Computer Physics Communications 317 (2025) 109819

Contents lists available at ScienceDirect

Computer Physics Communications

journal homepage: www.elsevier.com/locate/cpc

Computational Physics

GBEES-GPU: An efficient parallel GPU algorithm for high-dimensional nonlinear uncertainty propagation

Benjamin L. Hanson [a,*], Carlos Rubio [b], Adrián García-Gutiérrez [b], Thomas Bewley [a]

[a] Department of Mechanical and Aerospace Engineering, University of California San Diego, 9500 Gilman Dr, La Jolla, 92093, CA, USA
[b] Department of Aerospace Engineering, Universidad de León, Av. Facultad de Veterinaria, 25, León, 24004, Spain

ARTICLE INFO

The review of this paper was arranged by Prof. Andrew Hazel

Keywords:
Eulerian uncertainty propagation
Corner transport upwind
Dynamic gridding
Hashtables
CUDA
GPU

ABSTRACT

Eulerian nonlinear uncertainty propagation methods often suffer from finite domain limitations and computational inefficiencies. A recent approach to this class of algorithm, Grid-based Bayesian Estimation Exploiting Sparsity, addresses the first challenge by dynamically allocating a discretized grid in regions of phase space where probability is non-negligible. However, the design of the original algorithm causes the second challenge to persist in high-dimensional systems. This paper presents an architectural optimization of the algorithm for CPU implementation, followed by its adaptation to the CUDA framework for single GPU execution. The algorithm is validated for accuracy and convergence, with performance evaluated across distinct GPUs. Tests include propagating a three-dimensional probability distribution subject to the Lorenz '63 model and a six-dimensional probability distribution subject to the Lorenz '96 model. The results imply that the improvements made result in a speedup of over 1000 times compared to the original implementation.

1. Introduction

Uncertainty propagation (UP) is the process by which the initial uncertainty of a state is evolved over time. While closely related to the state estimation problem, UP focuses on the estimation of the state uncertainty rather than the state itself, although the latter may often be extracted from the former. Like state estimation, UP consists of two repeated steps known as *prediction* and *correction*. During prediction, state uncertainty is propagated via the dynamical system governing the state's evolution. At correction intervals, state measurements are sourced from a measurement model and combined with the predicted uncertainty. In this paper, high-dimensional UP refers to cases where the state dimension $n > 3$, which most commonly arise when the state depends on the position-velocity phase space. This is prevalent in fields such as orbital mechanics and space surveillance [1,2], attitude estimation [2], aircraft navigation [3], robotics and robust control [4], weather prediction and climate models [4], hydrology and geology [4], nuclear physics [4], biological and chemical models [4,5], reliability and safety studies of structural designs [6], and many others.

When the dynamical system and measurement model are linear, and the initial state uncertainty is Gaussian, state uncertainty remains Gaussian globally; therefore, only estimation of the mean and covariance is required for complete representation of the uncertainty, a tractable problem optimally solved by Kalman [7]. The Kalman filter is suboptimal when one or both of the dynamical system and measurement model are nonlinear. In this case, an infinite number of parameters may be required to represent the uncertainty. Nonlinear uncertainty propagation (NUP) numerical methods aim to represent this intractable uncertainty with finite abstractions. These abstractions must be computed efficiently while preserving accuracy, and typically adhere to one of three methodologies: Kalman, Langragian, and Eulerian.

The Eulerian approach to NUP considers and evolves uncertainty by discretizing phase space on a grid. Godunov [8] and Lax and Wendroff [9] designed first-order accurate, grid-based numerical schemes using conservation laws, but Arulampalam et al. [10] highlighted that standard grid-based methods predefined on finite domain spaces suffer from heavy computational cost. Bewley and Sharma [11] addressed this limitation with Grid-based Bayesian Estimation Exploiting Sparsity (GBEES), a novel unstructured gridding scheme that excels at NUP when uncertainty is highly non-Gaussian, tracking uncertainty only where it is non-negligible. Outside of the examples discussed in this paper, GBEES has been validated via the NUP of the Poincaré orbital element dynamics (2D), a Saturn-Enceladus Distant Prograde Orbit (4D), and a Jupiter-Europa Low Prograde Orbit (6D) [12]. A review of NUP methods by

* Corresponding author.
E-mail address: blhanson@ucsd.edu (B.L. Hanson).

https://doi.org/10.1016/j.cpc.2025.109819
Received 10 December 2024; Received in revised form 17 July 2025; Accepted 19 August 2025

Available online 28 August 2025

Hanson et al. [13] validated the accuracy of GBEES but emphasized its inefficiency compared with Kalman and Lagrangian approaches. Davis et al. [14] and Castro et al. [15] employed adaptive mesh refinement paired with grids stored in hash tables in attempts to address this computational cost; because these algorithms are primarily used in computational fluid dynamics (CFD), rarely do they address high-dimensional systems and their complexity.

Of the three numerical methodologies, we assert that the Eulerian approach addresses the problem of NUP most fundamentally, requiring the least abstraction to generally represent and propagate uncertainty. Eulerian approaches do not linearize during prediction or correction, do not require splitting procedures to maintain accuracy, and do not succumb to particle degeneracy. They also happen to be the least explored of the three for high-dimensional systems. Given the primary drawback of this class of methods is computational cost, we argue that the Eulerian approach warrants further investigation for high-dimensional NUP as computational solutions continue to emerge. GPUs are one such promising solution, given that the finite volume schemes are embarrassingly parallelizable, and have been employed often by CFD algorithms for two- and three-dimensional systems (e.g., Ji et al. [16] Jaber et al. [17]). These parallelized algorithms seldom extend to high-dimensional systems.

To address the limitations of high-dimensional Eulerian NUP methods, we introduce GBEES-GPU: a parallel extension of GBEES with improved efficiency achieved by:

1. Storing the dynamic grid in a hashtable
2. Time-marching with a CFL-minimized adaptive step size
3. Employing directional growing and pruning procedures
4. Translating the algorithm to CUDA for single GPU execution

These changes result in an efficient, high-dimensional parallel GPU algorithm for NUP, detailed in the remainder of the paper as follows: In Section 2 the finite volume formulation underlying GBEES is extended to n-dimensions. Section 3 outlines the first key deliverable referenced in the abstract, the improvements made to the CPU implementation, while Section 4 describes the second key deliverable referenced in the abstract, its adaptation to the CUDA architecture. Section 5 presents the use cases employed for validation and testing, including a quantitative measure of accuracy and the evaluation of convergence. Next, Section 6 compares the performance of the improved CPU and GPU implementations against the legacy version. Conclusions are presented in Section 7, and the hardware specifications for the tests are provided in Appendix A.

2. Grid-based Bayesian estimation exploiting sparsity

The equations of motion of a stochastic process $\boldsymbol{X}(t) \in \mathbb{R}^n$ governed by a combination of deterministic and random forces can be described by the following stochastic differential equation:

$$d\boldsymbol{X}(t) = \boldsymbol{f}(\boldsymbol{X}(t),t)dt + \boldsymbol{q}(\boldsymbol{X}(t),t)d\boldsymbol{W}(t), \tag{1}$$

where $\boldsymbol{f}(\boldsymbol{X}(t),t)$ is the drift vector, $\boldsymbol{q}(\boldsymbol{X}(t),t)$ is the diffusion vector, and $d\boldsymbol{W}(t) = \boldsymbol{\xi}(t)dt$ is a Wiener process, meaning $\boldsymbol{\xi}(t)$ is zero-mean, uncorrelated white noise (i.e., $\mathbb{E}[\boldsymbol{\xi}(t)] = \boldsymbol{0}$ and $\mathbb{E}[\boldsymbol{\xi}(t+\tau)\boldsymbol{\xi}^\top(t)] = \boldsymbol{\delta}(\tau)$, where $\mathbb{E}[\cdot]$ is the expectation of $\cdot$). In continuous-time, the Fokker-Planck equation gives the evolution of the probability density function (PDF) $p(\boldsymbol{x},t)$ of $\boldsymbol{X}(t)$ in Eq. (1) as follows:

$$\frac{\partial p(\boldsymbol{x},t)}{\partial t} = -\sum_{j=1}^{n} \frac{\partial f_j(\boldsymbol{x},t)p(\boldsymbol{x},t)}{\partial x_j} + \frac{1}{2}\sum_{j=1}^{n}\sum_{\ell=1}^{n} \frac{\partial^2 Q_{j\ell}(\boldsymbol{x},t)\,p(\boldsymbol{x},t)}{\partial x_j \partial x_\ell} \tag{2}$$

where $\boldsymbol{x} = (x_1,\dots,x_n)$ is a realization of the random variable $\boldsymbol{X}(t)$, $f_j(\boldsymbol{x},t)$ is the j^{th} component of $\boldsymbol{f}(\boldsymbol{x},t)$, and $Q_{j\ell}(\boldsymbol{x},t)$ is the $(j,\ell)^{\text{th}}$ component of the process noise spectral density matrix $Q(\boldsymbol{x},t) = \boldsymbol{q}(\boldsymbol{x},t)\boldsymbol{q}^\top(\boldsymbol{x},t)$. If $Q(\boldsymbol{x},t) > 0$ (i.e., it is positive definite), Eq. (2) is elliptic, but if $Q(\boldsymbol{x},t)$ is relatively small compared to the deterministic forces, Eq. (2) is hyperbolic and satisfies the conservative form of the n-dimensional advection equation:

$$\frac{\partial p(\boldsymbol{x},t)}{\partial t} + \sum_{j=1}^{n} \frac{\partial f'_j(\boldsymbol{x},t)}{\partial x_j} = 0, \tag{3}$$

where $f'_j(\boldsymbol{x},t) = f_j(\boldsymbol{x},t)p(\boldsymbol{x},t)$. At discrete measurement intervals $t^{(k)}$ the PDF is updated via Bayes' theorem:

$$p(\boldsymbol{x},t^{(k+)}) = \frac{p(\boldsymbol{y}^{(k)}|\boldsymbol{x})\,p(\boldsymbol{x},t^{(k-)})}{C}, \tag{4}$$

where $p(\boldsymbol{x},t^{(k+)})$ is the *a posteriori*, $p(\boldsymbol{y}^{(k)}|\boldsymbol{x})$ is the measurement likelihood, $p(\boldsymbol{x},t^{(k-)})$ is the *a priori*, and C is a normalization constant. GBEES performs the accurate, mixed continuous/discrete time-marching of $p(\boldsymbol{x},t)$ using numerical approximations of Eqs. (3) and (4). We first delve into the continuous-time prediction of $p(\boldsymbol{x},t)$ via a fully discrete flux-differencing method.

