https://doi.org/10.31891/2219-9365-2026-87-55
UDC 004.272.2:519.6:517.925

RAZOVYI Oleksandr
Khmelnytskyi National University
https://orcid.org/0009-0002-4708-7820
e-mail: razovyioo@khmnu.edu.ua
GOLEVYCH Oleg
Khmelnytskyi National University
https://orcid.org/0009-0001-2987-6722
e-mail: digasgo@gmail.com

PERFORMANCE COMPARISON OF EXPLICIT EULER IMPLEMENTATIONS FOR AN ENSEMBLE OF RUCKLIDGE SYSTEM TRAJECTORIES IN MATLAB

Large-scale integration of independent trajectories of dynamic system is limited not only by the cost of the recurrence step but also by the organization of parallel work, storage of intermediate states, and transfers between memory levels. This study compares explicit Euler implementations for an ensemble of Rucklidge-system trajectories under three output scenarios. A 200 × 200 parameter grid was evaluated with scalar and vectorized MATLAB code on a central processing unit (CPU), ThreadPool/parfor, gpuArray operations on a graphics processing unit (GPU), specialized Compute Unified Device Architecture (CUDA) kernels, and C++ MATLAB executable (MEX) code with Open Multi-Processing (OpenMP) and single instruction, multiple data (SIMD). Each of the 40,000 trajectories contained 307,674 Euler updates. Most configurations were measured in 15 complete repeats and summarized by the median and interquartile range (IQR). With only the final state retained in double precision, two-level CPU parallelism reduced the median time from 45.232 s (IQR 0.744 s) to 2.076 s (IQR 0.163 s), a factor of 21.78. A CUDA kernel that executed 5,000 sequential steps per launch required 2.119 s (IQR 0.0014 s) in double precision and 0.291 s (IQR 0.0014 s) in single precision. For full-state storage, CUDA strict single generated the time series in video memory in 0.598 s (IQR 0.00035 s), but sequential transfer of all blocks to host memory increased the estimated pipeline time to 25.35 s. The 18.873 s CPU parfor full-history result is a critical-path estimate based on the maximum worker time, not host-observed end-to-end runtime. The results show that implementation choice depends on the parallel axis, spatial and temporal block sizes, memory locality, arithmetic precision, and the location of downstream analysis.

Keywords: explicit Euler method; Rucklidge system; MATLAB; vectorization; parfor; CUDA; OpenMP; MEX; chaotic system; batch computation; memory bandwidth.

РАЗОВИЙ Олександр, ГОЛЕВИЧ Олег
Хмельницький національний університет

ПОРІВНЯННЯ ШВИДКОДІЇ РЕАЛІЗАЦІЙ ЯВНОГО МЕТОДУ ЕЙЛЕРА ДЛЯ АНСАМБЛЮ ТРАЄКТОРІЙ СИСТЕМИ РАКЛІДЖА В MATLAB

Швидкодія обчислення значень великих ансамблів незалежних траєкторій динамічних систем обмежується не лише швидкістю власне обчислень, але й організацією паралельної роботи, зберіганням проміжних станів системи і передаванням даних між рівнями пам'яті. У попередніх дослідженнях швидкодії зазвичай використовують різні моделі, розв'язувачі або апаратні конфігурації, тому ці чинники неможливо відокремити за незмінної рекурентної формули. У роботі визначено, яка реалізація явного методу Ейлера забезпечує найменший час виконання для ансамблю траєкторій системи Ракліджа у трьох сценаріях використання даних. На одній сітці параметрів 200 × 200 досліджено скалярні та векторизовані реалізації MATLAB на центральному процесорі (CPU), ThreadPool/parfor, операції gpuArray на графічному процесорі (GPU), спеціалізовані ядра CUDA та реалізації C++ MEX з OpenMP і SIMD. Повний експеримент охоплював 40 000 незалежних траєкторій, 307 674 кроки Ейлера для кожної траєкторії та 12,307 млрд оновлень трьох компонентів стану. Більшість конфігурацій виміряно у 15 повних повторах, а результати узагальнено за медіаною та міжквартильним розмахом (IQR). У сценарії зі збереженням лише кінцевого стану дворівнева CPU-реалізація з вісьмома workers і блоками по 1 750 траєкторій скоротила медіанний час обчислення з подвійною точністю з 45,232 с (IQR 0,744 с) до 2,076 с (IQR 0,163 с), тобто у 21,78 раза. Ядро CUDA, що виконувало 5 000 послідовних кроків Ейлера за один запуск, потребувало 2,119 с (IQR 0,0014 с) з подвійною точністю та 0,291 с (IQR 0,0014 с) з одинарною точністю. У сценарії збереження повної історії CUDA strict single записувала часові ряди у відеопам'ять (VRAM) за 0,598 с (IQR 0,00035 с), однак послідовне передавання всіх блоків до оперативної пам'яті (RAM) збільшувало оцінений час конвеєрної обробки до 25,35 с. Наведене значення 18,873 с для CPU classic+parfor зі збереженням повної історії є оцінкою критичного шляху за максимальним часом worker, а не повним часом розрахунку. Варіанти арифметики CUDA порівнювали за контрольними сумами кінцевих станів, які є діагностичними показниками, а не доказом еквівалентності окремих траєкторій.

Ключові слова: явний метод Ейлера; система Ракліджа; MATLAB; векторизація; parfor; CUDA; OpenMP; MEX; хаотична система; пакетні обчислення; пропускна здатність пам'яті.

Стаття надійшла до редакції / Received 27.05.2026
Прийнята до друку / Accepted 30.06.2026
Опубліковано / Published 10.09.2026

PROBLEM STATEMENT

Building two- or three-parameter bifurcation diagrams means integrating the same dynamical system for many independent parameter combinations. Chaotic systems make this computation heavy because transients are long

and the final regime still has to be classified. Rucklidge introduced the model as a description of double convection [1], and later work examined its bifurcations, routes to chaos, and analytical properties [2, 3]. In this work the model is used as a fixed test problem for comparing MATLAB implementations.

An earlier comparison of 84 continuous-time chaos generators used two-parameter maps together with correlation and spectral criteria, and identified the Rucklidge generator as a strong candidate for broadband telecommunication ensembles [4]. The related dissertation developed parameter-vector searches and parallel tools for identifying chaotic regimes [5]. That work explains why faster large parameter sweeps are useful; the present article focuses on the cost of generating the trajectories.

Steps within a single trajectory form a recurrence, so they must be evaluated in order. Trajectories for different K and L pairs are independent and can run in parallel. Performance therefore depends on how trajectories are grouped, how many CPU or GPU resources are used, whether intermediate states are stored, and how much data must be moved.

