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

Benjamin L. Hanson[a,c], 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*
[c] *Corresponding author, Email: blhanson@ucsd.edu, Tel: +1 (913) 626-5877,*

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 GPU execution. The algorithm is validated for correct convergence and accuracy, with performance evaluated across multiple 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.

Keywords:
Eulerian uncertainty propagation, Corner Transport Upwind, Dynamic gridding, Hashtables, CUDA, GPU

1. Introduction

Nonlinear uncertainty propagation methods can generally be classified as Kalman, Lagrangian, and/or Eulerian. The Kalman approach approximates uncertainty as Gaussian, or as a mixture of Gaussians. For linear dynamics and measurement models, Gaussian uncertainty remains Gaussian globally, but in the presence of nonlinearities, Kalman methods are suboptimal [1]. Analytical linearizations, in the case of the Extended Kalman Filter (EKF), and statistical linearizations, in the cases of the Unscented Kalman Filter (UKF) [2] and the Ensemble Kalman Filter (EnKF) [3], are utilized to more accurately represent nonlinearities, but both tend to diverge when Gaussian measurement corrections are relatively infrequent. Alternatively, Gaussian Mixture Models (GMMs) represent non-Gaussian uncertainty as a weighted superposition of Gaussians [4]. Certain GMM methods use splitting procedures triggered by entropy flags to increase the number of components in the superposition as true uncertainty becomes more non-Gaussian [5].

Langragian methods perform Sequential Monte Carlo (SMC) estimation on point mass representations of probability densities. Ensemble members are randomly drawn from an *a priori* distribution and then time-marched up to a measurement correction epoch via the true dynamics model. The simplest SMC method, Monte Carlo (MC) integration, attributes equal weight to each point [6]; this method prevents measurement corrections, limiting its applicability to uncertainty prediction. More sophisticated approaches assign weight based on an importance sampling distribution; these are known as Sequential Importance Sampling (SIS) methods, or, more commonly, particle filters [7]. For particle filters, measurement corrections are incorporated via weight adjustments and resampling procedures are utilized to avoid particle degeneracy [8, 9]. Hybrid Kalman/Langrangian methods attribute Gaussian kernels to each particle and update the associated moments according to the GMM formulation [10].

The Eulerian approach considers and updates probability at fixed points in space. For stochastic processes dominated by deterministic forces, the time-evolution of the full probability density function is a hyperbolic partial differential equation (PDE). Considerable effort has been put forth by the fluid mechanics community towards numerically solving these types of PDEs via finite difference/volume methods [11] which can be divided into two categories: semidiscrete and fully discrete. Semidiscrete methods, like the Essentially Non-Oscillatory (ENO) and Weighted Essentially Non-Oscillatory

(WENO) schemes [12], use adaptive stencils to create smooth interpolations of probability across discontinuities, then use Runge-Kutta time stepping to march the system of semidiscrete equations. Fully discrete methods use fluxes defined at grid cell interfaces to update discretized probability at grid cell centers. The Lax-Wendroff and Godunov methods are two, first-order accurate examples [13, 14], but higher-order corrections are necessary to achieve second-order accuracy, and flux limiters ensure these methods are total variation diminishing [15].

Of the three defined approaches, Eulerian methods are generally the least explored for high-dimensional nonlinear uncertainty propagation. This is most likely due to both the finite domain limitation for standard grid-based methods as well as the high computational intensity that accompanies the application of fluids-based finite volume methods to ($n > 3$)-dimensional probability density time-marching. However, Eulerian approaches do not require splitting procedures to maintain accuracy, do not succumb to particle degeneracy, and are extremely robust for chaotic dynamics models, underdetermined measurement models, and infrequent correction updates.

A novel Eulerian approach known as Grid-based Bayesian Estimation Exploiting Sparsity (GBEES) dynamically allocates grid cells in regions of non-negligible probability, unlocking propagation over all of phase space [16]. However, the algorithm's $\mathcal{O}(n^2)$ time complexity poses computational challenges for high-dimensional systems. In this paper, we optimize the computational architecture of GBEES 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. Implementing the algorithm in CUDA

These improvements result in GBEES-GPU, an efficient, high-dimensional parallel GPU algorithm for nonlinear uncertainty propagation. In Section 1 we outlined the landscape of nonlinear uncertainty propagation methods and defined our contributions towards the computational optimization of GBEES. In Section 2 the finite volume formulation underlying GBEES is extended to n-dimensions. Section 3 outlines the improvements made to the CPU implementation, while Section 4 describes its adaptation to the CUDA architecture. Section 5 presents the use cases employed for validation and testing, including the evaluation of the correct convergence and accuracy. 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 $d\boldsymbol{W}(t) = \boldsymbol{\xi}(t)dt$ is a Wiener process, meaning $\boldsymbol{\xi}(t)$ is zero-mean, uncorrelated white noise (i.e., $\mathbb{E}[\boldsymbol{\xi}(t)] = \boldsymbol{0}$ and $\mathbb{E}[\boldsymbol{\xi}(t+\tau)\boldsymbol{\xi}^T(t)] = \boldsymbol{\delta}(\tau)$). 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 $\boldsymbol{x} = (x_1, \ldots, x_n)$, $f_j(\boldsymbol{x}, t)$ is the j^{th} component of $\boldsymbol{f}(\boldsymbol{x}, t)$, $Q_{j\ell}(\boldsymbol{x}, t)$ is the $(j, \ell)^{\text{th}}$ component of $Q(\boldsymbol{x}, t) = \boldsymbol{q}(\boldsymbol{x}, t)\boldsymbol{q}^T(\boldsymbol{x}, t)$. If $Q(\boldsymbol{x}, t) > 0$, 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:

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

where $f_j'\big(p(\boldsymbol{x}, t)\big) = f_j(\boldsymbol{x}, t)p(\boldsymbol{x}, t)$. 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.

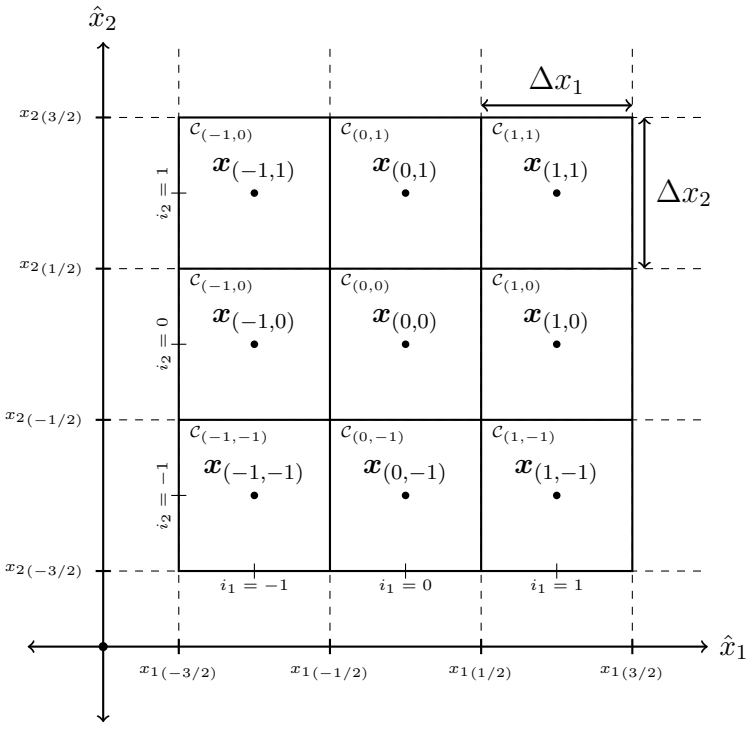

Figure 1: 2D schematic depicting the notation of GBEES.