2.1. Corner transport upwinding for n-dimensional systems

We use the notation from Colella et al. [18] to define an n-dimensional control volume $V_{\boldsymbol{i}}$ as

$$V_{\boldsymbol{i}} = \prod_{j=1}^{n}\left[x_{i_j} - \frac{h_j}{2}, x_{i_j} + \frac{h_j}{2}\right], \tag{5}$$

where the multi-index $\boldsymbol{i} = (i_1,\dots,i_n) \in \mathbb{Z}^n$ identifies the control volume center within the grid, and $\boldsymbol{h} = (h_1,\dots,h_n) \in \mathbb{R}^n_+$ is the grid spacing vector. From Eq. (3), p is assumed to be conserved over $V_{\boldsymbol{i}}$, thus the integral of p varies only due to flux across the boundaries of $V_{\boldsymbol{i}}$. The components of the flux vectors at the forward and backward grid cell interfaces of $V_{\boldsymbol{i}}$ at time step $t^{(k)}$ are defined as

$$F^{(k)}_{\boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}^j,j} \approx \frac{1}{\Delta t \prod_{\substack{\ell=1\\ \ell\neq j}}^{n} h_\ell} \int_{t^{(k)}}^{t^{(k+1)}} \int_{A_{\boldsymbol{i},j}} f'_j(\boldsymbol{x}_{\boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}^j},t)\,d\boldsymbol{A}\,dt, \tag{6}$$

where $A_{\boldsymbol{i},j}$ are the faces bounding the $V_{\boldsymbol{i}}$ with normals pointing in the j^{th} coordinate direction and $\boldsymbol{e}^j$ denotes the unit vector in the j^{th} coordinate direction. The numerical fluxes $F^{(k)}_{\boldsymbol{i}\pm\frac{1}{2}\boldsymbol{e}^j,j}$ are approximated via a Godunov-type finite volume method known as Corner Transport Upwinding (CTU). We now describe the n-dimensional generalization for the CTU method.

First, the Donor Cell Upwind (DCU) method is used to calculate the first-order accurate numerical fluxes. For $\boldsymbol{i} \in \mathcal{I}$, where $\mathcal{I}$ represents the complete set of multi-index vectors in the grid, the upwind flux in the j^{th} direction at time step k is calculated:

$$F^{(k)}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} = f^{+}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} P^{(k)}_{\boldsymbol{i}-\boldsymbol{e}^j} + f^{-}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} P^{(k)}_{\boldsymbol{i}}, \tag{7}$$

where $f^{\pm}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} = \max / \min(f^{(k)}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j}, 0)$, $f^{(k)}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} = f_j(\boldsymbol{x}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j}, t^{(k)})$, and $P^{(k)}_{\boldsymbol{i}}$ is the probability defined at grid cell $\boldsymbol{i}$ at time step $t^{(k)}$.

Eq. (7) only considers probability flowing normal to the grid cell interface, but in general, probability may flow oblique to the interfaces of $V_{\boldsymbol{i}}$. To obtain second-order accuracy, we must account for this with flux corrections. Because the n-dimensional CTU method is notationally complex, we provide it in full in Algorithm 1.

The CTU method is still not second-order accurate, as it is missing high-resolution correction terms. For brevity, and because these correction terms do not depend on dimensionality, we do not restate them here. Instead, they may be found in Bewley and Sharma [11]. In totality, Eq. (7), Algorithm 1, and the high-resolution correction terms result in a Godunov finite volume scheme that is formally second-order accurate; the truncation error analysis proving such may also be found in

Algorithm 1 Corner Transport Upwind.

Require: Perform DCU for $\boldsymbol{i} \in \mathcal{I}$

for $\boldsymbol{i} \in \mathcal{I}$ **do**
  for $j = 1$ to n **do**
    $F^* \leftarrow \frac{\Delta t}{2h_j}\left(P_{\boldsymbol{i}}^{(k)} - P_{\boldsymbol{i}-\boldsymbol{e}^j}^{(k)}\right)$
    for $\ell = 1$ to n, $\ell \neq j$ **do**
      $F^{(k)}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^\ell,\ell} \leftarrow F^{(k)}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^\ell,\ell} - f^{+}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} f^{+}_{\boldsymbol{i}+\frac{1}{2}\boldsymbol{e}^\ell,\ell} F^*$
      $F^{(k)}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^\ell,\ell} \leftarrow F^{(k)}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^\ell,\ell} - f^{+}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} f^{-}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^\ell,\ell} F^*$
      $F^{(k)}_{\boldsymbol{i}-\boldsymbol{e}^j+\frac{1}{2}\boldsymbol{e}^\ell,\ell} \leftarrow F^{(k)}_{\boldsymbol{i}-\boldsymbol{e}^j+\frac{1}{2}\boldsymbol{e}^\ell,\ell} - f^{-}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} f^{+}_{\boldsymbol{i}-\boldsymbol{e}^j+\frac{1}{2}\boldsymbol{e}^\ell,\ell} F^*$
      $F^{(k)}_{\boldsymbol{i}-\boldsymbol{e}^j-\frac{1}{2}\boldsymbol{e}^\ell,\ell} \leftarrow F^{(k)}_{\boldsymbol{i}-\boldsymbol{e}^j-\frac{1}{2}\boldsymbol{e}^\ell,\ell} - f^{-}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j} f^{-}_{\boldsymbol{i}-\boldsymbol{e}^j-\frac{1}{2}\boldsymbol{e}^\ell,\ell} F^*$
    end for
  end for
end for

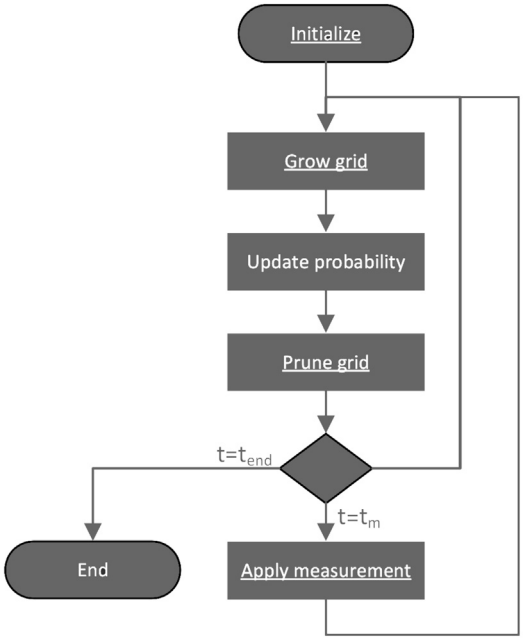

Fig. 1. GBEES simplified flowchart. The underscored operations modify the grid by adding or removing active cells. The PDF is propagated until the end time t_{end} or the time for the next measurement update t_m.

[11]. Given GBEES employs a regular, structured grid scheme, this analysis is sufficient evidence to claim second-order accuracy, as reported by Veluri et al. [19] and Diskin and Thomas [20].

3. Advancements made to the CPU implementation

GBEES excels where other finite volume methods fail by dynamically allocating grid cells where probability is above some threshold. Fig. 1 illustrates this process: in the grow grid phase, new cells are added to the discrete representation of the PDF, and in the prune grid phase, cells with probability below the threshold are discarded. This dynamic grid evolution is iterated either until the end time of the uncertainty propagation is reached or until the next measurement update occurs, at which point the grid-based PDF is updated using a Bayesian approach.

This process is implemented in the legacy algorithm; however, that implementation includes structures and subprocesses that are ripe for optimization. Prior to detailing the GPU implementation, we discuss the efficiency improvements made to the CPU implementation.

3.1. Dynamic grid stored in hashtable

The legacy implementation of GBEES stores the dynamic grid in a nested list data structure. Many functions within GBEES require a searching procedure to check if a given V_i exists in the grid. The time complexity of searching a nested list is $\mathcal{O}(N^2)$, which will result in computational bottlenecks for high-dimensional systems. The first attempt to address this issue employed a binary search tree [21], but overhead of the conversion from grid cell index vector $\boldsymbol{i}$ to unique, positive key value proved too large. Instead, a hashtable was utilized, as the structure allows for collisions between mappings, thus removing the overhead from ensuring bijectivity. Additionally, the time complexity of search for a hashtable is $\mathcal{O}(1)$. Hashtables are discussed further in Section 4.2.

3.2. CFL-minimized adaptive time-marching

For finite volume methods, the Courant–Friedrichs–Lewy (CFL) condition [22] $C \leq C_{\max}$ must be satisfied in order for the method to be stable. The legacy implementation of GBEES employed a static, overconservative time step to ensure stability for the entire propagation period. The new implementation of GBEES uses the following adaptive time step:

$$\Delta t^{(k)} = \min_{\boldsymbol{i}\in\mathcal{I}} \left[\left(\sum_{j=1}^{n} \frac{f^{(k)}_{\boldsymbol{i}-\frac{1}{2}\boldsymbol{e}^j,j}}{h_j} \right)^{-1} \right]. \tag{8}$$

Implementing Eq. (8) maximizes the time step size while ensuring the stability of the explicit finite volume method.

3.3. Directional growing and pruning

To exploit the sparsity of an n-dimensional PDF over phase space, grid cells are tracked where probability is above some threshold p^*. To ensure probability is not lost during time-marching, grid cells neighboring those above threshold are also tracked. In the legacy implementation of GBEES, during the growing procedure, the algorithm loops through all existing grid cells and checks if any of the $3^n - 1$ neighbors that do not exist must be inserted, regardless if probability is likely to flow into the new grid cell in the following step. This can create irrelevant grid cells that are deleted in future steps without ever increasing in probability. In the new GBEES implementation, the direction of the advection is used to determine if a neighboring grid cell is required for the next time step. As is demonstrated in Fig. 2, only the *downwind* grid cells are created; at maximum, this results in checking only $2^n - 1$ neighbors, saving on the number of cells that are inserted in each growth step.