ANALYSIS OF RECENT RESEARCH AND PUBLICATIONS

Previous studies parallelized independent parameter-grid points to build two-parameter bifurcation diagrams [6, 7]. Other work classified the resulting regimes with the 0-1 test, entropy measures, or Lyapunov exponents [8-10], and supervised learning has been used to reconstruct diagrams from time series [11]. Coarse-grained CPU parallelism was also evaluated for a fractional Selkov oscillator [12]. These studies treat trajectory generation and regime classification as separate stages, so their costs should also be reported separately.

CUDA studies have shown that bifurcation analysis maps naturally to many independent trajectories [13, 14]. General GPU solvers for ordinary differential-equation ensembles use a similar model, assigning one trajectory or small system to each thread [15]. The Rucklidge right-hand side, however, contains little arithmetic per step. When MATLAB launches work for every step, kernel-launch overhead and repeated global-memory access can dominate.

MATLAB documentation describes vectorization, ThreadPool/parfor, gpuArray, synchronized GPU timing, and the reduction of host-device transfers [16-19]. The cited application studies [6-15] and documentation do not compare all methods on the same recurrence, parameter grid, and initial state while separating three outputs: final state only, full history in local memory, and transfer from VRAM to RAM. The present benchmark fills that gap.

AIM AND OBJECTIVES

The performance of implementations of the same explicit Euler recurrence was evaluated to determine which one gives the lowest median time for an ensemble of independent trajectories under three output scenarios. The scheme, integration step, initial conditions, and parameter grid were kept fixed. MATLAB, GPU/CUDA, and MEX implementations were compared; the trajectory-block size B and the number q of steps executed per CUDA launch were varied; full-history storage in RAM or VRAM and transfer to MATLAB arrays were measured; and repeat variability and diagnostic numerical agreement were reported.

The comparison has three parts. First: CPU, GPU, and MEX implementations run the same recurrence on the same grid. Second: final-state computation, full-history storage, and GPU-to-RAM transfer are timed separately. Third: the experiments isolate the effects of B, q, memory type, and CUDA arithmetic mode.

MATHEMATICAL MODEL AND COMPUTATIONAL EXPERIMENT

The Rucklidge system was used as the test problem. Its state is defined by x, y, and z, dimensionless time by τ, and its dynamics by the dimensionless parameters K and L:

$$\begin{cases} dx/d\tau = -K \cdot x + L \cdot y - y \cdot z \\ dy/d\tau = x \\ dz/d\tau = -z + y^2 \end{cases} \quad (1)$$

System (1) was integrated with the explicit Euler method. For every K and L pair, the state at step n + 1 was computed only from the state at the preceding step n:

$$\begin{cases} x_{n+1} = x_n + \Delta t \cdot (-K \cdot x_n + L \cdot y_n - y_n \cdot z_n) \\ y_{n+1} = y_n + \Delta t \cdot x_n \\ z_{n+1} = z_n + \Delta t \cdot (-z_n + {y_n}^2) \end{cases} \quad (2)$$

Each new state depends on the previous state, so the n = 1,…,N loop remains sequential within one trajectory. The M trajectories for different K and L pairs are independent. Batch implementations therefore store x, y, z, K, and L as vectors and apply Eq. (2) elementwise to each block. Only the organization of independent trajectories changes; the Euler scheme does not [16].

Parameters K and L were varied uniformly over 200 values in each of the closed intervals stated below. Index pair (i, j) was mapped to trajectory m = (i − 1)·200 + j, yielding M = 40,000 trajectories. Each trajectory contained N = 307,674 recurrent updates of the three variables. The dimensionless integration step was Δt = 0.00487. One physical sample interval was Δt·R·C with R = 200 kΩ and C = 1 nF; hence N = round(0.3/(Δt·R·C)) corresponds to a 0.3 s signal of physical generator.

The three-component time series of one trajectory occupies about 7.04 MiB in double precision and 3.52 MiB in single precision. A contiguous 3 × 40,000 × 307,674 double-precision array would require 275.08 GiB. Time series were therefore processed in blocks, with one preallocated buffer reused across blocks and repeats.

The following terms are used: Central Processing Unit (CPU), Graphics Processing Unit (GPU), Random Access Memory (RAM), Video Random Access Memory (VRAM), MATLAB Executable (MEX), Open Multi-Processing (OpenMP), and MATLAB worker. Double and single denote 64-bit and 32-bit floating-point formats, respectively. Experimental parameters and complete software and hardware details are given in Table 1.

Table 1

Parameters of the complete computational experiment

Characteristic	Value
Parameter grid	200 × 200; M = 40,000 trajectories
Trajectory length	N = 307,674 states of x, y, z
Step and physical duration	Δt = 0.0048752849; R = 200 kΩ; C = 1 nF; Tsignal = 0.3 s
Parameter ranges	K = 1.246883…3.311203; L = 2.595717…7.393442
Initial state	Deterministic pseudorandom x_0, y_0, $z_0 \in [0, 5)$; rng(1)
Software and OS	MATLAB R2025b (25.2), Parallel Computing Toolbox 25.2, ThreadPool; Windows 10 Pro 22H2, build 19045.6456, 64-bit
CPU	AMD Ryzen 7 5800H; 8 physical cores, 16 logical processors; 8 ThreadPool workers
GPU	NVIDIA GeForce RTX 3060 Laptop GPU; 6 GB VRAM; compute capability 8.6; driver 610.47; CUDA Toolkit 12.2 reported by MATLAB
Memory	64 GB installed; 59.86 GiB physical; 38.86 GiB history-buffer budget
Compilers and flags	Microsoft Visual C++ 2022 (17.0); CPU MEX: -R2018a, /O2, /openmp:experimental; CUDA MEX: -R2018a, -O, NVCC flag -allow-unsupported-compiler

To relate memory limits to the actual computational granularity, Table 2 lists the partition of the parameter grid in each scenario.

Table 2

Partition of the 200 × 200 parameter grid into trajectory blocks

Scenario	B	Blocks	Last B	Concurrent work	Memory
CPU vector+parfor, final state	1,750	23	1,500	up to 8 tasks	RAM
CPU full history, 1 worker	5,650	8	450	1	RAM
CPU full history, 8 workers	706	57	464	up to 8 tasks	shared RAM
GPU full history, double	546	74	142	1 host block/launch	VRAM
GPU full history, single	1,092	37	688	1 host block/launch	VRAM
GPU strict single, optimum	1,024	40	64	1 host block/launch	VRAM
MEX full history	2,048	20	1,088	OpenMP	RAM

The complete experiment comprised 12.30696 billion recurrent updates of a three-component state. Every table, figure, and conclusion in this article uses complete N = 307,674 runs; shorter search and smoke runs were used only to select candidate configurations and are excluded from the reported results.

MEASUREMENT PROTOCOL AND REPRODUCIBILITY

The implementations differ in their parallel unit, sequential dimension, state location, and output scenario (Table 3).