2.1. Fully discrete flux-differencing methods

Consider a hyperrectangular grid cell of the form

$$\mathcal{C}_{(\boldsymbol{i})} = \prod_{j=1}^{n} [x_{j_{(i_j-1/2)}}, x_{j_{(i_j+1/2)}}] \tag{5}$$

where $\boldsymbol{i} = (i_1, \ldots, i_n)$ is the grid cell index vector corresponding to the grid cell center coordinate vector $\boldsymbol{x}_{(\boldsymbol{i})}$. The grid width of $\mathcal{C}_{(\boldsymbol{i})}$ in the x_j-direction is then

$$\Delta x_{j_{(\boldsymbol{i})}} = x_{j_{(i_j+1/2)}} - x_{j_{(i_j-1/2)}}; \tag{6}$$

for a uniform grid, $\Delta x_{j_{(\boldsymbol{i})}}$ is equal for all $\boldsymbol{i}$ (since a uniform grid is utilized in this work, the grid width is referenced as Δx_j for the remainder of the paper). Additionally,

$$\boldsymbol{x}_{(\boldsymbol{i} \pm \hat{\imath}_j/2)} = \boldsymbol{x}_{(\boldsymbol{i})} \pm \frac{\Delta x_j}{2} \hat{x}_j \tag{7}$$

is the coordinate vector a half-grid width forward/backward relative to $\boldsymbol{x}_{(\boldsymbol{i})}$ in the x_j-direction (Fig. 1 displays Eqs. (5)-(7) schematically). From Eq. (3), p is assumed to be conserved over $\mathcal{C}_{(\boldsymbol{i})}$, thus the integral of p varies only due to flux across the boundaries of $\mathcal{C}_{(\boldsymbol{i})}$:

$$\tfrac{d}{dt}\textstyle\int_{\mathcal{C}_{(\boldsymbol{i})}} p(\boldsymbol{x}_{(\boldsymbol{i})},t)d\boldsymbol{x} = \sum_{j=1}^{n}\left(\int_{\mathcal{C}^{\pm}_{(\boldsymbol{i}_{\sim i_j})}}\left[f_j'\big(p(\boldsymbol{x}_{(\boldsymbol{i}+\hat{\imath}_j/2)},t)\big) - f_j'\big(p(\boldsymbol{x}_{(\boldsymbol{i}-\hat{\imath}_j/2)},t)\big)\right]d\boldsymbol{x}_{\sim j}\right), \quad (8)$$

where

$$\begin{aligned}
\boldsymbol{i}_{\sim j} &= (i_1,\ldots,i_{j-1},i_{j+1},\ldots,i_n),\\
d\boldsymbol{x} = \prod_{j=1}^{n} dx_j, &\quad d\boldsymbol{x}_{\sim j} = \prod_{\substack{\ell=1\\ \ell\neq j}}^{n} dx_\ell,\\
\mathcal{C}^{\pm}_{(\boldsymbol{i}_{\sim i_j})} &= [x_{j(i_j\pm 1/2)}] \times \prod_{\substack{\ell=1\\ \ell\neq j}}^{n}[x_{\ell(i_\ell-1/2)}, x_{\ell(i_\ell+1/2)}];
\end{aligned}$$

$\mathcal{C}^{\pm}_{(\boldsymbol{i}_{\sim i_j})}$ is the $(n-1)$-dimensional grid cell interface a half-step forward/backward in the x_j-direction at $x_j = x_{j(i_j\pm 1/2)}$ with normal vector pointing parallel/antiparallel to $\hat{x}_j$. Integrating Eq. (8) from $t^{(k)}$ to $t^{(k+1)}$ and dividing by the grid cell area leads to the fully discrete flux-differencing method

$$P^{(k+1)}_{(\boldsymbol{i})} = P^{(k)}_{(\boldsymbol{i})} - \sum_{j=1}^{n}\frac{\Delta t}{\Delta x_j}\left[F_j{}^{(k)}_{(\boldsymbol{i}+\hat{\imath}_j/2)} - F_j{}^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j/2)}\right] \quad (9)$$

where $P^{(k+1)}_{(\boldsymbol{i})}$ is the discrete, updated probability cell average at grid cell $\mathcal{C}_{(\boldsymbol{i})}$, with

$$F_j{}^{(k)}_{(\boldsymbol{i}\pm\hat{\imath}_j/2)} \approx \frac{1}{\Delta t\prod_{\substack{\ell=1\\ \ell\neq j}}^{n}\Delta x_\ell}\int_{t^{(k)}}^{t^{(k+1)}}\int_{\mathcal{C}^{\pm}_{(\boldsymbol{i}_{\sim i_j})}} f_j'\big(p(\boldsymbol{x}_{(\boldsymbol{i}\pm\hat{\imath}_j/2)},t)\big)d\boldsymbol{x}_{\sim j}dt.$$

The numerical fluxes $F_j{}^{(k)}$ parallel to the $(n-1)$-dimensional cell faces are approximated via a Godunov-type finite volume method known as Corner Transport Upwinding (CTU).

2.2. The Corner Transport Upwind 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 set of grid cell index vectors in the grid, in all directions $j = 1, \ldots, n$, the upwind flux in each direction x_j at time step k is calculated:

$$F^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} = f^{+}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} P^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j)} + f^{-}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} P^{(k)}_{(\boldsymbol{i})}, \tag{10}$$

where

$$\begin{aligned} f^{+}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} &= \max\left(f^{(k)}_{j\ (\boldsymbol{i}-\hat{\imath}_j/2)}, 0\right), \\ f^{-}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} &= \min\left(f^{(k)}_{j\ (\boldsymbol{i}-\hat{\imath}_j/2)}, 0\right), \\ f^{(k)}_{j\ (\boldsymbol{i}-\hat{\imath}_j/2)} &= f_j(\boldsymbol{x}_{(\boldsymbol{i}-\hat{\imath}_j/2)}, t^{(k)}). \end{aligned}$$

Eq. (10) only considers probability flowing normal to the grid cell interface, but in general, probability may flow oblique to the interfaces of $\mathcal{C}_{(\boldsymbol{i})}$. To obtain second-order accuracy, we must account for this with flux corrections.