Similarly, during the pruning procedure, the algorithm loops through all existing grid cells, looking for those that are below threshold p^*. In the legacy GBEES implementation, before deleting the negligible cell, the algorithm checks each of the $3^n - 1$ neighbors to see if any are above p^*. Again, this results in redundant cells being saved, as even if a neighboring cell is above threshold, it does not necessarily mean that in the following time steps, it will flow probability into the considered cell. Instead, the new GBEES implementation takes a directional-approach to the growth procedure, wherein only the neighboring grid cells that are *upwind* are checked for probability above p^*. Fig. 3 shows that this requires the algorithm to check at maximum, $2^n - 1$ neighbors, a fraction of the total neighbor cells, while ensuring that negligible cells are not saved, again contribution to the efficiency of the new algorithm.

4. CUDA implementation

The CUDA implementation builds upon the enhancements made to the CPU version described in the previous section. This section details the implementation for the CUDA architecture [23] and the optimization strategies employed.

As described in Section 2, GBEES requires a dynamic grid in which cells are added or removed throughout the integration steps. The scheme depicted in the simplified flowchart of Fig. 1 is representative of the high-level operations of both the CPU and GPU versions. The underscored operations modify the grid by adding or removing active cells.

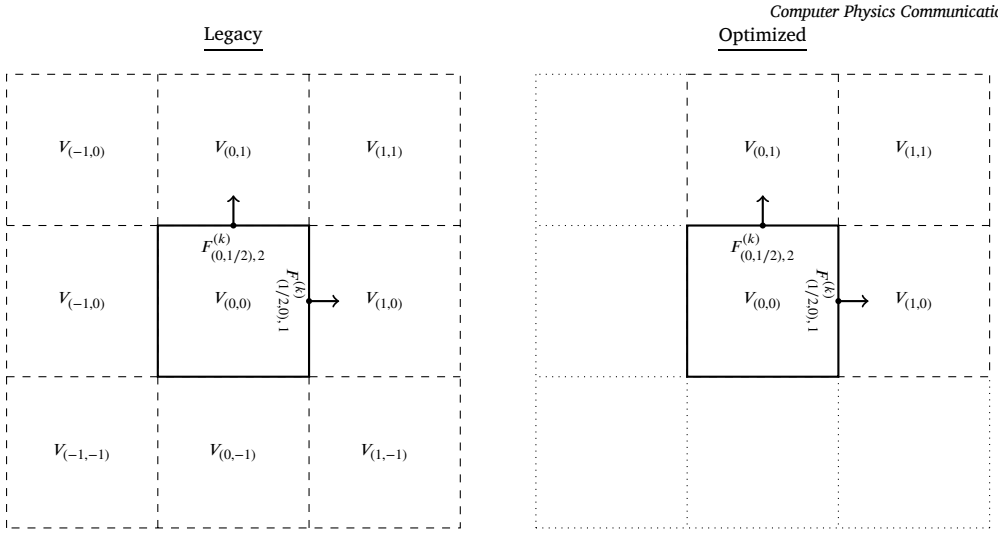

Fig. 2. 2D schematic demonstrating the differences in the growing procedure for the legacy and optimized implementations of GBEES. Solid border cells are those with probability above threshold, dashed border cells are those set to be created during the growing procedure, and dotted border cells are neglected.

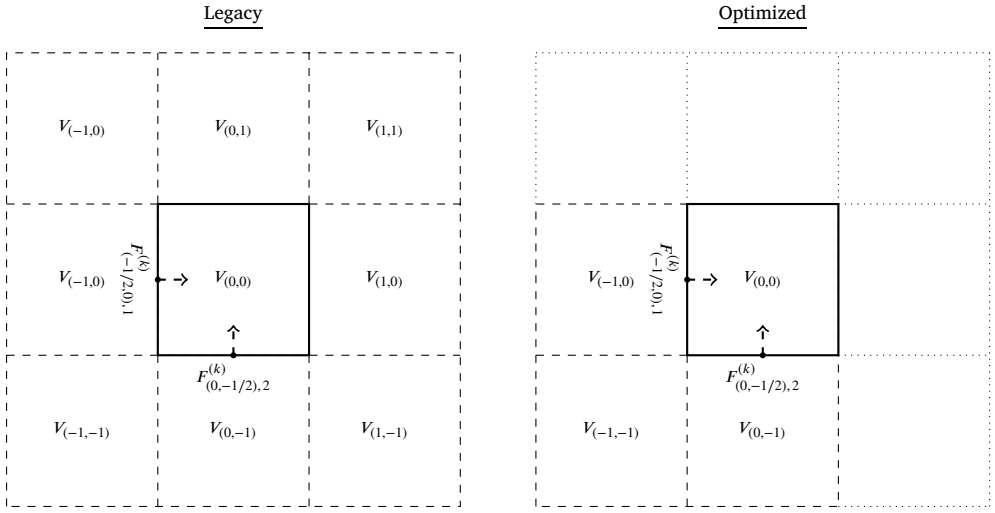

Fig. 3. 2D schematic demonstrating the differences in the pruning procedure for the legacy and optimized implementations of GBEES. Solid border cells are those with probability below threshold, dashed border cells are those checked during the pruning procedure, and dotted border cells are neglected.

Consequently, the dynamic grid not only changes at each integration step but may also be updated multiple times within a single iteration.

This dynamic nature of the grid prevents establishing a fixed mapping between execution threads and cells. In parallel implementations of finite volume software using a static grid [24–27], the grid is partitioned into subdomains (each one including also a boundary with additional halo or ghost cells) and these subdomains are then assigned to thread blocks on the GPU. However, with a dynamic grid, a flexible assignment of cells to threads is required, along with additional synchronization, as thread blocks can no longer operate independently within isolated subdomains.

The problem of handling a dynamic grid alongside GPU parallelization has not been previously addressed in the field of uncertainty propagation. However, a closely related problem arises in Adaptive Mesh Refinement (AMR). AMR [28,29] is used in CFD to refine mesh regions with high gradients, such as shocks or vortices, and is also applied in fields like astrophysics and structural analysis.

AMR involves mesh refinement and coarsening phases, similar to the growth and pruning of the grid in GBEES. Given the difficulty and performance penalty associated with synchronizing the dynamic memory structures on the GPU, some AMR implementations [30–32] adopt a hybrid CPU-GPU approach, where mesh modifications occur on the CPU while flow propagation is computed on the GPU. Other studies have pursued a fully GPU-based implementation by leveraging specialized low-level synchronization mechanisms and data structures, including lists/trees [33,16,34,17] and hashtables [35].

The implementation proposed in this paper combines specific data structures, synchronization algorithms, and parallel techniques to optimize the GBEES-GPU solver. While some elements resemble those used in AMR (such as using hashtables [35], stream compaction [34], and additional lists [33]) they are adapted to suit the distinct nature of the problem.

The dynamic structure of the grid impacts all aspects of the GPU implementation, necessitating more complex synchronization mechanisms and a highly efficient memory layout for storing the grid. This extra global-level synchronization required is provided by the Cooperative Kernel abstraction in CUDA [36]. A Cooperative Kernel requires all threads to be active concurrently, enabling the establishment of global synchronization barriers. This means the maximum number of threads equals the GPU device's maximum simultaneous threads. If the grid contains more cells than this limit, each thread must process multiple cells sequentially.

The thread-cell assignment is therefore dynamic and varies several times at each integration step. Given a set of ℓ threads $(th_1, \ldots, th_\ell)$ and

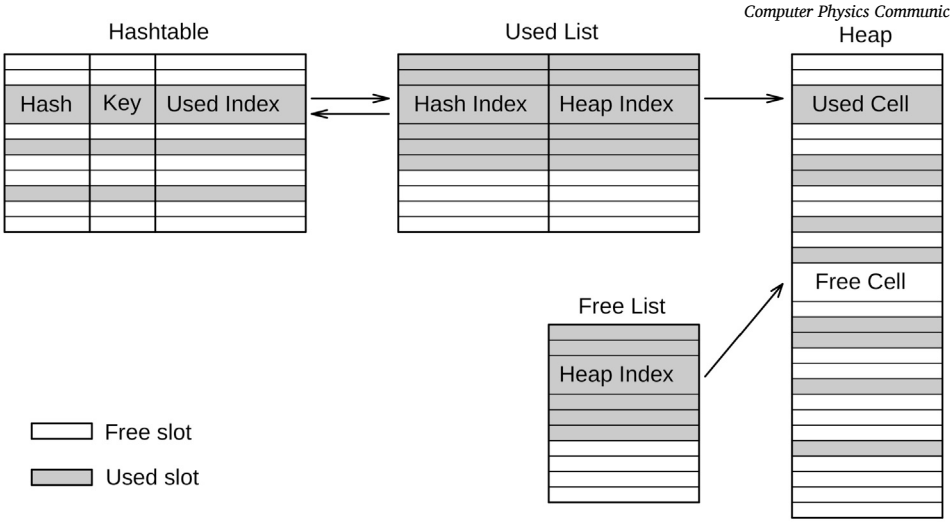

Fig. 4. Grid data structure. Notice that the Used List and the Free List are maintained compact, with all the used slots at the beginning and all the free slots at the end.

m active cells in the current integration step $(c_1, \dots, c_m)$, each thread j processes sequentially the cells $(c_j, c_{j+\ell}, c_{j+2\ell}, \dots)$. The ℓ threads run in parallel, so in the first stage, they process $(c_1, \dots, c_\ell)$; in the second stage, they process $(c_{\ell+1}, \dots, c_{2\ell})$; in the third, $(c_{2\ell+1}, \dots, c_{3\ell})$; and so on until all active cells in the grid have been processed.

4.1. Grid data structure

Given the synchronization requirements, the dynamic thread-cell assignments, and the need for each thread to process a variable number of cells, we propose the data structure for the grid depicted in Fig. 4. This data structure consists of a hashtable, a list to track used cells, another list for unused cells, and a heap table for cell storage.

The hashtable provides fast access to a cell by its key, the multi-index $\boldsymbol{i} = (i_1, \dots, i_n)$. This random access is crucial when accessing the neighbor cells in each dimension. The list of used cells in the grid serves two main purposes. First, it enables quick access to used cells for functions that process individual cells. More importantly, it allows an even distribution of workload among active threads. The Free List maintains a record of unused cells in the heap, reducing the time needed to locate a free slot. Finally, the heap stores the cells themselves. In CUDA, it is not possible to allocate memory directly within a device function in a kernel. Therefore, and for performance reasons, all memory structures are of fixed size, which, for a given configuration, sets the maximum number of cells in the grid.

The relationships among these memory structures are depicted in Fig. 4. In addition to the depicted relationships, each cell contains pointers to its neighboring cells in each dimension by directly storing their indexes of the Used List.