Table 3

Implementations compared in the computational experiment

Implementation	Parallel unit and sequential dimension	State location	Measured output scenario
MATLAB scalar	One trajectory at a time; sequential time loop	Current state or full history in RAM	Final state; full history
MATLAB scalar + parfor	Independent trajectories distributed across workers; time loop remains sequential	Worker-local RAM	Final state; full-history critical-path estimate
MATLAB vector direct	B trajectories updated elementwise; sequential time loop	Vector state or history in RAM	Final state; full history
MATLAB vector + parfor	Blocks B distributed across workers; vectorized update within each worker	Worker-local RAM	Final state; full-history critical-path estimate
GPU direct (gpuArray)	Several MATLAB-issued GPU operations per Euler step	Current vectors in VRAM	Final state
GPU arrayfun	One compiled GPU kernel per Euler step; N launches	Current vectors in VRAM	Final state
GPU q-step kernel	One GPU thread performs q sequential steps per launch	Current vectors in VRAM	Final state
CUDA history kernel	One thread per trajectory integrates and stores all N states	Full history in VRAM	Full history; strict and FMA
MEX C++	Serial, OpenMP, and SIMD loop organizations	Current state or full history in RAM	Final state; full history

Three scenarios were calculated and measured (Fig. 1):

1. In the final-state-only scenario, only the final x, y, z, and a checksum were retained.
2. In the full-history scenario, all N values of x, y, and z were written to a preallocated RAM buffer for CPU/MEX or a VRAM buffer for CUDA; the buffer was allocated once and reused across blocks and repeats.
3. In the third scenario, gather created an ordinary MATLAB array and copied each complete history block from VRAM. Disk input/output was not timed.

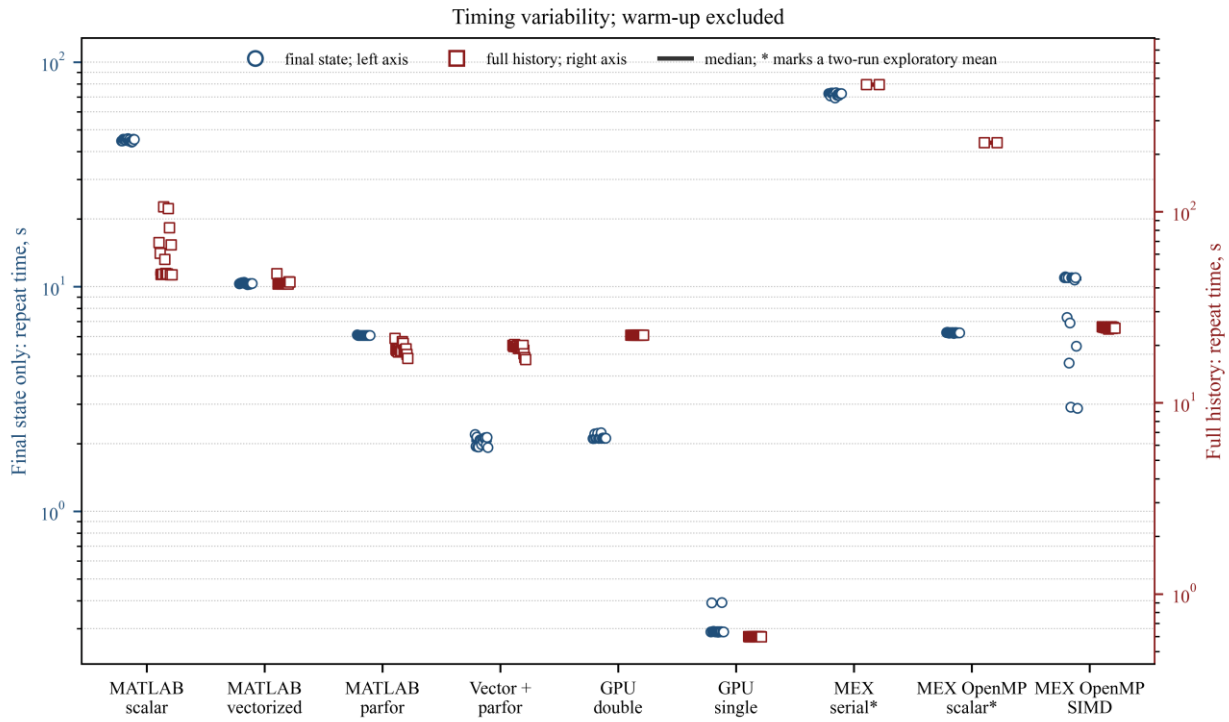


Fig. 1. Distribution of complete measured repeats for selected implementations

In Fig. 1 circles use the final-state-only scenario and the left logarithmic y-axis; squares use full-history storage and the right logarithmic y-axis. Individual values and the median are shown for 15-run series. The two slow MEX full-history configurations show both runs and their exploratory mean. CPU parfor full-history squares are maximum-worker critical-path estimates rather than host end-to-end times. Warm-up is excluded.

Throughput Q was defined as one trajectory update per time step, Q = MN/T, where T is the measured time. The speedup relative to scalar MATLAB was K_t = Tscalar/T. Parallel efficiency was E = (T_1/Tp)/p, where T_1 and Tp

are timings of the same algorithm with one and p workers. This definition keeps algorithmic vectorization gains separate from worker-level parallel efficiency.

For each algorithmic family, Fig. 1 shows the most effective tested configuration for each scenario. Parameters within a pair may therefore differ: vector+parfor uses B = 1,750 without history but B = 706 per worker when writing to RAM; the GPU final-state-only method performs q = 5,000 sequential steps per launch, whereas the full-history method uses a specialized CUDA kernel that computes and immediately stores every state. The figure compares practical output scenarios, not an isolated assignment operation.

A discarded warm-up run preceded every final-state-only series, and the order of configurations was randomized within each repeat. Full-history method families were executed in a fixed order without an additional discarded repeat. Most configurations used 15 complete repeats and are summarized by the median and IQR. The two slow MEX full-history configurations used two complete runs: OpenMP scalar took 229.857 and 229.998 s, and serial scalar took 461.613 and 462.402 s. Their means are reported only as exploratory values. The parpool("Threads", 8) pool was selected because thread-based workers share memory [17]. Reusable-buffer allocation, pool startup, and MEX compilation were excluded from timing. The GPU was synchronized before the timer stopped [18]. For CPU full-history parfor, timers were located inside each worker; the reported value is the median of the maximum worker time and is a critical-path estimate, not host wall time.

