to approximate the values of the function y at any time, that is, $y(x_n)$. In the Runge-Kutta family, there are a number of methods of different orders corresponding to different accuracies in the solution. In general, they all contain a Runge-Kutta matrix $\mathbf{A}$, a weight $\mathbf{b}$ and a node $\mathbf{c}$. They are solving the equation by taking one step forward at a time, and the formulas for each step forward are as follows:

$$\begin{aligned} \mathbf{k} &= f(x_i + h_i{}^*\mathbf{c}, y_i + h_i{}^*\mathbf{kA}) \\ x_{i+1} &= x_i + h_i \\ y_{i+1} &= y_i + h_i{}^*\mathbf{b} \cdot \mathbf{k} \end{aligned} \tag{6}$$

The Runge-Kutta methods are divided into explicit and implicit methods; the implicit method usually possesses better stability and smaller $\mathbf{k}$ of the same order and can be used to solve rigid equations (Supplementary Note 9). The difference between the explicit and implicit methods is that the matrix $\mathbf{A}$ of the explicit method is a lower triangular matrix, so it is a set of serial algorithms. From Equation (6), it can be seen that each step forward requires the calculation of all the k, and the calculation of each k depends on the result of all previous k, which is not suitable for CIM with high parallelism. As for the implicit method, the matrix $\mathbf{A}$ is no longer a lower triangular matrix, which makes the problem become solving a system of equations. So implicit solving is difficult to obtain the analytic solutions directly and requires more hardware resources, time, and energy.

We use fixed-point iteration to transform implicit solving into hardware friendly forms as shown in Equation (1). This method transforms the problem of solving a system of equations into a continuous iterative MVM, suitable for memristor-based CIM. The process also works for explicit RK methods. It can be proved mathematically that if the order of the basic method is p_0, then the order of the iteration is $\min(p_0, \sigma)$[39]. Note that the high parallelism of the memristors makes the time complexity identical when solving the explicit and implicit methods. Implicit methods tend to correspond to higher order and greater stability for the same length $\mathbf{k}$, which makes memristors well suited to implement high-performance ODE solvers. In addition, CIM's high parallelism can also support higher-order Runge-Kutta methods, such as the 10th order methods that has been reported[44,45]. All of the above methods are for first-order ODEs with only one unknown parameter. Multivariate high-order ODEs can be handled in a similar way, as detailed in Supplementary Note 3.

Data availability

The data generated in this study are provided in the Source Data file. The data used in this study are available in the Zenodo database[46]. Source data are provided in this paper.

Code availability

The code that needs to run on our memristor-based ODE solver hardware platform is available in the Zenodo database[47].

References

1. Riley, K. F., Hobson, M. P. & Bence, S. J. *Mathematical Methods for Physics and Engineering: A Comprehensive Guide* (Cambridge University Press, 2006).
2. Boyce, W. E., DiPrima, R. C. & Meade, D. B. *Elementary Differential Equations and Boundary Value Problems* (2021).
3. Mishi, A. H. et al. Application of differential equations in physics. *Global Sci. J.* **8**, 757–773 (2020).
4. Cui, H. et al. The application of linear ordinary differential equations. *Appl. Math.* **11**, 1292 (2020).
5. Jones, D. S., Plank, M. & Sleeman, B. D. *Differential equations and mathematical biology* (Chapman and Hall/CRC, 2009).
6. Ilea, M., Turnea, M. & Rotariu, M. Ordinary differential equations with applications in molecular biology. *Med. Surg. J.* **116**, 347–352 (2012).
7. Borgqvist, J., Ohlsson, F. & Baker, R. E. Symmetries of systems of first order ODEs: symbolic symmetry computations, mechanistic model construction and applications in biology. Preprint at https://doi.org/10.48550/arXiv.2202.04935 (2022).
8. Lee, C.-F. & Shi, J. Application of alternative ODE in finance and economics research. In *Handbook of Quantitative Finance and Risk Management*, 1293–1300 (Springer, 2010).
9. Boucekkine, R., Licandro, O. & Paul, C. Differential-difference equations in economics: on the numerical solution of vintage capital growth models. *J. Econ. Dyn. Control* **21**, 347–362 (1997).
10. Keller, A. A. Generalized delay differential equations to economic dynamics and control. *Am. Math* **10**, 278–286 (2010).