4.2. Hashtable

Because the maximum number of grid cells is fixed, we can set the hashtable size to ensure a maximum occupancy level. The hashtable size is configurable in the software as a multiple of the grid's maximum size, with a default setting of twice that size. This default configuration ensures a maximum occupancy factor $\alpha = 0.5$. This bounded occupancy allows us to use a simple open-addressing scheme with linear probing [37]. With a well-randomized hash function [38]

$$\mathbb{E}[\#\text{ of probes}] = \begin{cases} (1 + 1/(1-\alpha)^2)/2 & \text{unsuccessful search} \\ (1 + 1/(1-\alpha))/2 & \text{successful search} \end{cases}.$$

The open addressing scheme requires marking elements as deleted [37]. In GBEES, all cell deletions occur during the prune grid operation. Since the prune operation is executed only for a subset of integration steps, rehashing the hashtable after each grid prune has minimal impact on performance while ensuring that the maximum occupancy level is maintained. Moreover, in the CUDA implementation, this rehashing is fully parallelized, with all execution threads sharing the workload to rehash the hashtable entries concurrently.

Finally, achieving the expected efficiency requires a well-randomized hash function. The key being hashed is the multi-index $\boldsymbol{i}$, also known as n-gram. Hashing by cyclic polynomial, also known as BuzHash, is an effective method for hashing such n-grams, as described in [39]. Thus, the BuzHash is used by GBEES.

4.3. Main code blocks

From an implementation perspective, we can divide the main code blocks into operations that act on individual cells and those that act on the grid as a whole. The first group is straightforward to implement, as each thread modifies only its assigned cells without significant synchronization issues, requiring only quick access to the cell and its neighbors. This rapid access is achieved through the Used List. This category includes operations such as cell initialization, updating references to the neighbor cells, updating the time step based on the CFL condition [Eq. (8)], computing the DCU and CTU [Eq. (7), Algorithm 1, and high-resolution correction terms], probability distribution normalization, and applying new measurements [Eq. (4)].

The second category involves grid-wide operations, specifically the grid growth and grid pruning. These operations modify a shared global resource, the grid, and therefore require careful synchronization. To optimize CUDA performance, all synchronization is managed using atomic operations and synchronization barriers, either at the block or device level. The following sections detail the key synchronization aspects and parallel techniques applied in these code blocks.

4.4. Synchronization aspects

4.4.1. Grow grid operation

To maximize efficiency in the grid growth operation, a concurrent cell insertion method is required to avoid blocking threads during simultaneous cell creations. Additionally, when exploring different dimensions in the phase space, the algorithm frequently attempts to create cells that already exist. Algorithm 2 presents the chosen compromise solution, which ensures correct synchronization without thread blocking by utilizing atomic operations.

This implementation delays the complete initialization of the cell (using a callback function) until it is confirmed that the cell does not

Algorithm 2 Concurrent Cell Creation.

Require: usedList and freeList are compact
hash ← BuzHash(i)
for count ∈ size(hashtable) **do** ▷ **linear probing**
  hashIndex ← (hash + count) % size(hashtable)
  if hashtable[hashIndex] **is** free **then** ▷ **current slot is empty**
    usedIndex ← atomicAdd[size(usedList)] ▷ **reserve used slot**
    freeIndex ← atomicDec[size(freeList)] ▷ **reserve free slot**
    hashtable[hashIndex].i ← i ▷ **update hashtable and lists**
    usedList[usedIndex].heapIndex ← freeList[freeIndex]
    usedList[usedIndex].hashIndex ← hashIndex
    complete cell initialization with callback function
  else
    if hashtable[hashIndex].i **is** i **then** ▷ **cell already exists**
      break
    end if
  end if
end for

already exist in the grid, thereby improving performance. However, the selected approach has a trade-off: it can only check the existence of a new cell against the previous state of the grid and cannot guarantee successful checking with the other concurrent insertions. To address this limitation, the grid growth operation adopts a staged, directional cell growth strategy:

1. Growth is performed along the forward axis of all dimensions. A global synchronization barrier is then executed
2. Growth is performed along the backward axis of all dimensions, followed by another global synchronization step.
3. Edge growth is carried out in a similar staged manner in the four diagonal directions (forward-forward, forward-backward, backward-forward, and backward-backward).

This staged approach ensures that no concurrent thread attempts to insert the same cell at the same time.

4.4.2. Prune grid operation

The prune grid operation, outlined in Algorithm 3, is less performance-critical as it is executed only once every several integration steps. However, it requires specialized techniques to be performed in parallel by all threads. The operation begins by marking cells whose probability values fall below a specified threshold that do not neighbor any cells exceeding this threshold, identifying them as candidates for pruning. The next step involves performing a parallel prefix sum operation [40,41] to compact the Used List. The prefix sum, also known as scan, computes cumulative sums over a list to facilitate parallel data compaction. This scan is carried out by the active threads, as detailed in the following section on specific parallel techniques. After the scan, the Used List is compacted using a double-buffer scheme, and the freed slots are added to the Free List via atomic operations. Finally, the Hashtable is rehashed, also employing a double-buffer scheme and distributing the rehashing workload across all active threads.

Algorithm 3 Grid Prune Operation.

for $i \in \mathcal{I}$ **do**
  if $i.p < p^*$ **and** i is not a neighbor **then**
    i ← negligible
  end if
end for
perform a prefix sum process of usedList in shared memory
complete the prefix sum of usedList in global memory
compact usedList and update freeList
rehash hashtable
Ensure: perform a global synchronization at the end of each step.

4.5. Specific parallel techniques

The GBEES-GPU implementation employs two high-level parallel techniques: parallel reduction and parallel scan. Parallel reduction is utilized to compute the sum of grid cell probabilities for normalizing the distribution. Parallel scan is applied during the prune operation to compact the Used List. These techniques are widely recognized as standard methods [41], and only a brief description is provided here, focusing on their adaptation to the GBEES kernel's context, which involves multiple concurrent blocks and threads processing several cells each.

In the case of parallel reduction, each thread begins by summing the probability value of all the cells assigned to it. Next, a parallel reduction is performed within each thread block, utilizing shared memory. This intra-block reduction employs a sequential addressing scheme to obtain an optimal shared memory access. Once the reduction within shared memory is complete, a global reduction is performed, involving the first thread of each block. Unlike the intra-block reduction, which uses thread synchronization, the outer reduction relies on global barriers. The final result of the reduction is the sum of the probabilities of all cells.

For the scan operation required to compact the Used List, the process begins with a per-block scan using a double buffer in shared memory. Specifically, an inclusive scan with sequential addressing is employed. Unlike parallel reduction, it is not possible to pre-accumulate the values of all cells processed by each thread. Instead, multiple intra-block scans are performed within each block, with the sums orderly accumulated into a global array. Following this, a second outer scan is conducted at the global level by the first thread of each block. Once the corresponding prefix sums are obtained, each thread populates the compacted Used List in parallel and updates the Free List to account for unused or deleted cells.

5. Validation

5.1. Use cases

In order to validate the GBEES implementation we propagate uncertainty in the Lorenz '63 model (three-dimensional) and the Lorenz '96 model (six-dimensional).

5.1.1. Lorenz '63

The Lorenz '63 model, colloquially referred to as the Butterfly Effect, is often employed to validate the accuracy of NUP methods because of the highly non-Gaussian behavior exhibited [42]. The three-dimensional state and equations of motion are defined as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \sigma(x_2 - x_1) \\ -x_2 - x_1 x_3 \\ -b x_3 + x_1 x_2 - br \end{bmatrix},$$

where $(\sigma, b, r) = (4, 1, 48)$ results in the system being chaotic. In Fig. 5, a 3D Gaussian PDF is initialized at $\boldsymbol{x}^{(0)} = [-11.5, -10, 9.5]^\top$ with standard deviation $\sigma_{x_j} = 1$ for $j = 1, 2, 3$. The uncertainty is then propagated via the system dynamics until the next measurement correction at $t = 1$. To validate the accuracy of GBEES, a Monte Carlo (MC) simulation with identical initial conditions is propagated up to this epoch. Because MC cannot assimilate measurement corrections, the simulation and comparison with GBEES ends here. At $t = 1$, a discrete measurement update is performed with measurement $y^{(1)} = -8$, where the measurement model is

$$y = h(\boldsymbol{x}) = x_3,$$

and the measurement uncertainty $\sigma_y = 1$. The uncertainty is then propagated via the system dynamics until $t = 2$, when the GBEES simulation ends. Fig. 5 illustrates the rapid evolution of the PDF from Gaussian to highly non-Gaussian, with the PDF naturally bifurcating at $t = 1$. Since this model was also used to validate the legacy GBEES implementation,

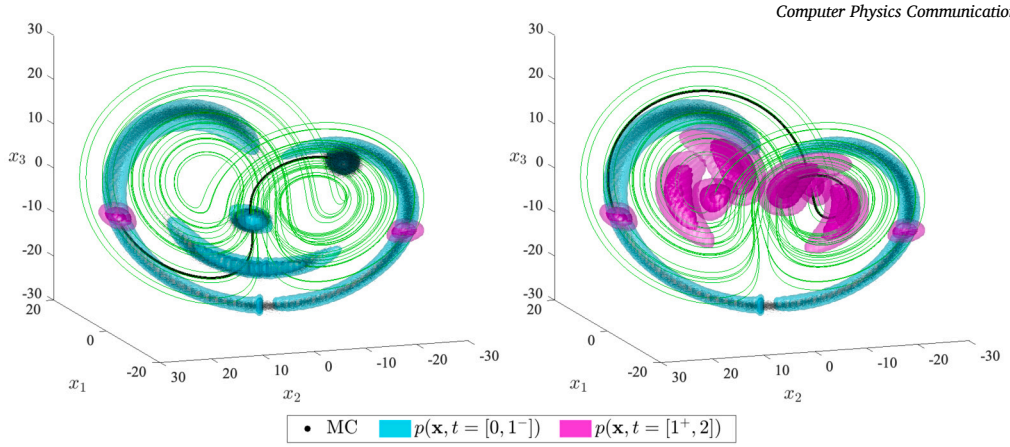

Fig. 5. PDF isosurfaces governed by the Lorenz '63 model in (x_1, x_2, x_3)-space at $p = 0.607$, $p = 0.135$, and $p = 0.011$ with grid cell width $h_j^* = 0.5$ for $j = 1, 2, 3$, compared with a MC simulation with 100,000 samples. On the left (a), the isosurfaces and MC distributions are at $t = 0$, $t = 1/3$, $t = 2/3$, and $t = 1^{\pm}$ and on the right (b), the isosurfaces are at $t = 1^{\pm}$, $t = 4/3$, $t = 5/3$ and $t = 2$. (For interpretation of the colors in the figure(s), the reader is referred to the web version of this article.)