We consider $\mathcal{C}^{-}_{(\boldsymbol{i}_{\sim i_1})}$, the $(n-1)$-dimensional grid cell interface a half-step backward in the x_1-direction at $x_1 = x_{1(i_1-1/2)}$. Generally, the direction of advection at this interface may not be perpendicular to this interface; thus, probability may propagate in any of the four ordinal directions depending on the signs of the components of $\boldsymbol{f}^{(k)}_{(\boldsymbol{i}-\hat{\imath}_1/2)}$, as shown schematically in Fig. 2. For second-order accuracy, this corresponds to $4(n-1)$ possible updates to neighboring fluxes in the $(n-1)$-directions, excluding the x_1-direction. 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 $\boldsymbol{i} \in \mathcal{I}$, in all directions x_j, the high-resolution correction terms are added to the (DCU + CTU)-calculated fluxes:

$$F^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} \leftarrow F^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} + \tfrac{1}{2}\left|f^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)}\right|\left(1 - \tfrac{\Delta t}{\Delta x_j}\left|f^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)}\right|\right)\tfrac{P^{(k)}_{(\boldsymbol{i})} - P^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j)}}{\Delta x_j}\,\phi\big(\theta^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j/2)}\big) \tag{11}$$

where

$$\theta^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j/2)} = \begin{cases} \big(P^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j)} - P^{(k)}_{(\boldsymbol{i}-2\hat{\imath}_j)}\big)/\big(P^{(k)}_{(\boldsymbol{i})} - P^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j)}\big) & \text{if } f^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} \geq 0, \\ \big(P^{(k)}_{(\boldsymbol{i}+\hat{\imath}_j)} - P^{(k)}_{(\boldsymbol{i})}\big)/\big(P^{(k)}_{(\boldsymbol{i})} - P^{(k)}_{(\boldsymbol{i}-\hat{\imath}_j)}\big) & \text{if } f^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)} < 0, \end{cases},$$

$$\phi(\theta) = \max\Big(0, \min\big[(1+\theta)/2, 2, 2\theta\big]\Big);$$

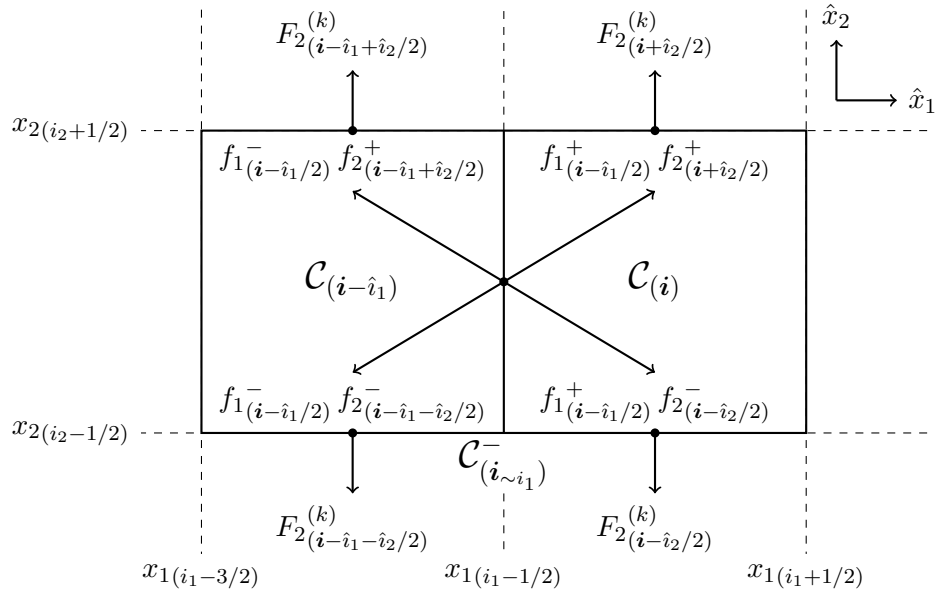

Figure 2: (x_1, x_2)-plane of an n-dimensional grid depicting the notation of the CTU method.

Algorithm 1 Corner Transport Upwind

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

for $\boldsymbol{i} \in \mathcal{I}$ **do**
  for $j = 0$ to n **do**
    $F^* \leftarrow \frac{\Delta t}{2\Delta x_j}\big(P_{(\boldsymbol{i})}^{(k)} - P_{(\boldsymbol{i}-\hat{\imath}_j)}^{(k)}\big)$
    for $\ell = 0$ to n, $\ell \neq j$ **do**
      $F_{\ell(\boldsymbol{i}+\hat{\imath}_\ell/2)}^{(k)} \leftarrow F_{\ell(\boldsymbol{i}+\hat{\imath}_\ell/2)}^{(k)} - f_{j(\boldsymbol{i}-\hat{\imath}_j/2)}^{+} f_{\ell(\boldsymbol{i}+\hat{\imath}_\ell/2)}^{+} F^*$
      $F_{l(\boldsymbol{i}-\hat{\imath}_\ell/2)}^{(k)} \leftarrow F_{\ell(\boldsymbol{i}-\hat{\imath}_\ell/2)}^{(k)} - f_{j(\boldsymbol{i}-\hat{\imath}_j/2)}^{+} f_{\ell(\boldsymbol{i}-\hat{\imath}_\ell/2)}^{-} F^*$
      $F_{\ell(\boldsymbol{i}-\hat{\imath}_j+\hat{\imath}_\ell/2)}^{(k)} \leftarrow F_{\ell(\boldsymbol{i}-\hat{\imath}_j+\hat{\imath}_\ell/2)}^{(k)} - f_{j(\boldsymbol{i}-\hat{\imath}_j/2)}^{-} f_{\ell(\boldsymbol{i}-\hat{\imath}_j+\hat{\imath}_\ell/2)}^{+} F^*$
      $F_{\ell(\boldsymbol{i}-\hat{\imath}_j-\hat{\imath}_\ell/2)}^{(k)} \leftarrow F_{\ell(\boldsymbol{i}-\hat{\imath}_j-\hat{\imath}_\ell/2)}^{(k)} - f_{j(\boldsymbol{i}-\hat{\imath}_j/2)}^{-} f_{\ell(\boldsymbol{i}-\hat{\imath}_j-\hat{\imath}_\ell/2)}^{-} F^*$
    end for
  end for
end for

$\phi(\theta)$ is a monotonized central difference limiter used for representing probability discontinuities with sharper resolution [17]. After the fluxes have been

calculated and plugged into Eq. (9), the discretized PDF is normalized

$$P_{(\boldsymbol{i})}^{(k+1)} = \left(\sum_{\boldsymbol{i} \in \mathcal{I}} P_{(\boldsymbol{i})}^{(k+1)} \right)^{-1} P_{(\boldsymbol{i})}^{(k+1)}. \tag{12}$$

Together, Eq. (10), Algorithm 1, and Eqs. (11) and (12) form a second-order accurate finite volume method employed by GBEES for numerically time-marching a discretized, n-dimensional PDF.