11. Filippov, A. F. *Differential Equations with Discontinuous Righthand Sides: Control Systems* (2013).
12. Sandoval, I. O., Petsagkourakis, P. & del Rio-Chanona, E. A. Neural ODEs as feedback policies for nonlinear optimal control. *IFAC-PapersOnLine* **56**, 4816–4821 (2023).
13. Brockett, R. W. & Willems, J. L. Discretized partial differential equations: Examples of control systems defined on modules. *Automatica* **10**, 507–515 (1974).
14. Al-Omari, A., Arnold, J., Taha, T. & Schüttler, H.-B. Solving large nonlinear systems of first-order ordinary differential equations with hierarchical structure using multi-GPGPUs and an adaptive Runge-Kutta ODE solver. *IEEE Access* **1**, 770–777 (2013).
15. Ahnert, K., Demidov, D. & Mulansky, M. Solving ordinary differential equations on GPUs. In *Numerical Computations with GPUs*, 125–157 (Springer, 2014).
16. Nagy, D., Plavecz, L. & Hegedüs, F. The art of solving a large number of non-stiff, low-dimensional ordinary differential equation systems on GPUs and CPUs. *Commun. Nonlinear Sci. Numer. Simul.* **112**, 106521 (2022).
17. Fasih, A., Do Trong, T., Chedjou, J. C. & Kyamakya, K. New computational modeling for solving higher order ODE based on FPGA. In *Proc. 2009 2nd International Workshop on Nonlinear Dynamics and Synchronization,* 49–53 (IEEE, 2009).
18. Huang, C., Vahid, F. & Givargis, T. A custom FPGA processor for physical model ordinary differential equation solving. *IEEE Embed. Syst. Lett.* **3**, 113–116 (2011).
19. Huang, C., Vahid, F. & Givargis, T. Automatic synthesis of physical system differential equation models to a custom network of general processing elements on FPGAs. *ACM TECS* **13**, 1–27 (2013).
20. Pano-Azucena, A. D., Tlelo-Cuautle, E., Rodriguez-Gomez, G. & De la Fraga, L. G. FPGA-based implementation of chaotic oscillators by applying the numerical method based on trigonometric polynomials. *AIP Adv.* **8**, 075217 (2018).
21. Cai, L. et al. Accelerating neural-ODE inference on FPGAs with two-stage structured pruning and history-based stepsize search. In *Proc. 2023 ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, 177–183 (2023).
22. Bhattacharya, S. & Chakraborty, D. Design and analysis of a hardware accelerator with FPU-based Runge-Kutta solvers. In *Proc. 2023 International Conference on Electrical, Communication and Computer Engineering* (ICECCE), 1–6 (IEEE, 2023).
23. Tao, Y. & Zhang, Z. Hima: A fast and scalable history-based memory access engine for differentiable neural computer. In *Proc. MICRO-54: 54th Annual IEEE/ACM International Symposium on Microarchitecture*, 845–856 (2021).
24. Yao, P. et al. Fully hardware-implemented memristor convolutional neural network. *Nature* **577**, 641–646 (2020).
25. Sebastian, A., Le Gallo, M., Khaddam-Aljameh, R. & Eleftheriou, E. Memory devices and applications for in-memory computing. *Nat. Nanotechnol.* **15**, 529–544 (2020).
26. Wen, T.-H. et al. Fusion of memristor and digital compute-in-memory processing for energy-efficient edge computing. *Science* **384**, 325–332 (2024).
27. Khwa, W.-S. et al. A mixed-precision memristor and SRAM compute-in-memory AI processor. *Nature* **639**, 617–623 (2025).
28. Zhu, Y. et al. MeMCISA: Memristor-enabled memory-centric instruction-set architecture for database workloads. In *Proc. 2024 57th IEEE/ACM International Symposium on Microarchitecture (MICRO)*, 1678–1692 (IEEE, 2024).
29. Zhu, J., Tao, Y. & Zhang, Z. eNODE: Energy-efficient and low-latency edge inference and training of neural ODEs. In *Proc. 2023 IEEE International Symposium on High-Performance Computer Architecture* (HPCA), 802–813 (IEEE, 2023).
30. Yu, L., Jing, Z., Yang, Y. & Tao, Y. Fast and scalable memristive in-memory sorting with column-skipping algorithm. In *Proc. 2022 IEEE International Symposium on Circuits and Systems* (ISCAS), 590–594 (IEEE, 2022).
31. Yu, L. et al. Fast and reconfigurable sort-in-memory system enabled by memristors. *Nat. Electron.* **8**, 597–609 (2023).
32. Lin, Y. et al. Deep Bayesian active learning using in-memory computing hardware. *Nat. Comput. Sci.* **5**, 27–36 (2024).
33. Qi, Y. et al. An efficient and robust partial differential equation solver by flash-based computing in memory. *Micromachines* **14**, 901 (2023).
34. Feng, Y. et al. An efficient flash-based computing-in-memory (CIM) demonstration of high-precision (32-bit) nonlinear partial differential equation (PDE) solver with ultra-high endurance and reliability. *IEEE Trans. Circuits Syst. I: Regul. Pap.* **72**, 3247–3257 (2024).
35. Zidan, M. A. et al. A general memristor-based partial differential equation solver. *Nat. Electron.* **1**, 411–420 (2018).
36. Song, W. et al. Programming memristor arrays with arbitrarily high precision for analog computing. *Science* **383**, 903–910 (2024).
37. De Vries, P. L. & Hasbun, J. E. *A First Course in Computational Physics* (Jones & Bartlett Publishers, 2011).
38. Jiang, Z. et al. COPS: An efficient and reliability-enhanced programming scheme for analog RRAM and on-chip implementation of denoising diffusion probabilistic model. In *Proc. 2023 International Electron Devices Meeting (IEDM),* 1–4 (IEEE, 2023).
39. Alexander, R. Solving ordinary differential equations I: Nonstiff problems (E. Hairer, SP Norsett, and G. Wanner). *SIAM Rev.* **32**, 485 (1990).
40. Lions, J.-L., Maday, Y. & Turinici, G. Résolution d'EDP par un schéma en temps "pararéel". *Comptes Rendus de l'Acad. Sci. Ser. I-Math.* **332**, 661–668 (2001).
41. Gander, M. J. & Vandewalle, S. Analysis of the parareal time-parallel time-integration method. *SIAM J. Sci. Comput.* **29**, 556–578 (2007).
42. Pentland, K., Tamborrino, M., Sullivan, T. J., Buchanan, J. & Appel, L. C. GParareal: a time-parallel ODE solver using Gaussian process emulation. *Stat. Comput.* **33**, 23 (2023).
43. ElShater, A. et al. A 10-mW 16-b 15-MS/s two-step SAR ADC with 95-dB DR using dual-deadzone ring amplifier. *IEEE J. Solid State Circuits* **54**, 3410–3420 (2019).
44. Zhang, D. K. An explicit 16-stage Runge-Kutta method of order 10 discovered by numerical search. *Numer. Alg.* **96**, 1243–1267 (2024).
45. Orapine, H. O., Oladele, J. O., Baidu, A. A. & Vafa, N. E. Derivation of five-stage implicit Runge-Kutta method of order 10 via Gauss-Legendre quadrature for ordinary differential equations. *Int. J. Sci. Glob. Sustain.* **9**, 51–61 (2023).
46. Yu, L. High-concurrency tri-mode memristor-based ordinary differential equation solver, Zenodo. https://doi.org/10.5281/zenodo.17824214 (2025).
47. Yu, L. & Zhu, Y. High-concurrency tri-mode memristor-based ordinary differential equation solver, Zenodo. https://doi.org/10.5281/zenodo.17825196 (2025).