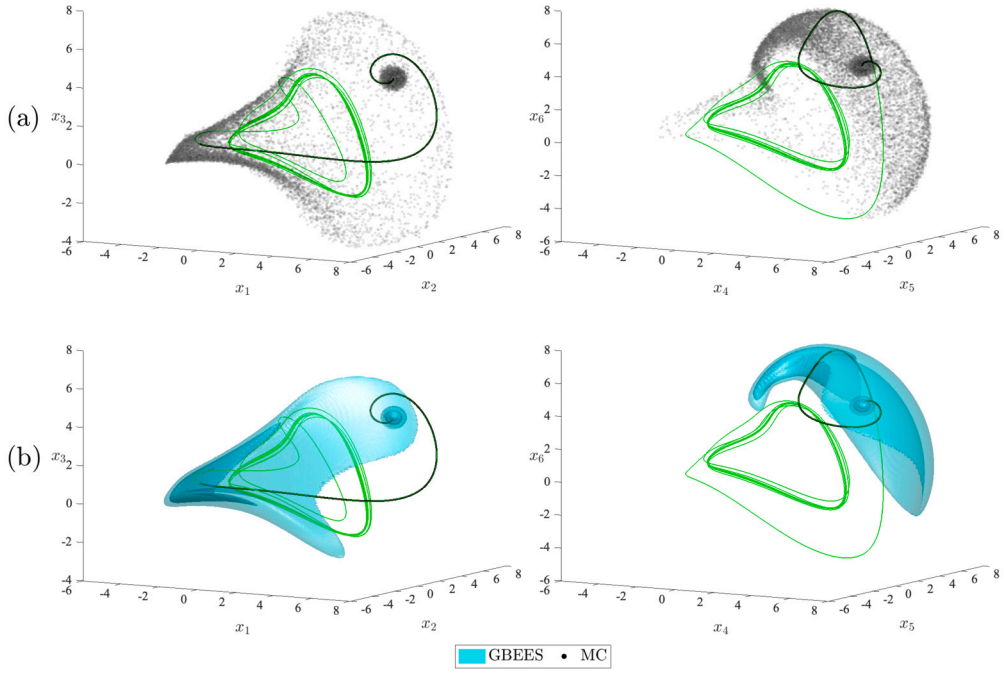

Fig. 6. PDF isosurfaces in (x_1, x_2, x_3)- and (x_4, x_5, x_6)-spaces governed by the Lorenz '96 model with grid cell width $h_j^* = 0.1$ for $j = 1, \ldots, 6$ and $F = 4$. On top (a), the MC point-mass distributions with 10,000 samples at $t = 0$ and $t = 1.3$ and on bottom (b), the GBEES isosurfaces at $p = 0.607$, $p = 0.135$, and $p = 0.011$ for $t = 0$ and $t = 1.3$.

it serves as a valuable basis for performance comparison, as discussed further in Section 6.

5.1.2. Lorenz '96

As an analog to the Lorenz '63 validation first performed in [11], the CPU-optimized and GPU implementations of GBEES are validated on the Lorenz '96 model, a generalized dynamical system that exhibits chaotic behavior [43]. The n-dimensional state and equations of motion are defined as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_j \\ \vdots \\ x_n \end{bmatrix}, \quad \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} (x_2 - x_{n-1})x_n - x_1 + F \\ \vdots \\ (x_{j+1} - x_{j-2})x_{j-1} - x_j + F \\ \vdots \\ (x_1 - x_{n-2})x_{n-1} - x_n + F \end{bmatrix},$$

where $(F, \ldots, F)$ is an unstable equilibrium, with F being a forcing constant. A 6D Gaussian PDF is initialized at $\boldsymbol{x}^{(0)} = [F + 0.5, F, \ldots, F]^\mathsf{T}$ where $F = 4$, with standard deviation $\sigma_{x_j} = 0.2$ for $j = 1, \ldots, 6$. The uncertainty is then propagated via the system dynamics until $t = 1.3$. No measurement update is performed in this simulation. To validate the accuracy of GBEES, an MC simulation with identical initial conditions is plotted, depicted in Fig. 6(a). The 3D PDFs in the (x_1, x_2, x_3)- and (x_4, x_5, x_6)-spaces, shown in Fig. 6(b), are calculated from the discretized 6D PDF by numerically integrating over the (x_4, x_5, x_6)- and (x_1, x_2, x_3)-spaces, respectively:

$$p(x_1, x_2, x_3, t) = \int_{\min(x_6)}^{\max(x_6)} \int_{\min(x_5)}^{\max(x_5)} \int_{\min(x_4)}^{\max(x_4)} p(\boldsymbol{x}, t) dx_4 dx_5 dx_6,$$

$$p(x_4, x_5, x_6, t) = \int_{\min(x_3)}^{\max(x_3)} \int_{\min(x_2)}^{\max(x_2)} \int_{\min(x_1)}^{\max(x_1)} p(\boldsymbol{x}, t) dx_1 dx_2 dx_3.$$

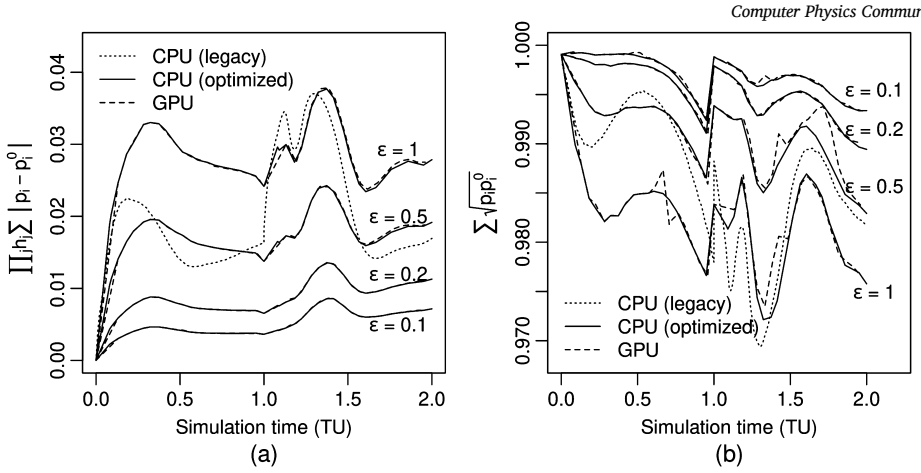

Fig. 7. Lorenz '63 comparison of the discrete probability distributions for different time step sizes (with respect to a reference propagation with $\epsilon = 0.01$). On the left (a), the comparison is based on the 1-norm $E(t)$. On the right (b), the comparison utilizes the Bhattacharyya Coefficient $BC(t)$.

Table 1
BC values comparing the Lorenz '63 and Lorenz '96 GBEES propagations with a KDE representation of the Monte Carlo simulation of 100,000 samples. Lorenz '96 was not tested in the CPU-legacy version due to its high computational cost.

	Lorenz '63 at $t = 1$	Lorenz '96 at $t = 1.3$
CPU-legacy	0.9047	
CPU-optimized	0.9027	0.9155
GPU	0.9027	0.9155

5.2. Accuracy and convergence

In addition to the qualitative validation against the MC simulation shown in Figs. 5 and 6, two quantitative validations are performed. The first compares the PDFs obtained using GBEES with those from a dense MC simulation to evaluate the accuracy of GBEES uncertainty propagation. The second compares different GBEES versions to validate the CUDA implementation and verify proper convergence as the step size is reduced.

5.2.1. Comparison with MC

The quantitative comparison with a dense MC simulation is performed using the Bhattacharyya Coefficient (BC) [44], defined as

$$BC(t) = \sum_{i \in I} \sqrt{p(\boldsymbol{x}_i, t) p^0(\boldsymbol{x}_i, t)}. \tag{9}$$

BC measures the similarity between two probability distributions, where $BC = 1$ indicates perfect coincidence and $BC = 0$ indicates complete dissimilarity. The BC was chosen over other similarity metrics due to $0 \leq BC \leq 1$. A more commonly-used metric, the Kullback–Leibler (KL) divergence [45], approaches ∞ as P and Q approach complete dissimilarity. To ensure the readability of figures illustrating the time evolution of the similarity between the truth and approximate distributions, the BC was selected for use in this paper.

The computation of BC requires that the MC set of samples be first represented as a PDF function. This representation was obtained using kernel density estimation (KDE) with Gaussian kernels and a bandwidth selection using the Scott factor [46]. Table 1 shows the BC values for the different GBEES implementations. In all cases, high values (> 0.9) of BC are obtained, which confirms the similarity observed in Figs. 5 and 6 between the PDFs obtained by the GBEES and the MC method. The specific threshold for sufficient UP accuracy is application-dependent. In this comparison, it is influenced by the number of MC samples, the grid and step sizes of the GBEES configurations, the KDE conversion, and the characteristics of the problem itself.

Finally, note the equality in values between the CPU-optimized and GPU implementations, as both follow the same GBEES algorithm with identical rules for variable step size based on the CFL condition and the same directional growth. That is, the only differences between the CPU-optimized and GPU versions stem from implementation details, which are minimal, as evaluated in the next section.

5.2.2. Convergence of GBEES

The second quantitative validation follows the framework for ensuring convergence delineated by Leveque [47]. To quantify the global error between an approximate distribution and a truth distribution, for methods that depend on conservation laws, the 1-norm is often used as a convergence metric [48,49], defined as:

$$E(t) = \left(\prod_{j=1}^{n} h_j \right) \sum_{i \in I} \left| p(\boldsymbol{x}_i, t) - p^0(\boldsymbol{x}_i, t) \right|,$$

where $p^0(\boldsymbol{x}_i, t)$ represents the truth distribution. A method is then convergent at time T if

$$\lim_{\Delta t \to 0} E(T) = 0. \tag{10}$$

Because the CPU-optimized and GPU versions of GBEES use an adaptive step size, we define Δt for them as a function of the CFL condition, or $\Delta t = \epsilon \times \text{CFL}$. As an analytical truth distribution may not exist in general, we represent $p^0(\boldsymbol{x}_i, t)$ using a reference propagation with an extremely small ϵ. As a supplementary metric of convergence, similarly used by Chen et al. [50], we also assess the approximate distributions using the BC, as previously defined in Eq. (9).