The 15 measured repeats demonstrate why robust summaries are necessary. CPU scalar full-history storage had a median of 47.565 s, an IQR of 20.911 s, and a mean of 61.476 s because several system delays exceeded 100 s. The IQRs for CUDA strict double and single were only 0.0056 and 0.00035 s. MEX OpenMP SIMD in the final-state-only scenario formed two timing bands despite an unchanged checksum, so its minimum is not a reproducible performance estimate. Individual repeat times are shown in Fig. 1.

FINAL-STATE-ONLY RESULTS

In the first scenario, no 3 × B × N history arrays were created. CPU and MEX implementations kept only the current x, y, and z vectors, K and L, and auxiliary data in RAM; GPU implementations kept the corresponding working vectors in VRAM. Final states and a checksum were retained after N steps. These timings therefore characterize recurrence evaluation without intermediate-state storage.

Table 4 and Fig. 2 summarize this scenario. Scalar MATLAB required a median of 45.232 s (IQR 0.744 s). Direct vectorization reduced the median to 10.331 s (IQR 0.087 s), a factor of 4.38. Distributing scalar trajectories through parfor required 6.073 s (IQR 0.004 s), a factor of 7.45. The best result, 2.076 s (IQR 0.163 s), used eight thread workers and vector blocks of 1,750 trajectories; K_t was 21.78 and throughput was 5.927 billion updates/s.

Table 4

Final-state-only performance for N = 307,674 and M = 40,000

Method	Resources	Precision	Median [IQR], s	K_t = Tscalar/T	Q, 10^9 updates/s
MATLAB scalar	CPU; 1	double	45.232 [0.744]	1.00	0.272
MATLAB vectorized	CPU; 1	double	10.331 [0.087]	4.38	1.191
MATLAB scalar + parfor	CPU; 8	double	6.073 [0.004]	7.45	2.027
MATLAB vector + parfor, B=1,000	CPU; 8	double	2.416 [0.052]	18.72	5.094
MATLAB vector + parfor, B=1,750	CPU; 8	double	2.076 [0.163]	21.78	5.927
MATLAB vector + parfor, B=2,000	CPU; 8	double	2.120 [0.031]	21.34	5.805
GPU direct single	RTX 3060	single	19.834 [0.612]	2.28	0.621
GPU direct double	RTX 3060	double	19.722 [0.414]	2.29	0.624
GPU arrayfun double	RTX 3060	double	21.751 [0.207]	2.08	0.566
GPU, q=5,000, double	RTX 3060	double	2.119 [0.0014]	21.35	5.808
GPU, q=5,000, single	RTX 3060	single	0.291 [0.0014]	155.41	42.285
MEX OpenMP scalar	CPU; 8	double	6.244 [0.017]	7.24	1.971
MEX OpenMP SIMD	CPU; 8	double	10.894 [4.814]	4.15	1.130

The same complete-run timings and throughputs are compared graphically in Fig. 2.

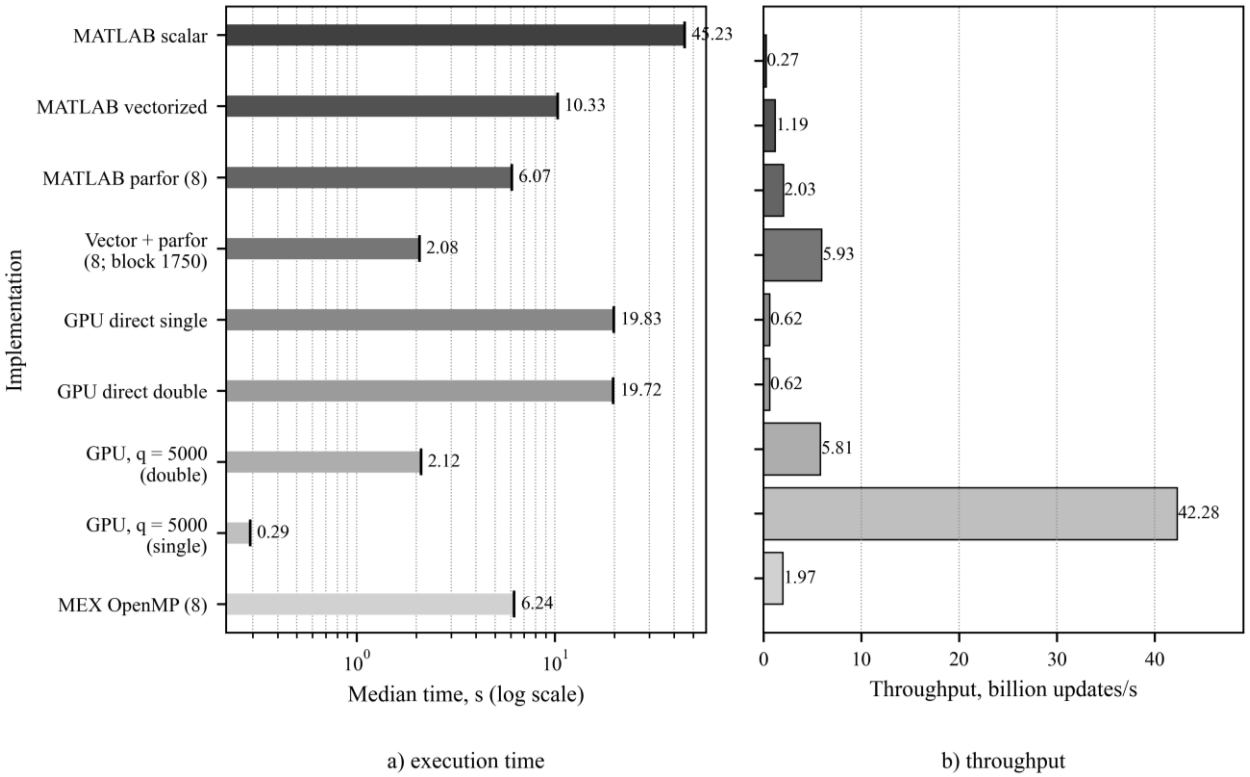


Fig. 2. Complete final-state-only run: a - median computation time; b - throughput. Current states were retained during integration, followed by final states and a checksum. Values summarize 15 complete repeats

For scalar+parfor with eight workers, K_t = 7.45 corresponds to 93.1% parallel efficiency and is consistent with trajectory independence. For the combined method, algorithmic and parallel gains must be separated: relative to single-thread vectorization, eight workers reduced time by a factor of 4.98, giving 62.2% parallel efficiency. The overall K_t = 21.78 combines vectorized data organization and subsequent block distribution; it is not evidence of superlinear parallel scaling.

With eight workers, B = 1,750 and B = 2,000 gave medians of 2.0763 and 2.1200 s, IQRs of 0.1628 and 0.0314 s, and standard deviations of 0.0905 and 0.0379 s. The 2.1% difference between medians is small relative to repeat variability. The evidence therefore supports a practical optimum region of B = 1,750-2,000 rather than a universal single optimum.