Acknowledgements

This work has been supported by the National Key R&D Program of China (2023YFB4502200; Y.Y.), Guangdong Provincial Key Laboratory of In-Memory Computing Chips (2024B1212020002; Y.Y.), Shenzhen Science and Technology Program (JCYJ20241202125907011; Y.Y.), Beijing Natural Science Foundation (L234026, L257010; Y.Y.), National Natural Science Foundation of China (92164302; Y.Y.), and Financial Support for Outstanding Scientific and Technological Innovation Talents Training Fund in Shenzhen (Y.Y.). This work has been supported by the New Cornerstone Science Foundation (Y.Y.).

Author contributions

L.Y. and Y.T. designed the entire concept and experiment. L.Y., Y.T., and T.Z. were in charge of the hardware system integration, designed

experimental methodologies for each part, and conducted related data analysis. L.Y., Y.T., T.Z., Y.H., B.W., Z.X., H.Z., J.L., L.Y., P.J.T., D.S., and L.C. contributed to memristor testing, chip design, chip fabrication, circuit design, PCB integration, hardware system verification and software simulations. L.Y., Y.T., Y.H., and Z.X. contributed to the interpretation of results. L.Y., Y.T., and Y.Y. wrote the paper with input from all authors. Y.T. and Y.Y. supervised the whole project.

Competing interests

The authors declare no competing interests.

Additional information

Supplementary information The online version contains supplementary material available at https://doi.org/10.1038/s41467-025-68122-4.

Correspondence and requests for materials should be addressed to Yaoyu Tao or Yuchao Yang.

Peer review information *Nature Communications* thanks Jianhua Yang, who co-reviewed with Wenhao Song, and the other anonymous reviewer(s) for their contribution to the peer review of this work. A peer review file is available.

Reprints and permissions information is available at http://www.nature.com/reprints

Publisher's note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Open Access This article is licensed under a Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License, which permits any non-commercial use, sharing, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if you modified the licensed material. You do not have permission under this licence to share adapted material derived from this article or parts of it. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by-nc-nd/4.0/.

© The Author(s) 2026