These convergence tests have two primary objectives. First, to ensure the new GBEES implementations qualitatively converge, as demonstrated by decreasing errors relative to the reference propagation when step and cell width sizes are reduced. Second, to assess the impact of the non-deterministic execution order of operations in CUDA. In the CUDA architecture, the execution order of different threads is non-deterministic [36]. Due to the non-associativity of floating-point arithmetic caused by the rounding errors, small variations in the computed probability distribution are expected.

For the temporal convergence test, the reference propagation with $\epsilon = 0.01$ was compared to CPU-legacy, CPU-optimized, and GPU approximate propagations. The CPU-legacy used a fixed step size $\Delta t = 0.005$ and the CPU-optimized and GPU versions used adaptive step sizes with ϵ =1.0, 0.5, 0.2, and 0.1. The results of these comparisons are shown in Fig. 7 for the Lorenz '63 case. The curves demonstrate convergence as defined by Eq. (10), with progressively more accurate values as the step size is reduced. Additionally, the differences caused by the non-deterministic execution order in CUDA are minimal compared to the

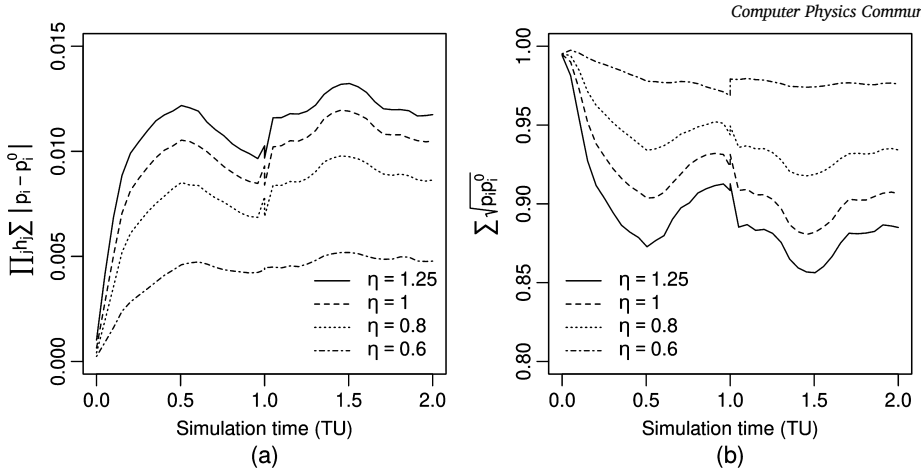

Fig. 8. Lorenz '63 comparison of the discrete probability distributions for different cell widths (with respect to a reference propagation with $\eta = 0.5$). On the left (a), the comparison is based on the 1-norm $E(t)$. On the right (b), the comparison utilizes the Bhattacharyya Coefficient $BC(t)$.

Table 2
Computational burden for the Lorenz '63 and Lorenz '96 models with a variable step size corresponding to a Δt =1.0×CFL and a cell size equal to half the standard deviation in each dimension of the first measurement.

	Lorenz '63	Lorenz '96
Maximum grid size	≈ 24k	≈ 50M
Integration steps	≈ 960	≈ 1050
Total cell computations	≈ 6.5M	≈ 30G

influence of other error sources in the simulation, such as step size variation.

For the spatial convergence test, we run simulations with varying grid cell widths as functions of the default width, or $h_j = \eta \times h_j^*$. As the non-deterministic effects of the GPU version of GBEES were trivial, we use only this version for this test. The reference propagation with $\eta = 0.5$ was compared to propagations with $\eta = 1.25$, 1, 0.8, and 0.6. A CFL number of $C = 0.1$ was set in the simulations for this validation. Fig. 8 indicates that the convergence behavior, in both metrics, is similar to that observed in the temporal convergence analysis.

6. Performance

Performance improvements for both the new CPU and GPU versions are evaluated using the same validation cases described in Section 5. Their computational effort is summarized in Table 2. The significant difference in computational load between the two cases is primarily due to the dimensionality of the models: the Lorenz '63 is formulated over a three-dimensional phase space, while the Lorenz '96 uses a six-dimensional one.

Fig. 9 shows the runtime comparison between the legacy and the new CPU versions for the Lorenz '63 model. The legacy version performs a fixed-step integration, while the new CPU version uses a variable-step scheme. To ensure a fair comparison, the fixed step size of the legacy CPU version was adjusted so that, during the integration, the BCs, computed using the same procedure as in Section 5, reach a similar maximum value.

The runtime results of these executions are also included in Table 3. The relative speed-up of the new CPU version relative to the legacy version is approximately 13.85 times faster (calculated as 1/0.072). To assess performance in the CUDA version, it is essential first to outline the launch configuration parameters and explain how these settings influence overall performance. The GPU launch configuration is determined by three key parameters: the number of blocks, the number of threads per block, and the number of cells each thread processes. The objective of the launch configuration is to maximize the GPU occupancy. Since we have a Cooperative Kernel, this is achieved by launching a total number of threads equal to the GPU's maximum simultaneous thread capacity.

If the maximum grid size exceeds this capacity, each thread must process multiple cells, requiring the parameter for cells processed per thread to be set to a value greater than one. Conversely, if the maximum grid size is smaller than this capacity, the configuration should launch only as many threads as the grid requires, with each thread processing only one cell. In this last case, the achieved occupancy will be less than the theoretical maximum because the model does not expose sufficient parallelism to fully utilize the GPU. Therefore, the launch configuration strategy is to keep the number of cells processed by each thread as low as possible. If the model is sufficiently large, the product of the number

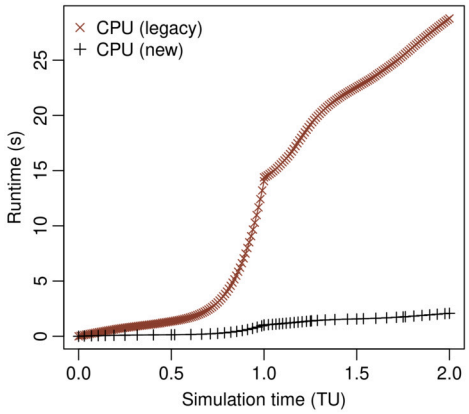

Fig. 9. Runtime comparison between the legacy and the new CPU version for the Lorenz '63 model.

Table 3
Total runtime, number of cells processed per second, and relative speed-up compared to a single-core CPU running the optimized version of GBEES for the Lorenz '63 model. See Appendix A for full specifications of processing units.

Device	Runtime (ms)	Cells/s	Speed-up
CPU-legacy: Apple M2 MAX	28777	≈0.54M/s	0.072
CPU-optimized: Apple M2 MAX	2077	≈3.13M/s	1
GPU 1: NVIDIA Tesla V100	244	≈26.6M/s	8.5
GPU 2: NVIDIA A100	258	≈25.2M/s	8.1
GPU 3: NVIDIA H100	226	≈28.8M/s	9.2
GPU 4: NVIDIA H200	230	≈28.3M/s	9.0

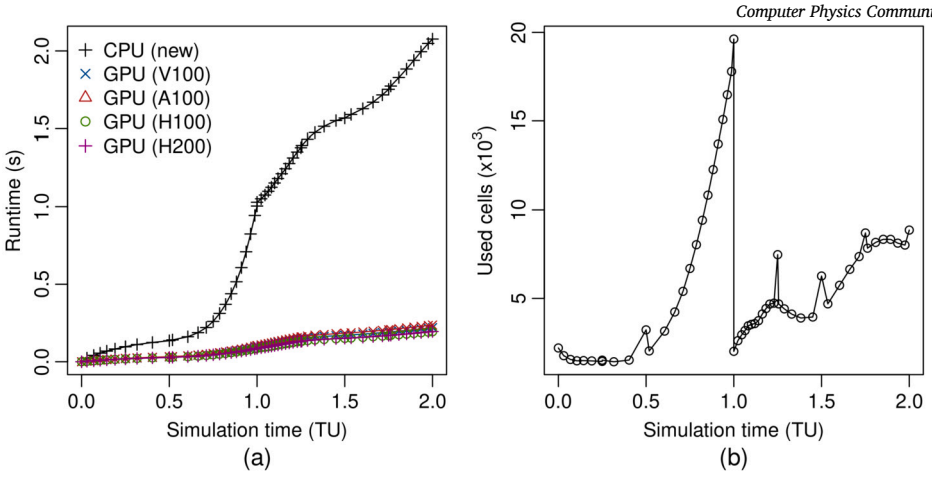

Fig. 10. Runtime comparison between the new CPU version and GBEES-GPU on various GPU devices (a) and number of used cells (b) for the Lorenz '63 model.

of blocks and the number of threads per block should equal the GPU's maximum thread capacity. This product can be achieved through various combinations of blocks and threads per block.

This balance between blocks and threads per block is very subtle. Setting the number of threads per block to the maximum (1024 in the current CUDA architectures) benefits the parallel reduction and scan processes described in Section 4.5, as more computation is performed at the block level using shared memory. However, the performance differences are minimal, and in some tests, using fewer threads per block than the maximum has resulted in slightly better runtimes.

Fig. 10 represents, for the Lorenz '63 model, the runtime comparison between the new CPU version and the CUDA implementation running on different GPU devices. Fig. 10(a) represents the program runtime as a function of the simulated time, where steeper regions correspond to moments when the grid contains more cells. The number of cells during the simulation is plotted in Fig. 10(b). The drop in the number of cells at $t = 1.0$ TU corresponds to a discrete measurement update.

The graph in Fig. 10(a) shows a significant performance boost from parallelizing and executing the algorithm on the GPU. Table 3 summarizes the total runtime, the number of processed cells per second, and the speed-up values. The Lorenz '63 model represents a case where the grid size is not large enough to achieve maximum occupancy on the tested GPUs. This limitation causes the performance to be similar across all devices. Despite this, the speed-up achieved by using the CUDA version ranges from 8.5 to 9.0, depending on the specific GPU tested.

The Lorenz '96 model requires a high computational burden and exposes enough parallelism to fully utilize the performance of the tested GPUs. This results in a significant performance difference between the CPU and CUDA versions. To highlight this difference and facilitate representation, Fig. 11 first presents the runtime comparison between the new CPU version and the CUDA implementation running on a V100 GPU device, which is the slowest among the tested GPUs.