FULL-HISTORY RESULTS

In the second scenario, every computed x, y, and z value was written to preallocated arrays (Table 5).

Table 5

Computation with storage of every trajectory state in RAM or VRAM

Method	Memory	Precision	B	Time, s	Statistic and timing scope
CPU classic + parfor	RAM	double	706 × 8	18.873 [IQR 0.470]	median of maximum worker time
CPU vector + parfor	RAM	double	706 × 8	19.647 [IQR 0.501]	median of maximum worker time
CPU vector direct	RAM	double	5,650	41.999 [IQR 0.317]	median host wall time
CUDA strict	VRAM	double	546	22.595 [IQR 0.0056]	median host wall time
CUDA FMA	VRAM	double	546	11.419 [IQR 0.0013]	median host wall time
CUDA strict	VRAM	single	1,024	0.598 [IQR 0.00035]	median host wall time
CUDA FMA	VRAM	single	1,092	0.891 [IQR 0.00064]	median host wall time
MEX OpenMP SIMD	RAM	double	2,048	24.820 [IQR 0.228]	median host wall time
MEX OpenMP scalar	RAM	double	2,048	229.857; 229.998	two complete runs; exploratory mean 229.928 s
MEX serial scalar	RAM	double	2,048	461.613; 462.402	two complete runs; exploratory mean 462.008 s

These arrays resided in RAM for CPU and MEX and in VRAM for CUDA. Each method allocated its buffer before timing and reused it across parameter-grid blocks and repeats. The 38.86 GiB RAM budget allowed either 5,650 serial trajectories or 706 trajectories on each of eight workers. The 3.75 GiB usable VRAM budget held 546 double-precision or 1,092 single-precision trajectories. A pass generated 275.08 GiB of logical double-precision history, but only one active block was resident at a time. Results are listed in.

The distinction between device-local storage and subsequent placement in RAM is shown in Fig. 3.

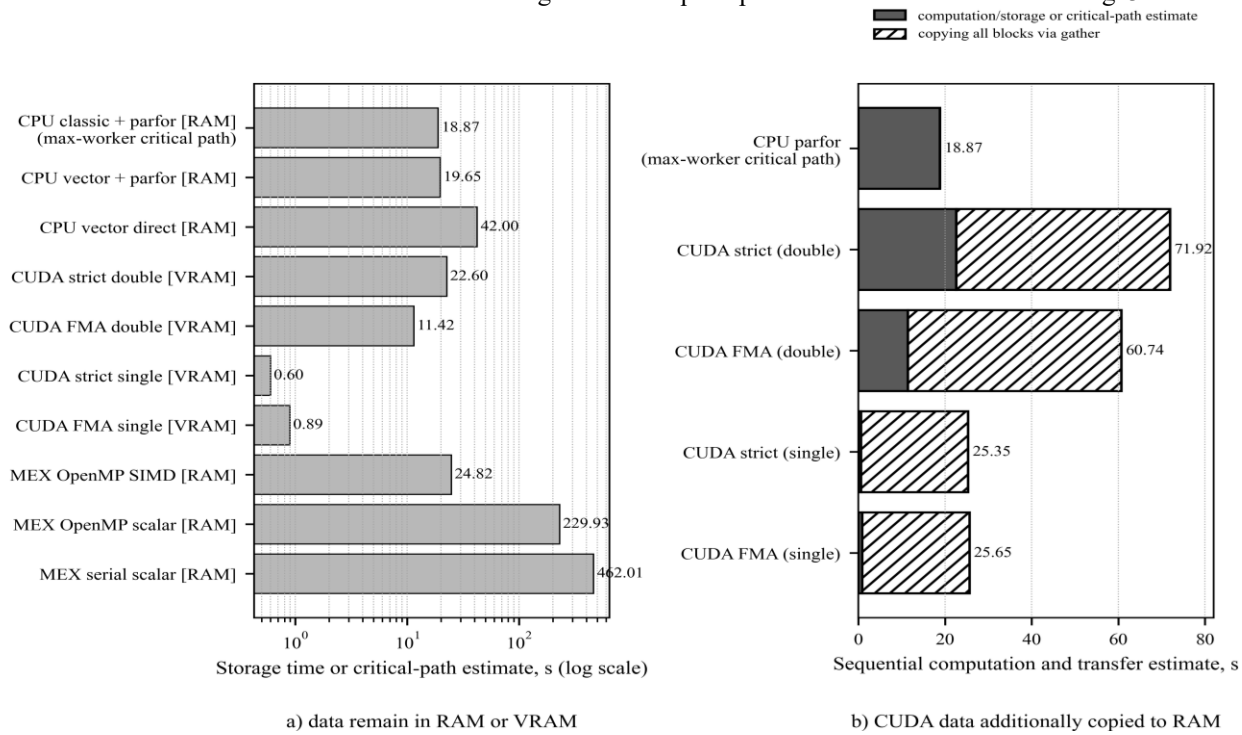


Fig. 3. Full-history scenario: a - computation and storage, after which CPU/MEX data remain in RAM and CUDA data remain in VRAM; CPU parfor values are maximum-worker critical-path estimates; b - estimated sequential pipeline when every CUDA block is additionally copied from VRAM to RAM. Bars show point estimates; IQRs and timing scopes are reported in Table 5.

In double precision, CPU classic+parfor produced a maximum-worker critical-path estimate of 18.873 s (IQR 0.470 s), whereas CUDA strict wrote the histories to VRAM in a host-observed median of 22.595 s (IQR 0.0056 s). The CPU value is not the wall time around the complete parfor statement and must not be interpreted as end-to-end runtime. The methods also finish with data in different memory types, and the complete histories were not compared element by element. CPU-side analysis requires an additional VRAM-to-RAM transfer for CUDA, whereas GPU-side analysis does not. In single precision, CUDA strict with B = 1,024 generated the histories in VRAM in 0.598 s (IQR 0.00035 s). CUDA FMA double was faster but used a different rounding sequence and is treated separately.

GPU WITH FINAL-STATE-ONLY OUTPUT

The initial GPU implementation produced an apparently contradictory result: direct double and direct single required 19.722 and 19.834 s, respectively. Single precision offered no advantage and was about 9.6 times slower than the best CPU vector+parfor result. This does not characterize peak CUDA performance. MATLAB controlled the time loop: direct code issued approximately three GPU operations at each of 307,674 steps, while stepwise arrayfun launched one kernel per step. The recurrence prevents overlap of adjacent steps, so launch overhead and repeated global-memory traffic accumulated [18].