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. However, the legacy algorithm architecture contains 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 $\mathcal{C}_{(\boldsymbol{i})}$ 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 [18], 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 [19]

$$\Delta t \left(\sum_{j=1}^{n} \frac{f_j}{\Delta x_j} \right) \leq C_{\max}$$

must be satisfied in order for the method to be stable, where $C_{\max} = 1$ for explicit methods (as GBEES is an explicit finite volume method, we assume

$C_{\max} = 1$ for the remainder of the paper). In general, because the advection is variable across spatial coordinates, Δt must be small enough such that this condition holds for $\boldsymbol{i} \in \mathcal{I}$:

$$\Delta t \leq \left(\sum_{j=1}^{n} \frac{f_{j(\boldsymbol{i}-\hat{\imath}_j/2)}}{\Delta x_j} \right)^{-1}. \tag{13}$$

The legacy implementation of GBEES employed a static, over-conservative time step to ensure stability for the entire propagation period. The new implementation of GBEES finds the minimum allowable Δt such that Eq. (13) is true for $\boldsymbol{i} \in \mathcal{I}$ at time step k:

$$\Delta t^{(k)} = \min_{\boldsymbol{i} \in \mathcal{I}} \left[\left(\sum_{j=1}^{n} \frac{f^{(k)}_{j(\boldsymbol{i}-\hat{\imath}_j/2)}}{\Delta x_j} \right)^{-1} \right]. \tag{14}$$

Because of the dynamic nature of the grid, $\Delta t^{(k)}$ may change depending on where in phase space probability is focused. Implementing Eq. (14) 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. 3, only the *downwind* grid cells are created; this process saves 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,

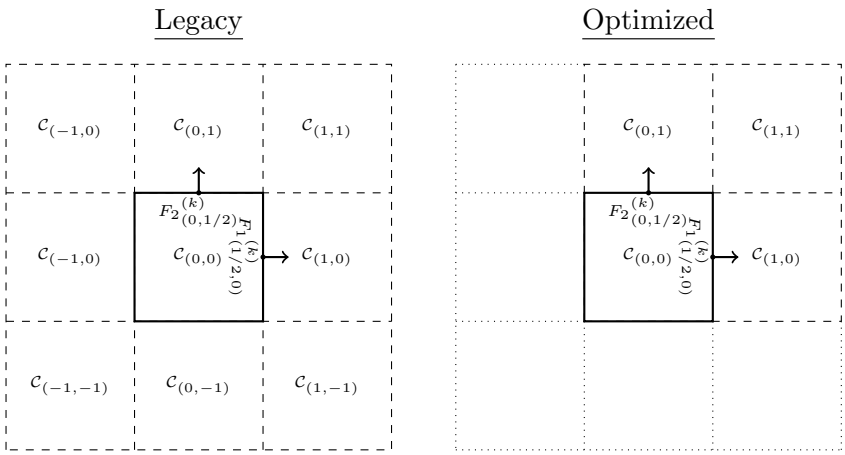

Figure 3: 2D schematic demonstrating the differences in the growing procedure for the legacy and optimized implementations of GBEES. Solid border cells are those with probability above threshold, dashed border cells are those set to be created during the growing procedure, and dotted border cells are neglected.

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. 4 shows that this requires the algorithm to check 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 describes the specifics of the CUDA architecture [20] and the optimization strategies employed.

The CUDA architecture is designed for parallel processing, with GPUs containing multiple Streaming Multiprocessors (SMs). Each SM can execute groups of threads organized into blocks, which are further divided into smaller units of 32 threads called warps. Warps execute instructions in a single-instruction, multiple-thread (SIMT) fashion, meaning all threads in a warp perform the same instruction simultaneously on different data.

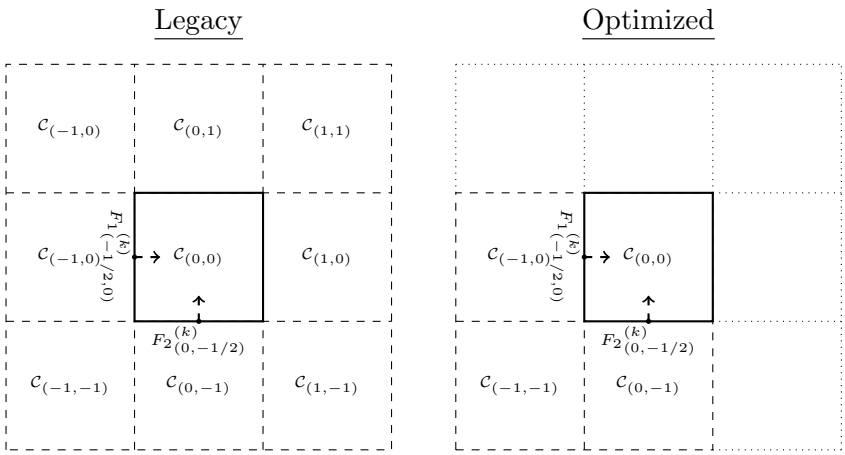

Figure 4: 2D schematic demonstrating the differences in the pruning procedure for the legacy and optimized implementations of GBEES. Solid border cells are those with probability below threshold, dashed border cells are those checked during the pruning procedure, and dotted border cells are neglected.

CUDA GPUs also feature a layered memory hierarchy. Global (or device) memory provides large storage capacity but has higher latency. Each SM has faster, limited shared memory that is accessible only to threads within a block, enabling efficient data sharing. Additionally, each thread has its own private registers, the fastest type of memory, used for intermediate calculations.

As described in Section 2, the GBEES method requires a dynamic grid in which cells are added or removed throughout the integration steps. This dynamic nature prevents establishing a fixed mapping between execution threads and cells. In traditional finite volume software using a static grid, the grid is partitioned into subdomains —each one including also a boundary with additional halo cells— and these subdomains are then assigned to thread blocks on the GPU [21]. 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. This dynamic structure 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 by the dynamic grid is provided by the Cooperative Kernel abstraction in CUDA [22]. A Coop-

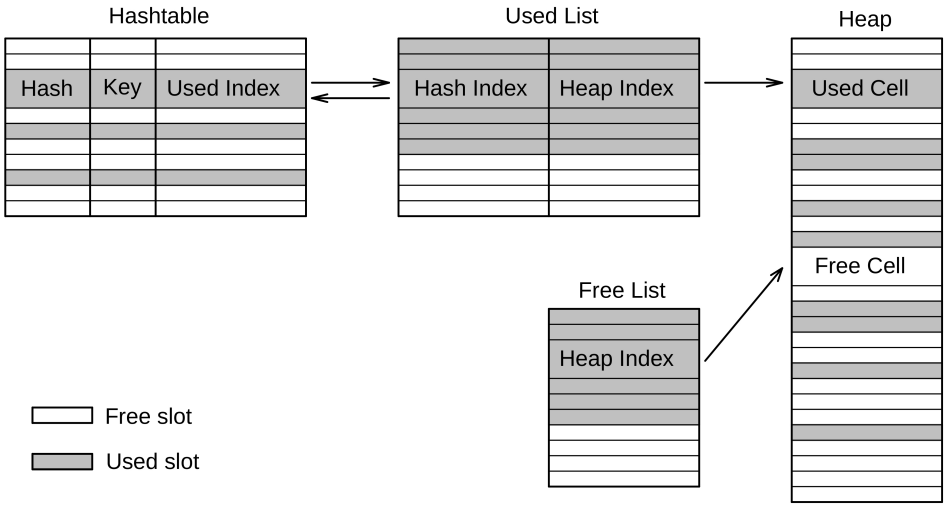

Figure 5: Grid data structure. Notice that the Used List and the Free List are maintained compact, with all the used slots at the beginning and all the free slots at the end.

erative 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.