The runtime data for this comparison, along with comparisons to other GPU devices, are included in Table 4. The execution of the Lorenz '96 model is 17.8 times faster in the CUDA version on the V100 device compared to the new CPU version. For the other devices, Fig. 12 shows that, as the model fully utilizes the GPUs, there is a progressive reduction in execution times corresponding to the increasing computing power of the different test devices. The speed-ups achieved are 56.4 times for the A100 device, 106.6 times for the H100, and 132.5 times faster for the H200 device.

The observed execution time improvements surpass one order of magnitude between the legacy and new CPU versions —as demonstrated in the Lorenz '63 use case— and two orders of magnitude between the new CPU version and the GPU implementation. Together, these results indicate that the enhanced GBEES algorithm achieves a total performance improvement of more than three orders of magnitude.

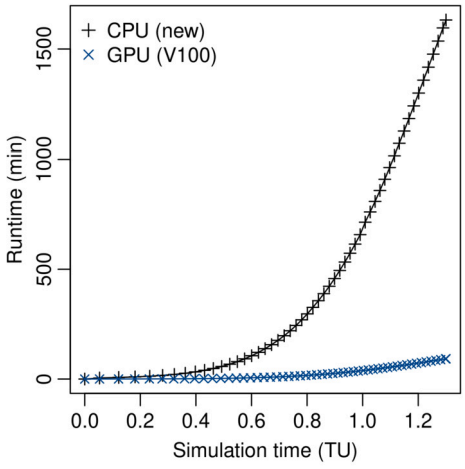

Fig. 11. Runtime comparison between the new CPU version and the GBEES-GPU execution on a V100 GPU device for the Lorenz '96 model.

Table 4
Total runtime, number of cells processed per second, and relative speed-up compared to a single-core CPU running the new version of GBEES for the Lorenz '96 model.

Device	Runtime (s)	Cells/s	Speed-up
CPU-optimized: Apple M2 MAX	97927	≈0.3M/s	1
GPU 1: NVIDIA Tesla V100	5513	≈5.4M/s	17.8
GPU 2: NVIDIA A100	1736	≈17.3M/s	56.4
GPU 3: NVIDIA H100	919	≈32.6M/s	106.6
GPU 4: NVIDIA H200	739	≈40.6M/s	132.5

7. Conclusions

This paper presents a CPU-optimized implementation and a GPU implementation of GBEES, a second-order accurate, Eulerian algorithm for robust nonlinear uncertainty propagation. To address the computational limitations associated with the legacy CPU implementation of GBEES, the main data structure was changed from a linked list to a hashtable, a CFL-minimized adaptive step size was used, and the grid growing and pruning procedures were adjusted to consider the advection direction. Once the CPU implementation was optimized, the algorithm was translated to CUDA for single GPU execution.

The CUDA implementation is heavily influenced by the dynamic nature of the grid required by the GBEES method. This dynamic grid demands more sophisticated synchronization mechanisms and an efficient memory layout for its storage. To address these challenges, the

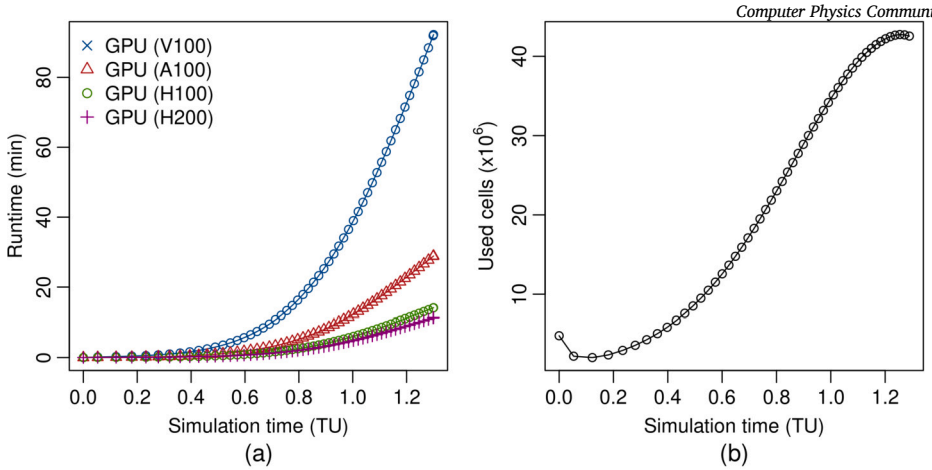

Fig. 12. Runtime comparison between the new CPU version and GBEES-GPU on various GPU devices (a) and number of used cells (b) for the Lorenz '96 model.

CUDA implementation employs the Cooperative Kernel abstraction, an ad-hoc data structure to store the grid, non-blocking algorithms to modify it, and parallel-specific optimization techniques.

We validate the two novel implementations using accuracy and convergence. To measure accuracy, we calculate the Bhattacharyya Coefficient (BC) of the GBEES distribution compared with a dense MC distribution, assuming values greater than 0.9 represent sufficient similarity. To measure convergence, we evolve a truth distribution of GBEES using a time step that is $1/100^{\text{th}}$ of the CFL-based stability limit. We then compare various GBEES implementations, beginning with the CFL time step then decreasing in step size, evaluating convergence through both the 1-norm and the BC. We deem "convergence" as the approach of the reference propagation as the adaptive step size is reduced.

The framework is applied to two chaotic systems: the first is the three-dimensional Lorenz '63 model. For this use case, the BC remains above 0.9 for each of the GBEES implementations. The inability of the MC distribution to assimilate corrections emphasizes its inefficacy when a measurement model is present. Additionally, both metrics indicate convergence. The CPU-optimized implementation has a performance increase of 14× relative to the CPU-legacy implementation and the GPU implementation has a performance increase of 9× relative to the CPU-optimized implementation.

Finally, we demonstrate the full capability of the new implementations on a six-dimensional variation of the Lorenz '96 model, an n-dimensional chaotic system. For this use case, the BC for the CPU-optimized and GPU implementations compared with a dense MC distribution remains above 0.9 (implementing this example with the CPU-legacy version is computationally infeasible). Convergence metrics also demonstrated convergence. The high dimensionality of the system highlights the efficacy of the parallelized algorithm; the GPU implementation has a performance increase of 133× relative to the CPU-optimized implementation, implying a 1000-fold increase in performance relative to the CPU-legacy implementation.

CRediT authorship contribution statement

Benjamin L. Hanson: Writing – original draft, Visualization, Validation, Software, Methodology. **Carlos Rubio:** Writing – original draft, Validation, Software, Methodology. **Adrián García-Gutiérrez:** Validation. **Thomas Bewley:** Writing – review & editing, Supervision, Project administration, Methodology, Conceptualization.

Code availability

In the interest of facilitating further research, promoting its use, and allowing the reproduction of the experiments, the complete source code is available in the following repositories:

- The GBEES-CPU-optimized code is available at https://github.com/bhanson10/gbees under the BSD 3-Clause License.
- The GBEES-GPU code is available at https://github.com/Cx-Rubio/gbees-cuda under the BSD 3-Clause License.

The provided software can be easily extended with new dynamic models. Documentation on how to compile and execute the software, the format of the input and output data, and instructions for extending the software by adding additional models can be found in the repositories themselves and in the corresponding instruction guide [12].

Declaration of competing interest

The authors declare the following financial interests/personal relationships which may be considered as potential competing interests: Benjamin Hanson reports financial support was provided by the NASA Space Technology Graduate Research Opportunities Fellowship (80NSSC23K1219). Benjamin Hanson reports equipment, drugs, or supplies was provided by San Diego Supercomputer Center at UC San Diego. If there are other authors, they declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgements

The authors thank the San Diego Supercomputer Center for a computing time allocation on the Triton Shared Computing Cluster.

Appendix A. Benchmark hardware specifications

The CPU execution times were measured on a system with the following specifications:

- CPU: Apple M2 Max, 12-core CPU. Clock frequency 8 cores × 3.7GHz, 4 cores × 3.4 GHz. L2 cache size 36 MB.

The GPU performance tests were executed in the next GPU devices:

- GPU 1: NVIDIA Tesla V100-SXM2-32GB, CUDA architecture Volta. Stream multiprocessors (SMs) 80. Maximum threads per SM 2048. SM clock frequency 1.530 GHz. Memory clock frequency 0.877 GHz. Memory 32 GB HBM2.
- GPU 2: NVIDIA A100-SXM4-40GB, CUDA architecture Ampere. Stream multiprocessors (SMs) 108. Maximum threads per SM 2048. SM clock frequency 1.410 GHz. Memory clock frequency 1.215 GHz. Memory 40 GB HBM2e.

- GPU 3: NVIDIA H100-80GB, CUDA architecture Hopper. Stream multiprocessors (SMs) 132. Maximum threads per SM 2048. SM clock frequency 1.980 GHz. Memory clock frequency 2.619 GHz. Memory 80 GB HBM3.
- GPU 4: NVIDIA H200-141GB, CUDA architecture Hopper. Stream multiprocessors (SMs) 132. Maximum threads per SM 2048. SM clock frequency 1.980 GHz. Memory clock frequency 3.201 GHz. Memory 141 GB HBM3e.

Appendix B. Supplementary material

Supplementary material related to this article can be found online at https://doi.org/10.1016/j.cpc.2025.109819.