To test the combined effect of launch count and global-memory traffic, a gpuArray.arrayfun-compiled function performed q sequential Euler steps within one CUDA kernel [13, 19]. MATLAB launched only ceil(N/q) kernels, while local x, y, z, K, and L values were reused within each thread. At q = 5,000, the launch count fell to 62. Single precision required a median of 0.291 s (IQR 0.0014 s), 155.4 times faster than scalar CPU and 7.13 times faster than the best CPU vector+parfor configuration. Double precision required 2.119 s (IQR 0.0014 s), only 2.1% slower than the best CPU result. The dependence on q is shown in Fig. 5.

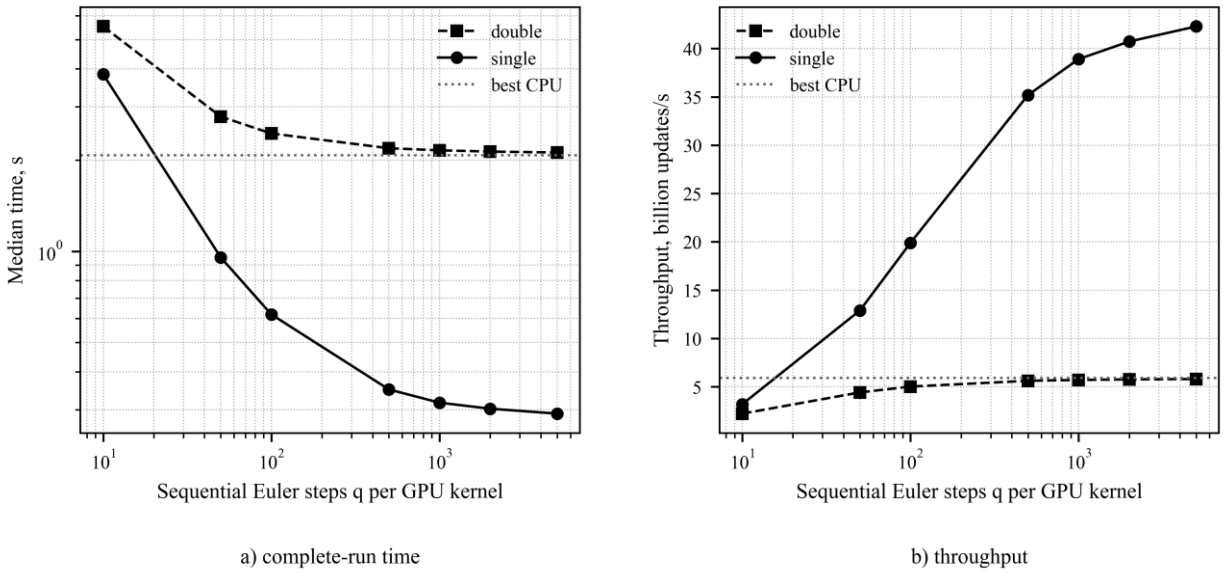


Fig. 4. Effect of q sequential Euler steps per CUDA kernel. Values are medians of 15 complete repeats. Both panels use logarithmic x-axes; panel a also uses a logarithmic y-axis. The dotted line is the best CPU vector+parfor median. Launch count falls from 30,768 to 62, and single-precision throughput reaches 42.28 billion updates/s.

Grouping steps reduces both launches and repeated global-memory traffic. A stepwise kernel for one trajectory reads at least x, y, z, K, and L and writes three new states - approximately 64 bytes per double-precision step. Across 12.307 billion updates, this is about 787.6 GB of logical traffic. At q = 5,000, the same states are read and written only 62 times while intermediate values remain in thread registers, reducing estimated state traffic to about 0.159 GB. High VRAM bandwidth alone therefore did not make the stepwise implementation fast; it could not offset hundreds of thousands of launches and repeated full-array passes.

The single-versus-double difference became visible only after step grouping. Fixed launch costs were distributed across q steps, reducing overhead per step. In direct stepwise code, single precision was 0.6% slower, indicating overhead dominance. At q = 5,000, single was 7.28 times faster than double. For compute capability 8.6, NVIDIA documents substantially higher FP32 than FP64 arithmetic throughput [20]. The measured ratio is also limited by the dependent recurrence and loop control. Double precision favors CPU vector+parfor; when single precision is validated against the downstream analysis metric, the q-step GPU implementation is preferable.

CUDA ARITHMETIC WITH FULL-HISTORY STORAGE

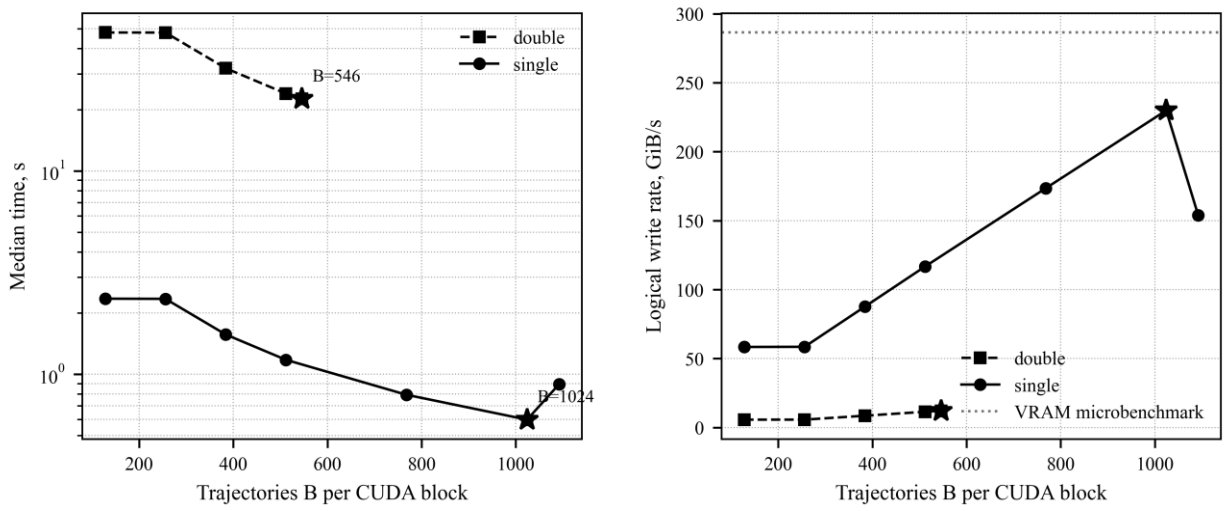
Two arithmetic variants were evaluated for the specialized CUDA kernel that integrates a trajectory and writes every state to VRAM in one invocation. Strict mode reproduces a sequence of separate MATLAB-like multiplications and additions; FMA mode permits hardware contraction. Arithmetic results are given in Table 6, and the CUDA trajectory-block sweep is shown in Fig. 5.