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 memory structure for the grid depicted in Fig. 5.

The 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 state coordinates $\boldsymbol{i} = (i_1, \ldots, i_n)$. 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 not used 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. 5. 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 [23]. With a well-randomized hash function, the expected number of probes during an unsuccessful search is [24]

$$(1 + 1/(1 - \alpha)^2)/2$$

and for a successful search

$$(1 + 1/(1 - \alpha))/2.$$

The open addressing scheme requires marking elements as deleted [23]. In the GBEES method, 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. Fig. 6 shows collision graphs for different hash functions, with each plot representing the number of collisions per entry in a 2D projection for the 3D Lorenz '63 model —one of the test cases included in this study. Based on these patterns, the BuzHash algorithm, also known as hashing by cyclic polynomial, was selected and implemented as described in [25].

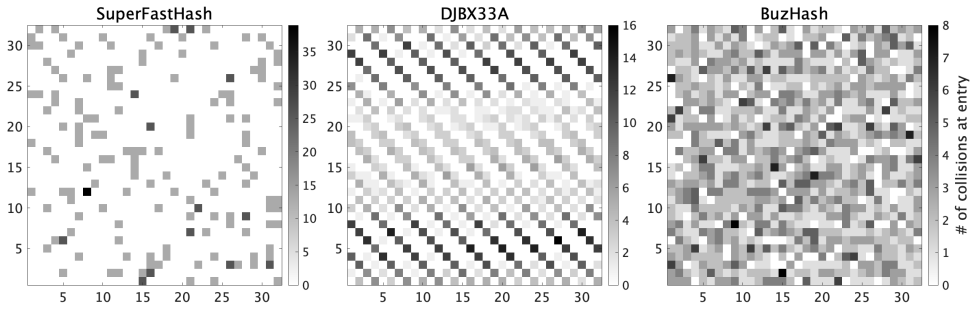

Figure 6: Number of collisions for different hash functions represented as 2D projections, where the capacity of the hashtable is set to $1024 = 32 \times 32$. For these examples, the hashtable stores the initial grid from the Lorenz '63 example, explained further in Section 5.1.1.

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. (14)—, computing the DCU and CTU —Eq. (10), Algorithm (1), and Eq. (11)—, probability distribution normalization —Eq. (12)—, and applying new measurements —Eq. (4)—.

The second category involves grid-wide operations, specifically the grid growth and grid pruning. These operations modifies 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 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. Specifically:

- Growth is first performed along the forward axis of all dimensions. A global synchronization barrier is then executed.
- Growth is subsequently performed along the backward axis of all dimensions, followed by another global synchronization step.
- Finally, 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, identifying them as candidates for pruning. The next step involves performing a parallel prefix sum operation [26, 27] 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 2 Concurrent Cell Creation

Require: The Used List and the Free List are compact.

Obtain the hashtable slot by applying the hash function to the cell key.
loop
    Check if the hashtable slot is free using an atomicCAS() operation. If it is free, reserve it with the RESERVED flag; if not, obtain the current Used Index.
    if The existing cell is the same **then**
        break
    end if
    if An empty slot is reached **then**
        Reserve a Used List slot using an atomicAdd() operation on the list size.
        Obtain a free heap slot from the Free List using an atomicDec() operation on the list size.
        Update the Hashtable and Used List contents.
        Update the heap content and complete cell initialization with a callback function.
    end if
    Move to the next hashtable slot.
end loop

Algorithm 3 Grid Prune Operation

Mark the negligible cells in the heap.
Perform a prefix sum process of the Used List in shared memory.
Complete the prefix sum of the Used List in global memory.
Compact the Used List and update the Free List.
Rehash the Hashtable.

Ensure: To 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 following Eq. (12). Parallel scan is applied during the prune operation to compact the Used List. These techniques are widely recognized as standard methods [27], 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 consider the following use cases

- The Lorenz '63 model (three-dimensional)
- 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 uncertainty propagation methods because of the highly non-Gaussian behavior exhibited [28]. The three-

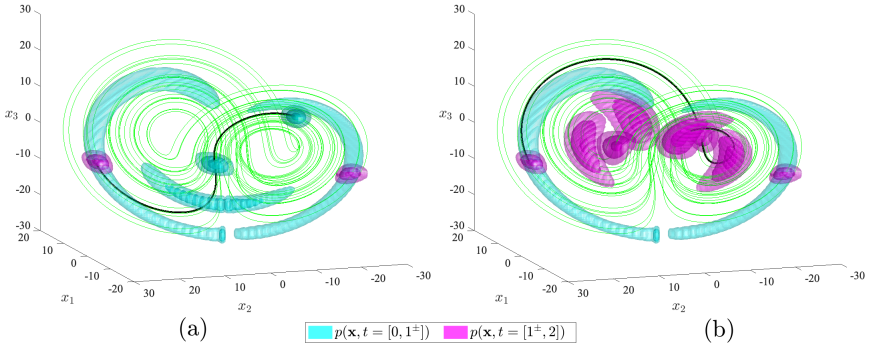

Figure 7: PDF isosurfaces governed by the Lorenz '63 model in (x_1, x_2, x_3)-space at $p = 0.607$, $p = 0.135$, and $p = 0.011$ with $\Delta x_j = 0.5$ for $j = 1, 2, 3$. On the left (a), the isosurfaces are at $t = 0$, $t = 1/3$, $t = 2/3$,and $t = 1^{\pm}$ and on the right (b), the isosurfaces are at $t = 1^{\pm}$, $t = 4/3$, $t = 5/3$ and $t = 2$.

dimensional state and equations of motion are defined as

$$\boldsymbol{x} = \begin{bmatrix} x_1 \\ x_2 \\ x_3 \end{bmatrix}, \quad \frac{d\boldsymbol{x}}{dt} = \boldsymbol{f}(\boldsymbol{x}) = \begin{bmatrix} \sigma(x_2 - x_1) \\ -x_2 - x_1 x_3 \\ -bx_3 + x_1 x_2 - br \end{bmatrix},$$

where $(\sigma, b, r) = (4, 1, 48)$ results in the system being chaotic. In Fig. 7, a 3D Gaussian PDF is initialized at $\boldsymbol{x}^{(0)} = (-11.5, -10, 9.5)$ with standard deviation $\sigma_{x_j} = 1$ for $j = 1, 2, 3$. The uncertainty is then continuous-time propagated using GBEES until $t^{(1)} = 1$, where 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 continuous-time propagated till $t = 2$, when the simulation ends. Fig. 7 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, 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 [16], the CPU-optimized and GPU implementations of GBEES are validated on the Lorenz