Data availability

The code is available on Github

References

[1] A.B. Poore, J.M. Aristoff, J.T. Horwood, Covariance and Uncertainty Realism in Space Surveillance and Tracking, Tech. Rep., Numerica Corporation, 2016.
[2] M. Vetrisano, Uncertainty Quantification and State Estimation for Complex Nonlinear Problems in Space Flight Mechanics, Ph.D. thesis, University of Strathclyde, 2016.
[3] S. Sankararaman, M. Daigle, Uncertainty quantification in trajectory prediction for aircraft operations, in: AIAA Guidance, Navigation, and Control Conference, American Institute of Aeronautics and Astronautics, Grapevine, Texas, 2017, p. 1724.
[4] R.C. Smith, Uncertainty Quantification: Theory, Implementation, and Applications, Computational Science & Engineering, vol. 12, SIAM, Society for Industrial and Applied Mathematics, Philadelphia, 2014.
[5] H.N. Najm, B.J. Debusschere, Y.M. Marzouk, S. Widmer, O.P. Le Maître, Uncertainty quantification in chemical systems, Int. J. Numer. Methods Eng. 80 (6–7) (2009) 789–814, https://doi.org/10.1002/nme.2551.
[6] B. Sudret, Uncertainty propagation and sensitivity analysis in mechanical models, Ph.D. thesis, Université Blaise Pascal - Clermont Ferrand II, 2007.
[7] R.E. Kalman, A new approach to linear filtering and prediction problems, J. Basic Eng. 82 (1) (1960) 35–45, https://doi.org/10.1115/1.3662552.
[8] S.K. Godunov, A difference scheme for numerical solution of discontinuous solution of hydrodynamic equations, Mat. Sb. 47 (1959) 271–306, https://cir.nii.ac.jp/crid/1573387449783562752.
[9] P. Lax, B. Wendroff, Systems of conservation laws, in: Selected Papers Volume I, Springer, 2005, pp. 263–283.
[10] M.S. Arulampalam, S. Maskell, N. Gordon, T. Clapp, A tutorial on particle filters for online nonlinear/non-Gaussian Bayesian tracking, IEEE Trans. Signal Process. 50 (2) (2002) 174–188.
[11] T.R. Bewley, et al., Efficient grid-based Bayesian estimation of nonlinear low-dimensional systems with sparse. Non-Gaussian PDFs, Automatica 48 (7) (2012) 1286–1290, https://doi.org/10.1016/j.automatica.2012.02.039.
[12] B. Hanson, An instruction guide to gbees: grid-based Bayesian estimation exploiting sparsity, https://bhanson10.github.io/gbees.pdf, 2025. (Accessed 11 March 2025).
[13] B.L. Hanson, A.J. Rosengren, T.R. Bewley, T.A. Ely, Non-Gaussian recursive Bayesian filtering for outer planetary orbilander navigation, in: AAS/AIAA Space Flight Mechanics Meeting, 2025, p. 194.
[14] A.L. Davis, S. Guzik, X. Gao, On the parallel performance of a novel brick-based hash-table cfd library for distributed computing, in: AIAA SCITECH 2025 Forum, 2025, p. 1868.
[15] L.P.d. Castro, A.P. Pinheiro, V. Vilela, G.M. Magalhães, R. Serfaty, J.M. Vedovotto, Implementation of a hybrid Lagrangian filtered density function–large eddy simulation methodology in a dynamic adaptive mesh refinement environment, Phys. Fluids 33 (4) (2021).
[16] H. Ji, F.-S. Lien, F. Zhang, A GPU-accelerated adaptive mesh refinement for immersed boundary methods, Comput. Fluids 118 (2015) 131–147, https://doi.org/10.1016/j.compfluid.2015.06.011.
[17] K. Jaber, E.E. Essel, P.E. Sullivan, GPU-native adaptive mesh refinement with application to lattice Boltzmann simulations, Comput. Phys. Commun. 311 (2025) 109543, https://doi.org/10.1016/j.cpc.2025.109543.
[18] P. Colella, M.R. Dorr, J.A. Hittinger, D.F. Martin, High-order, finite-volume methods in mapped coordinates, J. Comput. Phys. 230 (8) (2011) 2952–2976.
[19] S.P. Veluri, C.J. Roy, E.A. Luke, Comprehensive code verification techniques for finite volume CFD codes, Comput. Fluids 70 (2012) 59–72, https://doi.org/10.1016/j.compfluid.2012.04.028.
[20] B. Diskin, J.L. Thomas, Notes on accuracy of finite-volume discretization schemes on irregular grids, Appl. Numer. Math. 60 (3) (2010) 224–226, https://doi.org/10.1016/j.apnum.2009.12.001.
[21] B.L. Hanson, A.J. Rosengren, T.R. Bewley, State estimation of chaotic trajectories: a higher-dimensional, grid-based, Bayesian approach to uncertainty propagation, in: AIAA SCITECH 2024 Forum, 2024, p. 0426.
[22] R. Courant, K. Friedrichs, H. Lewy, On the partial difference equations of mathematical physics, IBM J. Res. Dev. 11 (2) (1967) 215–234, https://doi.org/10.1147/rd.112.0215.
[23] NVIDIA-Corporation, About CUDA, https://developer.nvidia.com/about-cuda, 2024. (Accessed 19 November 2024).
[24] H.S. Tang, R.D. Haynes, G. Houzeaux, A review of domain decomposition methods for simulation of fluid flows: concepts, algorithms, and applications, Arch. Comput. Methods Eng. 28 (3) (2021) 841–873, https://doi.org/10.1007/s11831-019-09394-0.
[25] K. Karzhaubayev, L.-P. Wang, D. Zhakebayev, DUGKS-GPU: an efficient parallel GPU code for 3D turbulent flow simulations using discrete unified gas kinetic scheme, Comput. Phys. Commun. 301 (2024) 109216, https://doi.org/10.1016/j.cpc.2024.109216.
[26] W. Xue, C.W. Jackson, C.J. Roy, An improved framework of GPU computing for CFD applications on structured grids using OpenACC, J. Parallel Distrib. Comput. 156 (2021) 64–85, https://doi.org/10.1016/j.jpdc.2021.05.010.
[27] C.-C. Ye, P.-J.-Y. Zhang, Z.-H. Wan, R. Yan, D.-J. Sun, Accelerating CFD simulation with high order finite difference method on curvilinear coordinates for modern GPU clusters, Adv. Aerodyn. 4 (1) (2022) 7, https://doi.org/10.1186/s42774-021-00098-3.
[28] J. Berger, J. Marsha, J. Oliger, Adaptive mesh refinement for hyperbolic partial differential equations, J. Comput. Phys. 53 (3) (1984) 484–512.
[29] M. Berger, P. Colella, Local adaptive mesh refinement for shock hydrodynamics, J. Comput. Phys. 82 (1) (1989) 64–84, https://doi.org/10.1016/0021-9991(89)90035-1.
[30] P. Wang, T. Abel, R. Kaehler, Adaptive mesh fluid simulations on GPU, New Astron. 15 (7) (2010) 581–589, https://doi.org/10.1016/j.newast.2009.10.002.
[31] D. Beckingsale, W. Gaudin, A. Herdman, S. Jarvis, Resident block-structured adaptive mesh refinement on thousands of graphics processing units, in: 2015 44th International Conference on Parallel Processing, IEEE, Beijing, China, 2015, pp. 61–70.
[32] H.-Y. Schive, Y.-C. Tsai, T. Chiueh, GAMER: a GPU-accelerated adaptive mesh refinement code for astrophysics, Astrophys. J. Suppl. Ser. 186 (2) (2010) 457–484, https://doi.org/10.1088/0067-0049/186/2/457.
[33] X. Luo, L. Wang, W. Ran, F. Qin, GPU accelerated cell-based adaptive mesh refinement on unstructured quadrilateral grid, Comput. Phys. Commun. 207 (2016) 114–122, https://doi.org/10.1016/j.cpc.2016.05.018.
[34] A. Giuliani, L. Krivodonova, Adaptive mesh refinement on graphics processing units for applications in gas dynamics, J. Comput. Phys. 381 (2019) 67–90, https://doi.org/10.1016/j.jcp.2018.12.019.
[35] W. Raateland, T. Hädrich, J.A.A. Herrera, D.T. Banuti, W. Pałubicki, S. Pirk, K. Hildebrandt, D.L. Michels, DCGrid: an adaptive grid structure for memory-constrained fluid simulation on the GPU, Proc. ACM Comput. Graph. Interact. Tech. 5 (1) (2022) 1–14, https://doi.org/10.1145/3522608.
[36] NVIDIA-Corporation, CUDA C++ programming guide, https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html, 2024. (Accessed 19 November 2024).
[37] D.P. Mehta (Ed.), Handbook of Data Structures and Applications, Chapman & Hall/CRC Computer and Information Science Series, vol. 3, Chapman & Hall/CRC, Boca Raton, Fla, 2005.
[38] D.E. Knuth, The Art of Computer Programming, vol. 3, second edition, Addison-Wesley Publ, Reading (Mass.) London Manila [etc.], 1973.
[39] J.D. Cohen, Recursive hashing functions for n-grams, ACM Trans. Inf. Syst. 15 (3) (1997) 291–320, https://doi.org/10.1145/256163.256168.
[40] R.E. Ladner, M.J. Fischer, Parallel prefix computation, J. ACM 27 (4) (1980) 831–838, https://doi.org/10.1145/322217.322232.
[41] W. Hwu, D. Kirk, I. El Hajj, Programming Massively Parallel Processors: a Hands-on Approach, fourth edition, Elsevier: Morgan Kauffmann, Cambridge, MA, 2023.
[42] E.N. Lorenz, Deterministic nonperiodic flow, J. Atmos. Sci. 20 (2) (1963) 130–141, https://doi.org/10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.
[43] E.N. Lorenz, Predictability: a problem partly solved, in: Proc. Seminar on Predictability, vol. 1, Reading, 1996, pp. 40–58.
[44] A. Bhattacharyya, On a measure of divergence between two multinomial populations, Sankhya 7 (4) (1960) 401–406, http://www.jstor.org/stable/25047882.
[45] S. Kullback, R.A. Leibler, On information and sufficiency, Ann. Math. Stat. 22 (1) (1951) 79–86.
[46] D.W. Scott, Multivariate Density Estimation: Theory, Practice, and Visualization, 2nd edition, Wiley Series in Probability and Statistics, Wiley, Somerset, 2015.
[47] R.J. LeVeque, Finite Volume Methods for Hyperbolic Problems, vol. 31, Cambridge University Press, 2002.
[48] B. Medi, M. Amanullah, Application of a finite-volume method in the simulation of chromatographic systems: effects of flux limiters, Ind. Eng. Chem. Res. 50 (3) (2011) 1739–1748, https://doi.org/10.1021/ie100617c.
[49] H.F. Schwaiger, R.P. Denlinger, L.G. Mastin, Ash3d: a finite-volume, conservative numerical model for ash transport and tephra deposition, J. Geophys. Res., Solid Earth 117 (B4) (2012), https://doi.org/10.1029/2011JB008968.
[50] M. Chen, T. Guo, C. Chen, W. Xu, Data-driven arbitrary polynomial chaos expansion on uncertainty quantification for real-time hybrid simulation under stochastic ground motions, Exp. Tech. 44 (6) (2020) 751–762, https://doi.org/10.1007/s40799-020-00381-w.