Table 6

Arithmetic modes of the specialized CUDA kernel; 15 complete repeats

Mode	Precision	B / blocks	Median [IQR], s	Checksum CPU	Absolute checksum difference: CPU and CUDA
CUDA strict double	double	546 / 74	22.595 [0.0056]	187794.788165805	1.98×10^{-9}
CUDA FMA double	double	546 / 74	11.419 [0.0013]	186846.967865913	9.48×10^{2}
CUDA strict single	single	1,024 / 40	0.598 [0.00035]	186194.433793289	1.60×10^{3}
CUDA FMA single	single	1,092 / 37	0.891 [0.00064]	187211.916350607	5.83×10^{2}

The effects of B on time and logical VRAM write rate are shown in Fig. 5.



a) computation and VRAM storage time b) logical VRAM write rate

Fig. 5. Effect of CUDA trajectory-block size B during strict full-history storage in VRAM. Points are medians of 15 complete repeats; panel a uses a logarithmic y-axis. A star marks the smallest measured time. The dotted line in panel b is a separate VRAM-bandwidth microbenchmark, not an error bound

Due to NVIDIA CUDA Compiler (NVCC) can contract operations - separate arithmetic tests were required. FMA evaluates a·b + c with one rounding, whereas separate multiply and add instructions round twice. In a chaotic recurrence, even a small local difference can accumulate. At the same B = 546, CUDA FMA double reduced the median from 22.595 to 11.419 s, a factor of 1.98, but its final checksum differed from MATLAB by 947.82. The run therefore cannot be regarded as elementwise equivalent after 307,674 steps; a single checksum also cannot locate or characterize trajectory-wise differences.

CUDA strict used explicit round-to-nearest addition, subtraction, and multiplication intrinsics (__dadd_rn, __dsub_rn, __dmul_rn, and their float counterparts) to prevent FMA contraction and preserve operation order. The double-precision final-state checksum agreed with MATLAB to the printed precision, although the complete histories were not compared element by element. Strict and FMA are separate methods. At B = 1,092, strict single (0.894 s) and FMA single (0.891 s) were nearly identical, consistent with a storage-limited regime. At B = 1,024, strict single fell to 0.598 s, 33.1% below B = 1,092. Because 1,024 is divisible by the CUDA block size of 256 while 1,092 requires a partially occupied fifth block, launch geometry is a plausible explanation, but hardware-counter profiling is required to establish the mechanism.

A final checksum difference does not establish whether FMA changes the qualitative signal properties. Chaotic trajectories can rapidly lose pointwise correlation while remaining on the same attractor with similar invariant characteristics. Comparison of strict, FMA, and single precision using largest Lyapunov exponents, correlation and fractal dimensions, entropy measures, spectra, distributions of extrema, and classification stability on the K-L map will be the objects of the following works. Until then, FMA is a faster but numerically different mode, not an unconditional replacement for strict double. Measured VRAM-to-RAM transfer values are listed in Table **7**.

Table 7

Transfer of one full-history block from VRAM to RAM with gather

Precision	B	Block, GiB	Median [IQR], s	GiB/s	Full grid, s
double	546	3.755	0.673 [0.052]	5.58	49.33
single	1,092	3.755	0.676 [0.027]	5.56	24.76

The gather benchmark created an ordinary MATLAB array in RAM and copied one complete time-series block from VRAM; GPU allocation and initialization were excluded. In double precision, copying 3.755 GiB took a median of 0.673 s (IQR 0.052 s; 5.58 GiB/s), giving a sequential estimate of 49.33 s for all 74 blocks. In single precision, the median was 0.676 s per block (IQR 0.027 s) and 24.76 s for all 37 blocks. The sequential computation-plus-gather estimates are therefore 71.92 s for strict double, 60.74 s for FMA double, and 25.35 s for the best strict single. PCIe transfer dominates when histories are required in RAM. Double buffering could overlap computation and transfer but requires a separate asynchronous test.

MEX LOOP ORGANIZATION AND MEMORY LOCALITY

Compilation to MEX did not guarantee high performance. Without history, serial scalar MEX required a median of 72.488 s, eight-thread OpenMP scalar 6.244 s, and OpenMP SIMD 10.894 s. SIMD repeats formed two timing bands between 2.88 and 10.9 s with an unchanged checksum, so the minimum was not used as the reported result. With full-history storage, the SIMD loop placed time outside and adjacent trajectories inside, producing contiguous RAM writes and a median of 24.820 s (IQR 0.228 s). The scalar MEX loop placed trajectories outside, so successive samples of one trajectory were separated by a trajectory-count stride. The two exploratory complete runs were 461.613 and 462.402 s for serial scalar (mean 462.008 s) and 229.857 and 229.998 s for OpenMP scalar (mean 229.928 s). Loop nesting and spatial write locality, rather than compilation alone, determined performance.

RESULTS AND DISCUSSION

Taken together, the results show that launch frequency and data placement matter as much as raw memory bandwidth. The measured logical read-plus-write bandwidth was 17.78 GiB/s for RAM and 286.39 GiB/s for VRAM; RAM-to-GPU and GPU-to-RAM transfers reached 6.05 and 5.30 GiB/s. Stepwise GPU code remained slow because it paid launch and state-traffic costs at every step. PCIe transfer became the main cost only when complete histories had to be moved into ordinary MATLAB arrays. Table 8 summarizes these results.

Table 8

Memory characteristics and full-history storage cost

Measurement	Result	Conditions and note
RAM read + write	17.78 GiB/s	512 MiB
VRAM read + write	286.39 GiB/s	512 MiB
RAM → GPU	6.05 GiB/s	512 MiB
GPU → RAM	5.30 GiB/s	512 MiB
Full-grid time series	275.08 GiB	calculated logical volume, double
Reusable CPU buffer	38.86 GiB RAM	up to 5,650 trajectories or 706 × 8 workers
Specialized CUDA kernel, double	3.75 GiB VRAM	546 trajectories; 74 blocks
Specialized CUDA kernel, single	3.75 GiB VRAM	1,092 trajectories; 37 blocks

A contiguous 3 × M × N history array would require about 275.08 GiB and cannot fit in the available RAM or VRAM. The final-state-only scenario kept only the last x, y, z, and a checksum. For full-history runs, the grid was split into memory-safe blocks and one preallocated buffer was reused. The 275.08 GiB value is the total data written during a pass, not memory occupied at once.

Keeping every sample is a useful stress test, but it is not required for every bifurcation workflow. Local maxima, Poincaré points, chaos indicators, or other compact features can be computed during integration so that only the final classification is returned [7-11].

CUDA strict double was checked through the final-state checksum; its absolute difference from MATLAB was 1.98×10^{-9}. A checksum is diagnostic and does not prove trajectory-wise equivalence or equality of the stored histories. Single and FMA checksums diverge after 307,674 steps because rounding errors accumulate differently. Their attractors require comparison through distributions, invariant characteristics, and regime classification rather than long-horizon pointwise equality.