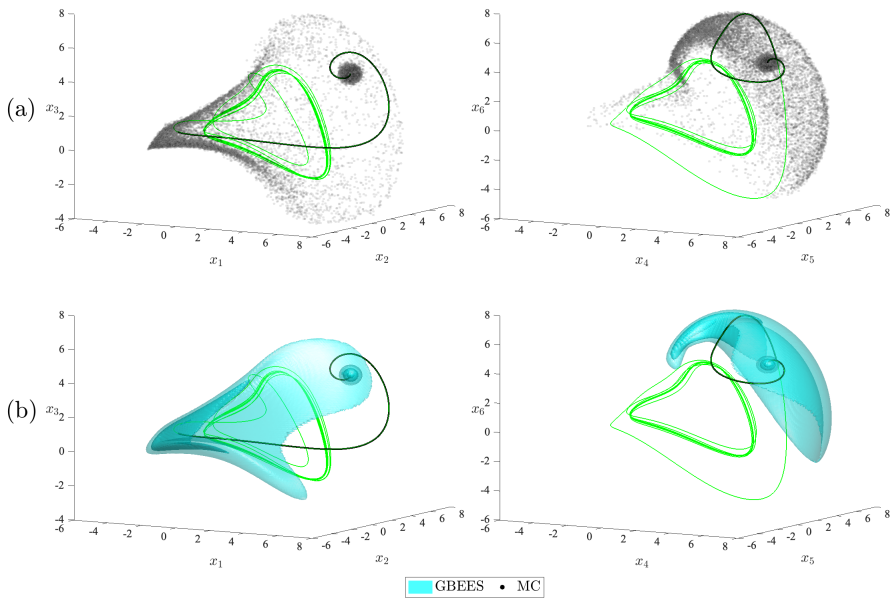

Figure 8: PDF representations in (x_1, x_2, x_3)- and (x_4, x_5, x_6)-spaces governed by the Lorenz '96 model with $\Delta x_j = 0.01$ for $j = 1, \ldots, 6$ and $F = 4$. On top (a), the MC point-mass distributions with 10,000 particles at $t = 0$ and $t = 1.3$ and on bottom (b), the GBEES isosurfaces at $p = 0.607$, $p = 0.135$, and $p = 0.011$ for $t = 0$ and $t = 1.3$.

'96 model, a generalized dynamical system that exhibits chaotic behavior [29]. 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 $\boldsymbol{x}^{(0)} = (F + 0.5, F, \ldots, F)$ where $F = 4$, with standard deviation $\sigma_{x_j} = 0.02$ for $j = 1, \ldots, 6$. The uncertainty is then propagated using GBEES until $t = 1.3$. No measurement update is performed in this simulation. To validate the accuracy of GBEES qualitatively, an MC simulation with identical initial conditions is plotted, depicted in Fig.

8(a). The 3D PDFs in the (x_1, x_2, x_3)- and (x_4, x_5, x_6)-spaces, shown in Fig. 8(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:

$$p(x_1, x_2, x_3, t) = \int_{\min(x_6)}^{\max(x_6)} \int_{\min(x_5)}^{\max(x_5)} \int_{\min(x_4)}^{\max(x_4)} p(\boldsymbol{x}, t) dx_4 dx_5 dx_6,$$
$$p(x_4, x_5, x_6, t) = \int_{\min(x_3)}^{\max(x_3)} \int_{\min(x_2)}^{\max(x_2)} \int_{\min(x_1)}^{\max(x_1)} p(\boldsymbol{x}, t) dx_1 dx_2 dx_3.$$

5.2. Convergence and accuracy

In addition to the qualitative validation against the MC simulation shown in Figs. 7 and 8, a quantitative validation is conducted by comparing the results to a propagation with a very small step size, which serves as the reference "true" solution for this test.

This test has two primary objectives. First, to ensure the GBEES implementation converge correctly, as demonstrated by decreasing errors relative to the reference propagation when step 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 [22]. Due to the non-associativity of floating-point arithmetic caused by the rounding errors, small variations in the computed probability distribution are expected.

The errors in the probability distribution are assessed using two metrics. The first metric measures the sum of the absolute differences between the probability values in each cell. Since the mesh is equispaced, there is no need to assign different weights to the cells. Additionally, because the total sum of the probabilities in the cells is 1, this metric is already normalized. The error for this metric is calculated as follows:

$$E(t) = \sum_{\boldsymbol{i} \in \mathcal{I}} |p(\boldsymbol{x}_{(\boldsymbol{i})}, t) - p^0(\boldsymbol{x}_{(\boldsymbol{i})}, t)|,$$

where $p^0(\boldsymbol{x}_{(\boldsymbol{i})}, t)$ represents the discrete probability distribution obtained from the reference propagation.

The second metric is the Bhattacharyya Coefficient [30] defined as

$$BC(t) = \sum_{\boldsymbol{i} \in \mathcal{I}} \sqrt{p(\boldsymbol{x}_{(\boldsymbol{i})}, t) p^0(\boldsymbol{x}_{(\boldsymbol{i})}, t)},$$

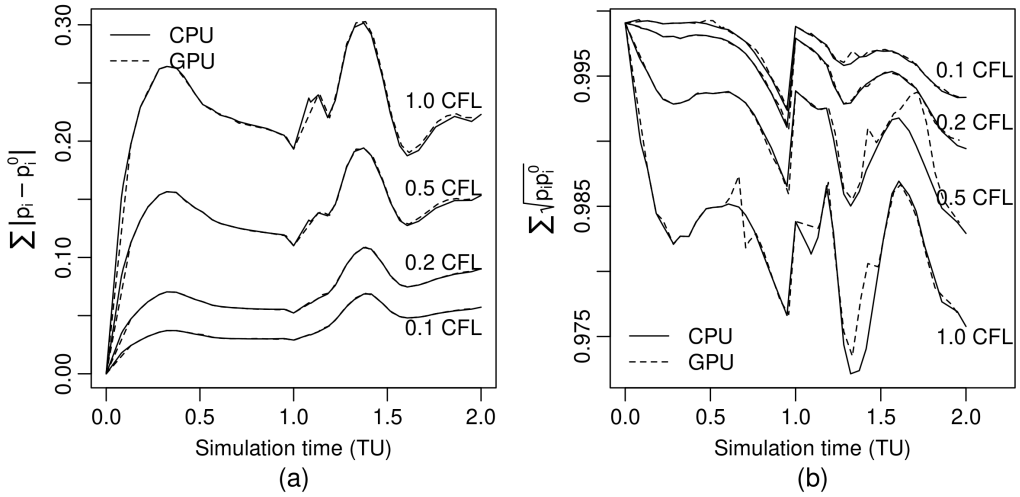

Figure 9: Comparison of the discrete probability distributions over simulated time with a reference propagation of step size 0.01×CFL. On the left (a), the comparison is based on the absolute probability differences between cells $E(t)$. On the right (b), the comparison utilizes the Bhattacharyya Coefficient $BC(t)$.

that measures the similarity between two probability distributions with a value of 1 indicating that the distributions coincide and 0 when the distributions are entirely distinct.

Using these metrics, the reference propagation of a variable step size of 0.01×CFL was compared to CPU and GPU propagations with step sizes of 1.0, 0.5, 0.2, and 0.1 times CFL.

The results of these comparisons are shown in Fig. 9. The curves demonstrate proper convergence, 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 influence of other error sources in the simulation, such as step size variation.

6. Performance

Performance improvements for both the new CPU and CUDA versions are evaluated using the same validation cases described in Section 5. Their computational effort is summarized in Table 1. 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

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

Table 1: Computational burden for the Lorenz '63 and Lorenz '96 models with a variable step size corresponding to a 1.0×CFL and a cell size equal to half the standard deviation in each dimension of the first measurement.

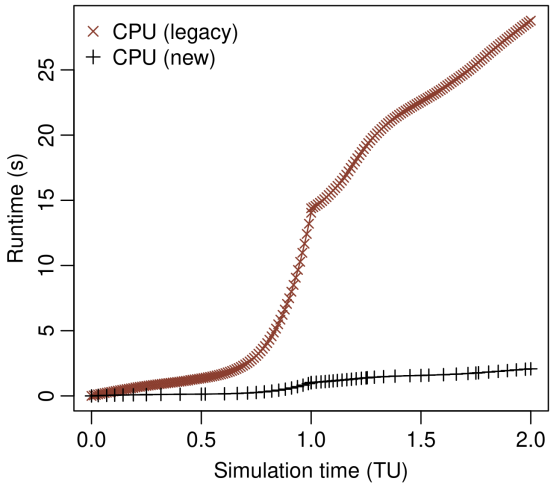

Figure 10: Runtime comparison between the legacy and the new CPU version for the Lorenz '63 model.

phase space, while the Lorenz '96 uses a six-dimensional one.

Fig. 10 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 Bhattacharyya coefficients, 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 2. 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.

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: 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

Table 2: 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.

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 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.

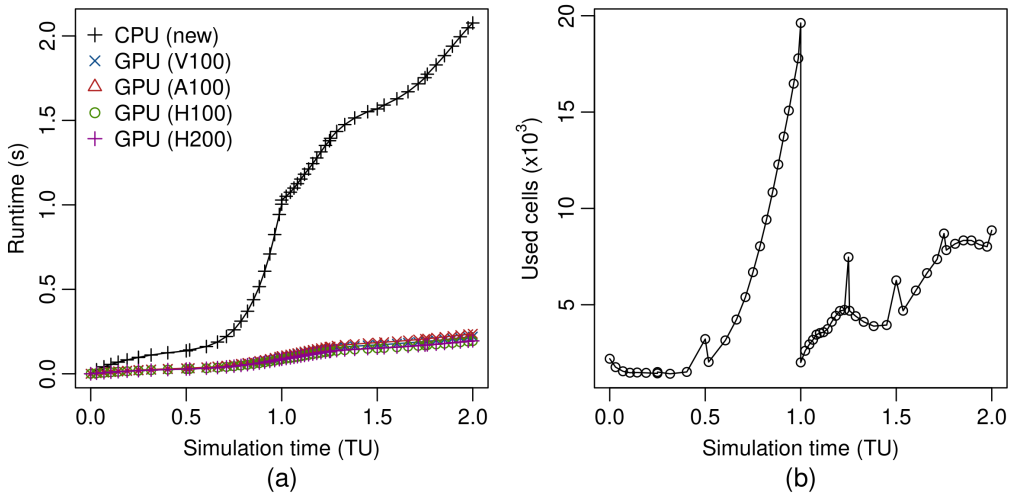

Figure 11: Runtime comparison between the new CPU version and GBEES-GPU on various GPU devices (a) and number of used cells (b) for the Lorenz '63 model.

Fig. 11 represents, for the Lorenz '63 model, the runtime comparision between the new CPU version and the CUDA implementation running on different GPU devices. Fig. 11(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. 11(b). The drop in the number of cells at $t = 1.0$ TU corresponds to a discrete measurement update.

The graph in Fig. 11(a) shows a significant performance boost from parallelizing and executing the algorithm on the GPU. Table 2 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. 12 first presents the runtime comparison between the new CPU version and the

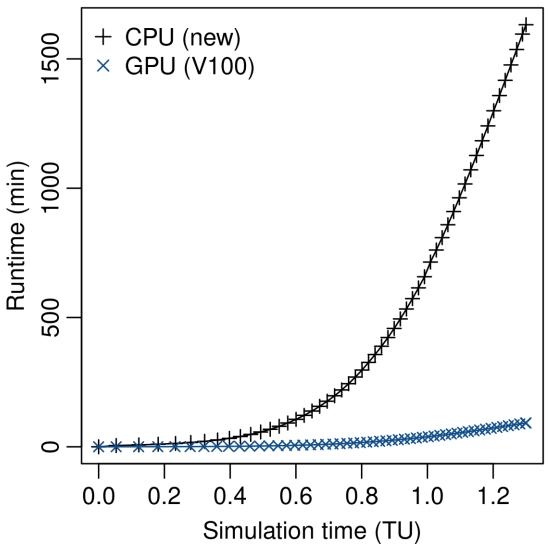

Figure 12: Runtime comparison between the new CPU version and the GBEES-GPU execution on a V100 GPU decvice for the Lorenz '96 model.

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 3. 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. 13 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.

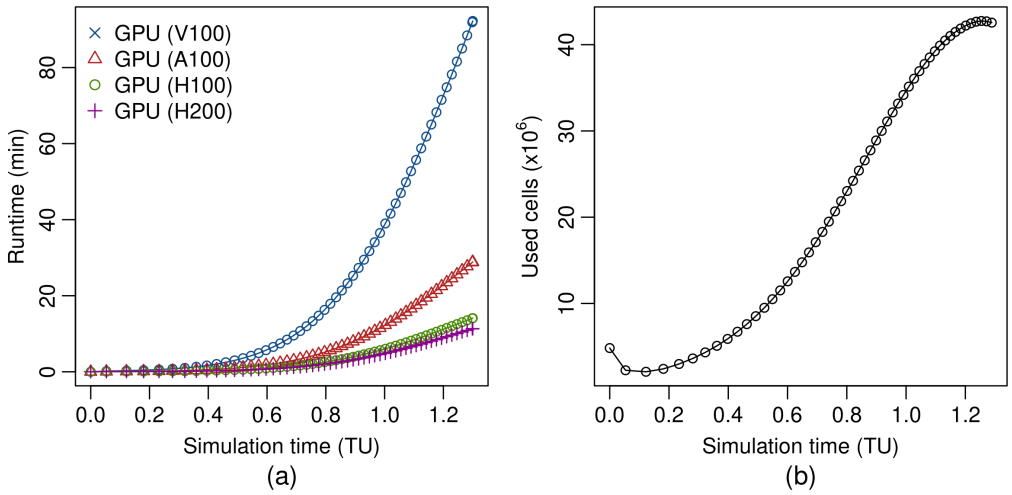

Figure 13: Runtime comparison between the new CPU version and GBEES-GPU on various GPU devices (a) and number of used cells (b) for the Lorenz '96 model.

Device	Runtime (s)	Cells/s	Speed-up
CPU-optimized: Apple M2 MAX	97927	≈0.3M/s	1
GPU 1: 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

Table 3: 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.

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 CPU implementation of GBEES, the main data structure was changed from a linked list to a hashtable, a CFL-minimized adaptive time step 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 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 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.