LIMITATIONS AND SCOPE OF APPLICABILITY

The absolute timings apply only to the tested hardware and software. RAM and VRAM limits also restricted the full-history values of B, so the measured set cannot reveal a universal MATLAB threshold. Adaptive integrators, other difference schemes, and more complex right-hand sides need separate measurements. Full-history method families ran in a fixed order, so method effects cannot be fully separated from temporal drift.

GPU-to-RAM time was estimated as the sequential sum of actual gather operations for individual blocks; asynchronous overlap requires a dedicated experiment. parfor+gpuArray was not tested because the workstation had one GPU, while a valid test requires a multi-GPU system with one device per worker. Chaos classification, map construction, and invariant-based validation of FMA and single precision remain outside the present benchmark.

CONCLUSIONS AND FUTURE WORK

On this workstation, the fastest method depended on the required output. For final states, eight CPU workers reduced double-precision time from 45.232 to 2.076 s (21.78-fold), while the q = 5,000 GPU method took 2.119 s in double and 0.291 s in single precision. For full histories, CUDA strict single took 0.598 s in VRAM. CPU classic+parfor gave an 18.873 s maximum-worker critical-path estimate in RAM, not host-observed total time; CUDA

strict double took 22.595 s in VRAM and 71.92 s after transfer to RAM. FMA changed the checksum, so future work should measure full parfor wall time and compare invariant properties across precisions and solvers.

References

1. Rucklidge A. M. Chaos in models of double convection. Journal of Fluid Mechanics. 1992. Vol. 237. P. 209-229. https://doi.org/10.1017/S0022112092003392

2. Wang X. Si'lnikov chaos and Hopf bifurcation analysis of Rucklidge system. Chaos, Solitons & Fractals. 2009. Vol. 42, No. 4. P. 2208-2217. https://doi.org/10.1016/j.chaos.2009.03.137

3. Ene R.-D., Pop N., Badarau R. Exact Parametric and Semi-Analytical Solutions for the Rucklidge-Type Dynamical System. Mathematics. 2025. Vol. 13, No. 13. Art. 2052. https://doi.org/10.3390/math13132052

4. Golevych O. B., Pyvovar O. S., Trotsyshyn I. V. The ordering of chaotic signals' ensembles and methods of their use in ultra-wideband telecommunication systems. Digital Technologies. 2015. No. 17. P. 181-191. https://elar.khmnu.edu.ua/handle/123456789/4325

5. Golevych O. B. Ensembles of broadband chaotic telecommunication signals with reduced system interference: Candidate of Technical Sciences dissertation, specialty 05.12.13. Odesa National Academy of Telecommunications named after O.S. Popov. Odesa, 2015. Registration No. 0415U005890. https://uacademic.info/ua/document/0415U005890#!

6. Marszalek W., Sadecki J. Parallel Computing of 2-D Bifurcation Diagrams in Circuits With Electric Arcs. IEEE Transactions on Plasma Science. 2019. Vol. 47, No. 1. P. 706-713. https://doi.org/10.1109/TPS.2018.2871576

7. Marszalek W., Podhaisky H., Sadecki J. Computing Two-Parameter Bifurcation Diagrams for Oscillating Circuits and Systems. IEEE Access. 2019. Vol. 7. P. 115829-115835. https://doi.org/10.1109/ACCESS.2019.2936175

8. Walczak M., Marszalek W., Sadecki J. Using the 0-1 Test for Chaos in Nonlinear Continuous Systems With Two Varying Parameters: Parallel Computations. IEEE Access. 2019. Vol. 7. P. 154375-154385. https://doi.org/10.1109/ACCESS.2019.2948989

9. Marszalek W., Walczak M., Sadecki J. Two-Parameter 0-1 Test for Chaos and Sample Entropy Bifurcation Diagrams for Nonlinear Oscillating Systems. IEEE Access. 2021. Vol. 9. P. 22679-22687. https://doi.org/10.1109/ACCESS.2021.3055715

10. Walczak M., Marszalek W. Computational Issues in Nonlinear Dynamical Systems: Creating Bifurcation Diagrams of Lyapunov Exponents Using a Novel Adaptive Approach. IEEE Access. 2025. Vol. 13. P. 141234-141244. https://doi.org/10.1109/ACCESS.2025.3587410

11. Hassona S., Marszalek W., Sadecki J. Time series classification and creation of 2D bifurcation diagrams in nonlinear dynamical systems using supervised machine learning methods. Applied Soft Computing. 2021. Vol. 113. Art. 107874. https://doi.org/10.1016/j.asoc.2021.107874

12. Tverdyi D., Parovik R. A Study of the Efficiency of Parallel Computing for Constructing Bifurcation Diagrams of the Fractional Selkov Oscillator with Variable Coefficients and Memory. Computation. 2026. Vol. 14, No. 2. Art. 32. https://doi.org/10.3390/computation14020032

13. Pala A., Machaczek M. Computing of 3D Bifurcation Diagrams With Nvidia CUDA Technology. IEEE Access. 2020. Vol. 8. P. 157773-157780. https://doi.org/10.1109/ACCESS.2020.3019633

14. Rybin V., Butusov D., Shirnin K., Ostrovskii V. Revealing Hidden Features of Chaotic Systems Using High-Performance Bifurcation Analysis Tools Based on CUDA Technology. International Journal of Bifurcation and Chaos. 2024. Vol. 34, No. 11. Art. 2450134. https://doi.org/10.1142/S0218127424501347

15. Ahnert K., Demidov D., Mulansky M. Solving Ordinary Differential Equations on GPUs. In: Numerical Computations with GPUs. Cham: Springer, 2014. P. 125-157. https://doi.org/10.1007/978-3-319-06548-9_7

16. MathWorks. Using Vectorization. MATLAB Documentation. [Online]. Available: https://www.mathworks.com/help/matlab/matlab_prog/vectorization.html

17. MathWorks. Choose Between Thread-Based and Process-Based Environments. Parallel Computing Toolbox Documentation. [Online]. Available: https://www.mathworks.com/help/parallel-computing/choose-between-thread-based-and-process-based-environments.html

18. MathWorks. Measure and Improve GPU Performance. Parallel Computing Toolbox Documentation. [Online]. Available: https://www.mathworks.com/help/parallel-computing/measure-and-improve-gpu-performance.html

19. MathWorks. arrayfun: Apply Function to Each Element of Array on GPU. Parallel Computing Toolbox Documentation. [Online]. Available: https://www.mathworks.com/help/parallel-computing/gpuarray.arrayfun.html

20. NVIDIA. CUDA C++ Programming Guide: Throughput of Native Arithmetic Instructions. [Online]. Available: https://docs.nvidia.com/cuda/archive/12.6.0/cuda-c-programming-guide/index.html