For validation, we test the two novel implementations on the Lorenz '63 model, a three-dimensional chaotic system utilized in the original GBEES paper. Two quantitative metrics are analyzed: the sum of the absolute differences and the Bhattacharyya Coefficient. For a truth model, we time-march an initially Gaussian PDF at $1/100^{\text{th}}$ the expected stable time step size subject to the chaotic dynamics. Both metrics indicate that the propagated probability distribution converges to the truth as the time step decreases for both the CPU-optimized and GPU implementations. For this low-dimensional example, the performance increase relative to the CPU implementation is one order of magnitude for the CPU-optimized implementation and two orders of magnitude for the GPU 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. The application of finite volume methods to this dimensionality is uncommon, but necessary for the uncertainty propagation of second-order dynamical systems. For this example, we validate with a densely-populated MC simulation, qualitatively confirming that the point mass distributions and probability isosurfaces match at the start and end epochs. The high dimensionality of the system highlights the efficacy of the parallelized algorithm; for the Lorenz '96 system, the GPU implementation has a performance increase of two orders of magnitude relative to the CPU-optimized implementation, implying a 1000-fold increase in performance relative to the original CPU implementation.

CRediT authorship contribution statement

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

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

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-hash` 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.

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 $\times$ 3.7GHz, 4 cores $\times$ 3.4 GHz. L2 cache size 36 MB.

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

- GPU 1: 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.

References

[1] R. E. Kalman, A New Approach to Linear Filtering and Prediction Problems, Journal of Basic Engineering 82 (1) (1960) 35–45. doi:10.1115/1.3662552.

[2] S. J. Julier, et al., Unscented filtering and nonlinear estimation, Proceedings of the IEEE 92 (3) (2004) 401–422. doi:10.1109/JPROC.2003.823141.

[3] G. Evensen, The ensemble kalman filter: Theoretical formulation and practical implementation, Ocean dynamics 53 (2003) 343–367. doi:10.1007/s10236-003-0036-9.

[4] H. W. Sorenson, D. L. Alspach, Recursive bayesian estimation using gaussian sums, Automatica 7 (4) (1971) 465–479. doi:10.1016/0005-1098(71)90097-5.

[5] K. J. DeMars, et al., Entropy-based approach for uncertainty propagation of nonlinear dynamical systems, Journal of Guidance, Control, and Dynamics 36 (4) (2013) 1047–1057. doi:10.2514/1.58987.

[6] J. M. Hammersley, K. W. Morton, Poor man's monte carlo, Journal of the Royal Statistical Society Series B: Statistical Methodology 16 (1) (1954) 23–38. doi:10.1111/j.2517-6161.1954.tb00145.x.

[7] N. J. Gordon, D. J. Salmond, A. F. Smith, Novel approach to nonlinear/non-gaussian bayesian state estimation, in: IEE proceedings F (radar and signal processing), Vol. 140, IET, 1993, pp. 107–113. doi:10.1049/ip-f-2.1993.0015.

[8] M. K. Pitt, et al., Filtering via simulation: Auxiliary particle filters, Journal of the American statistical association 94 (446) (1999) 590–599. doi:10.1080/01621459.1999.10474153.

[9] J. V. Candy, Bootstrap particle filtering, IEEE Signal Processing Magazine 24 (4) (2007) 73–85. doi:10.1109/MSP.2007.4286566.

[10] J. L. Anderson, et al., A monte carlo implementation of the nonlinear filtering problem to produce ensemble assimilations and forecasts, Monthly weather review 127 (12) (1999) 2741–2758. doi:10.1175/1520-0493(1999)127<2741:AMCIOT>2.0.CO;2.

[11] R. J. Leveque, Finite difference methods for ordinary and partial differential equations: steady-state and time-dependent problems, SIAM, 2007.

[12] B. Cockburn, C.-W. Shu, C. Johnson, E. Tadmor, C.-W. Shu, Essentially non-oscillatory and weighted essentially non-oscillatory schemes for hyperbolic conservation laws, Springer, 1998.

[13] P. Lax, B. Wendroff, Systems of conservation laws, in: Selected Papers Volume I, Springer, 2005, pp. 263–283.

[14] S. K. Godunov, A difference scheme for numerical solution of discontinuous solution of hydrodynamic equations, Math. Sbornik 47 (1959) 271–306.
URL https://cir.nii.ac.jp/crid/1573387449783562752

[15] P. K. Sweby, High resolution schemes using flux limiters for hyperbolic conservation laws, SIAM journal on numerical analysis 21 (5) (1984) 995–1011. doi:10.1137/0721062.

[16] 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. doi:10.1016/j.automatica.2012.02.039.

[17] B. Van Leer, Towards the ultimate conservative difference scheme. iv. a new approach to numerical convection, Journal of computational physics 23 (3) (1977) 276–299. doi:10.1016/0021-9991(77)90095-X.

[18] 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. doi:10.2514/6.2024-0426.

[19] R. Courant, K. Friedrichs, H. Lewy, On the partial difference equations of mathematical physics, IBM journal of Research and Development 11 (2) (1967) 215–234. doi:10.1147/rd.112.0215.

[20] NVIDIA-Corporation, About CUDA, accessed on 19 November 2024 (2024).
URL https://developer.nvidia.com/about-cuda

[21] 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, Computer Physics Communications 301 (2024) 109216. doi:10.1016/j.cpc.2024.109216.

[22] NVIDIA-Corporation, CUDA C++ Programming Guide, accessed on 19 November 2024 (2024).
URL https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html

[23] D. P. Mehta (Ed.), Handbook of data structures and applications, no. 3 in Chapman & Hall/CRC computer and information science series, Chapman & Hall/CRC, Boca Raton, Fla, 2005.

[24] D. E. Knuth, The art of computer programming, second ed. Edition, Vol. 3, Addison-Wesley publ, Reading (Mass.) London Manila [etc.], 1973.

[25] J. D. Cohen, Recursive hashing functions for n-grams, ACM Transactions on Information Systems (TOIS) 15 (3) (1997) 291–320. doi:10.1145/256163.256168.

[26] R. E. Ladner, M. J. Fischer, Parallel Prefix Computation, J. ACM 27 (4) (1980) 831–838. doi:10.1145/322217.322232.

[27] W. Hwu, D. Kirk, I. El Hajj, Programming massively parallel processors: a hands-on approach, fourth ed. Edition, Elsevier : Morgan Kauffmann, Cambridge, MA, 2023.

[28] E. N. Lorenz, Deterministic nonperiodic flow, Journal of atmospheric sciences 20 (2) (1963) 130–141. doi:10.1175/1520-0469(1963)020<0130:DNF>2.0.CO;2.

[29] E. N. Lorenz, Predictability: A problem partly solved, in: Proc. Seminar on predictability, Vol. 1, Reading, 1996, pp. 40–58. doi:10.1017/CBO9780511617652.004.

[30] A. Bhattacharyya, On a Measure of Divergence between Two Multinomial Populations, Sankhyā: The Indian Journal of Statistics 7 (4) (1960) 401–406.
URL http://www.jstor.org/stable/25047882